Algorithm Engineering for fundamental Sorting and Graph Problems

genehmigte

Dissertation

von

Vitaly Osipov

aus Sverdlovsk

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
von der
Fakultät für Informatik
des
Karlsruher Instituts für Technologie

Tag der mündlichen Prüfung:	14. April 2014
Erster Gutachter:	Prof. Dr. Peter Sanders
Zweite Gutachter:	Prof. Dr. Ulrich Meyer

Abstract

Fundamental Algorithms build a basis knowledge for every computer science undergraduate or a professional programmer. It is a set of basic techniques one can find in any (good) coursebook on algorithms and data structures. Thus, if one checks the contents of one such recent book by Mehlhorn and Sanders [105] he would find that we address most of the algorithmic problems covered in the book. In particular, we present current advances in such classical topics as *sorting*, *graph traversals*, *shortest paths*, *minimum spanning trees*, graph *matchings* and *partitioning*

One can argue that all these problems are folklore with optimal worst-case behaviour. It is certainly true as long as the assumptions we use to derive the runtime bounds hold for a particular problem or available architecture at hand. In most of the coursebooks it is a *Random Access Machine* (RAM) model assuming that the input fits into the main memory of the computer and access time is unform throughout the size of the available memory. Unfortunately, it is often not the case for the real world architectures that posses a number of memory hierarchy layers including several levels of cache, main memory and an external hard drive. For instance, classical graph traversal algorithms become orders of magnitude slower as soon as data does not fit into the main memory and the operating system is forced to use a hard drive. Another problem for the classical coursebook algorithms is a rapidly growing parallelism. Currently it ranges from several cores in commodity home workstations (*multicores*) up to hundreds of cores available in *Graphical Processing Units* (GPUs). Classical approaches often become inefficient or even inapplicable in such settings. Even being on the safe side, that is in sequential RAM model, the worst-case optimality alone does not guarantee the best performance in practice. The reason for it is that the worst cases are rare in practice and thus the theoretically worse algorithms often perform better than their worst-case optimal counterparts. Clever heuristics often have further positive impact on the performance.

In this thesis we try to close the gap between theoretically worst-case optimal classical algorithms and the real-world circumstances one face under the assumptions imposed by the data size, limited main memory or available parallelism.

Contents

1. Introduction **7**
1.1. Our main contributions. 8

2. Models for Algorithm Design **10**
2.1. Random Access Machine 10
2.2. External Memory Model 12
2.3. Parallel Random Access Machine 13
2.4. General Purpose Computation on GPUs 14

3. Algorithm Engineering Infrastruture **16**
3.1. Algorithm Engineering Methodology 16
3.2. Machine Configurations 17
3.3. Data Analysis 17

4. Sorting **19**
4.1. Divide and Conquer Approaches 19
4.2. Sorting on GPU 20
4.3. GPU Sample Sort 21
4.3.1. Algorithm Design 23
4.3.2. Implementation Details for the Original Algorithm 24
4.3.3. Implementation Details for the Improved Algorithm 25
4.3.4. Special Case: 16- and 32-way Distribution 26
4.3.5. Sorting Small Buckets. 26
4.3.6. Tuning Architecture-Dependent Parameters 27
4.3.7. Experimental Study of Original Algorithm on Tesla 30
4.3.8. Experimental Study of Algorithms on Fermi and Performance Portability 34
4.3.9. Evaluation of a Special-case Treatment for 16-and 32-way Distribution 36
4.4. Conclusion 39

5. Suffix Sorting **40**
5.1. Preliminaries . . . 40
5.2. Related Work . . . 41
5.3. Prefix-Doubling Algorithms . . . 42
5.4. Induced Sorting . . . 43
5.5. Parallel Suffix Array Construction for Shared Memory Architectures . . . 44
5.5.1. Algorithm Design . . . 46
5.5.2. Experimental Study . . . 49
5.5.3. Discussion . . . 50
5.6. Suffix Array Construction in External Memory . . . 51
5.6.1. Algorithm Design . . . 51
5.6.2. Splitting Large Tuples. . . . 53
5.6.3. I/O Analysis of eSAIS with Split Tuples . . . 56
5.6.4. Experimental Study . . . 58

6. Breadth First Search on Massive Graphs **63**
6.1. Algorithms . . . 64
6.1.1. Related Work . . . 65
6.1.2. Our Contribution . . . 66
6.1.3. Improvements of MR_BFS and MM_BFS_R . . . 66
6.2. Algorithm Design of MM_BFS_D . . . 67
6.2.1. A Heuristic for Maintaining the Pool . . . 69
6.2.2. Experimental Study . . . 70
6.2.3. Results with Heuristic. . . . 74
6.3. Conclusion . . . 74
6.3.1. Discussion . . . 75

7. Single Source Shortest Paths on Massive Graphs **76**
7.1. Overview . . . 77
7.2. Algorithm Design . . . 78
7.2.1. Experimental Study . . . 82
7.2.2. Early Results on Flash Memory. . . . 88
7.3. Conclusions . . . 89

8. Minimum Spanning Tree **90**
8.1. Overview . . . 90
8.2. Kruskal's Algorithm . . . 92
8.3. Algorithm Design . . . 93
8.3.1. Results for Random Edge Weights. . . . 93
8.3.2. Implementation. . . . 95
8.3.3. Parallelization. . . . 95
8.3.4. More Sophisticated Variants. . . . 96
8.3.5. Experiments . . . 97
8.4. Conclusions . . . 101

9. Graph Matching **103**
9.1. Global Path Algorithm . . . 104
9.1.1. Filter-GPA . . . 105
9.1.2. Analysis for Random Edge Weights . . . 106
9.2. Massively Parallel Matchings . . . 109
9.3. Local Max . . . 110
9.3.1. Sequential Model . . . 111
9.3.2. CRCW PRAM Model. . . . 111
9.3.3. MapReduce Model. . . . 111
9.3.4. External Memory Models. . . . 111
9.3.5. $\mathcal{O}(\log^2 n)$ work-optimal CREW solution . . . 111
9.4. Implementation and Experimental Study . . . 113
9.4.1. Sequential Speed and Quality . . . 114
9.4.2. GPU Implementation . . . 117
9.5. Conclusions And Future Work . . . 119

10. Graph Partitioning **120**
10.1. Multilevel Graph Partitioning . . . 120
10.2. n-Level Graph Partitioning . . . 122
10.2.1. Local Search Strategy . . . 123
10.2.2. Trial Trees . . . 124
10.2.3. Experimental Study . . . 125
10.3. Conclusion . . . 130

11. Discussion **132**

Bibliography **133**

A. Complete Data Sets **147**

B. List of Publications **154**

C. Zusammenfassung **157**

CHAPTER 1

Introduction

Fundamental Algorithms build a basis knowledge for every computer science undergraduate or a professional programmer. It is a set of basic techniques one can find in any (good) coursebook on algorithms and data structures. Thus, if one checks the contents of one such recent book by Mehlhorn and Sanders [105] he would find that we address most of the algorithmic problems covered in the book. In particular, we present current advances in such classical topics as *sorting*, *graph traversals*, *shortest paths*, *minimum spanning trees*, graph *matchings* and *partitioning*

One can argue that all these problems are folklore with optimal worst-case behaviour. It is certainly true as long as the assumptions we use to derive the runtime bounds hold for a particular problem or available architecture at hand. In most of the coursebooks it is a *Random Access Machine* (RAM) model assuming that the input fits into the main memory of the computer and access time is unform throughout the size of the available memory. Unfortunately, it is often not the case for the real world architectures that posses a number of memory hierarchy layers including several levels of cache, main memory and an external hard drive. For instance, classical graph traversal algorithms become orders of magnitude slower as soon as data does not fit into the main memory and the operating system is forced to use a hard drive. In this case we say that the problem is stated in *External Memory* (EM) settings.

Another problem for the classical coursebook algorithms is a rapidly growing parallelism. Currently it ranges from several cores in commodity home workstations (*multicores*) up to hundreds of cores available in *Graphical Processing Units* (GPUs). Classical approaches often become inefficient or even inapplicable in such settings.

Even being on the safe side, that is in sequential RAM model, the worst-case optimality alone does not guarantee the best performance in practice. The reason for it is that the worst cases are rare in practice and thus the theoretically worse algorithms often perform better than their worst-case optimal counterparts. Clever heuristics often have further positive impact on the performance.

In this thesis we try to close the gap between theoretically worst-case optimal classical algorithms and the real-world circumstances one face under the assumptions imposed by the data size, limited main memory or available parallelism.

1.1. Our main contributions.

Sorting. We consider two problems – a general-purpose *comparison-based sorting* and a special case of string sorting – a *suffix array construction*.

General purpose sorting is a basic building block for many advanced algorithms, and graph algorithms in particular we consider later. *Suffix array* is a text index data structure that is widely used in text processing, compression, web search and computational biology. We propose massively parallel algorithms and their implementation on commodity Graphical Processing Units (GPUs) for both of this problems with a world leading performance [92, 125]. For the suffix array construction we also give an external memory algorithm, which is about factor two faster than its counterpart [22].

Having sorting at hand, we turn to the classical graph algorithms.

Massive Graph Breadth First Search Traversal As soon as the input graph does not fit into the internal memory of the computer, classical algorithm become unviable. We we show how to traverse graphs in external memory settings in *hours* or at most *days* where the classical coursebook approaches would take at least *months* [6].

Single Source Shortest Paths. Using ideas from our external memory graph traversal algorithm we extend it to another classical problem of computing single source shortest paths in graphs. We propose the first implementation of a semi-external memory algorithm making the problem solvable for graphs, which allow storing one bit per vertex in internal memory. Thus, the problem becomes viable for graphs, whose size exceeds the size of available memory by a large margin [109].

Minimum Spanning Tree. In this case we stay in the RAM model and propose a simple heuristic that improves a classical Kruskal's algorithm. Our heuristic avoids sorting of edges that are "obviously" not in the MST. For random graphs with random edge weighs we can also show that the expected runtime of our algorithm is $\mathcal{O}(m + n \log n \log \frac{m}{n})$, that is linear for not too sparse graphs. The algorithm has also very good practical performance over the entire range of edge densities. Moreover, an equally simple parallelization seems to be the currently best practical algorithm on multicore machines [127].

Graph Matchings. We show that the the same idea of avoiding resorting of edges that are guaranteed not to be part of the solution can be extended to another classical problem – a *maximum weight matching* and considerably improve the Global Path Algorithm (GPA) [103]. The performance evaluation confirms very good practical performance as in sequential as in multicore settings.

Unfortunately due to its limited parallelizability GPA is not suitable for massively parallel architectures. Therefore we turn to even simpler *local max* algorithm [66]. We prove that the algorithm performs linear work and allows implementation in several models of parallel computation. We also evaluate its implementation for GPUs [23]. Though both our approaches are approximation algorithms with the worst-case guarantee of 1/2 we show that for real world instances they are only a few percent off from the optimal solution.

Graph Partitioning. Next, we show that even NP-complete *graph partitioning* problem can be solved efficiently using fairly simple approximation and local optimization techniques [126]. Indeed, we propose an algorithm based on an extreme idea of approximating initial graph by building a graph hierarchy of up to potentially n levels. We start with the initial graph and obtain the next level by contracting a *single* edge. This allows a very fine-grained contraction of a graph and therefore leads to a better abstraction of the initial graph on the higher levels of the hierarchy. Experimental evaluation showed that the algorithm scales well for large inputs and produces the best known partitioning results for many real world graphs. For example, in the well known Walshaw's benchmark tables we achieve 155 improvements dominating the entries for large graphs.

CHAPTER 2

Models for Algorithm Design

Algorithm design requires a precise model of the architecture the algorithm is intended for. Due to complexity of real-world architectures it is often counterproductive to cover all architectural details in the model. A large variety of parameters make algorithm design and analysis complicated. It is often a good practice to simplify the model by concentrating on a few essential parameters and neglecting the rest that does not make considerable impact on performance. There a number of widely-used machine models that proved to reflect the predicted theoretical behaviour in practice. In the following chapters we will consider two major models that are relevant for us: a *random access machine* (RAM) or *von Neumann model* [120] and its simple *extension external memory model* [3, 169]. We will also consider more exotic one describing NVidia Graphic Processing Units (GPUs) codenamed *compute unified device architecture* (CUDA).

2.1. Random Access Machine

Random access machine is a variant of von Neumann's Model [120] introduced by Sheperdson and Sturgis [155] in 1963. In the following we describe essential RAM's features, for details refer to [105]. RAM has a single processing unit with uniform unbounded *main memory*, meaning that all memory cell accesses cost the same constant amount of time. Each memory cell $C[i], i = [0, \ldots, \infty)$ is capable of storing a *word*, an integer, whose bit length is logarithmic in the input size. And there can be only a finite number of memory cells occupied at any point of time. Besides the main memory, RAM possesses a number of *registers* $R_1, R_2, \ldots, R_k$, and is capable of performing the following operations on them:

- $R_l := C[R_r]$ *loads* a word from the memory cell indexed by a word contained in R_r into R_l

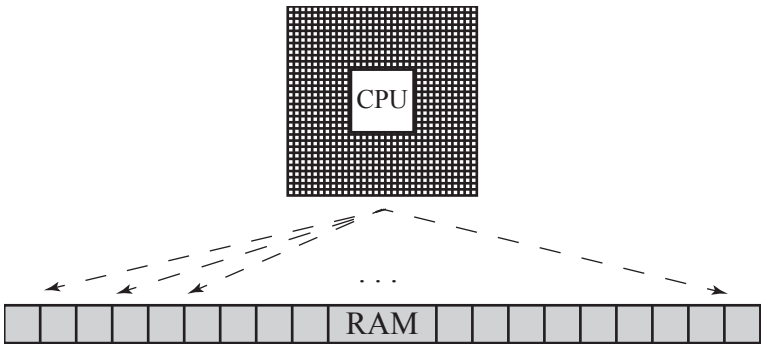


Figure 2.1.: RAM model

- $C[R_l] := R_r$ *stores* a word contained in R_r into the memory cell indexed by a word contained in R_l.
- binary operations $R_b{=}R_l \odot R_r$, where $\odot$ can be an

  arithmetic $+,-,\cdot$

  bitwise | (or), & (and), >> (shift right), << (shift left), $\oplus$ (exclusive or, xor)

  comparison $\leq, \geq, <, >$ resulting in $true$ (1) or $false$ (0)

  logical $\vee, \wedge$ operating on logical $true$ (1) and $false$ (0)

  casting reinterpreting a word contained in a register as a floating-point number.

  operation.
- unary operations $R_l = \odot R_r$, where $\odot$ can be $-$, $\neg$ (logical not), $\sim$ (bitwise not)
- $R_l := C$ assigns a constant value to R_l
- $JZ\ j, R_r$ if R_r is 0 continues execution at the memory cell j
- $J\ j$ continues execution at the memory cell j

The success of this model is due to its simple design and powerful features. Von Neumann's model proved to be an elegant and accurate enough abstraction of a vast variety of much more complex modern hardware that did not even exists in the time it was invented. It still serves as a standard programming model for most of industrial applications.

Though modern architectures are quite far from the simplistic von Neumann's model, introduction of a more accurate model would make it much more complex and difficult to handle.

Unfortunately, in some cases, such refinements are impossible to avoid. This is the case in particular, when the favourable assumption of unbounded main memory does not hold in real life. As soon as the input data does not fit into the main memory, RAM is not capable anymore to predict the algorithm's performance in practice.

In the next section we describe a slight extension of RAM called *external memory model* that overcomes this shortcoming and targets scenarios we pictured above.

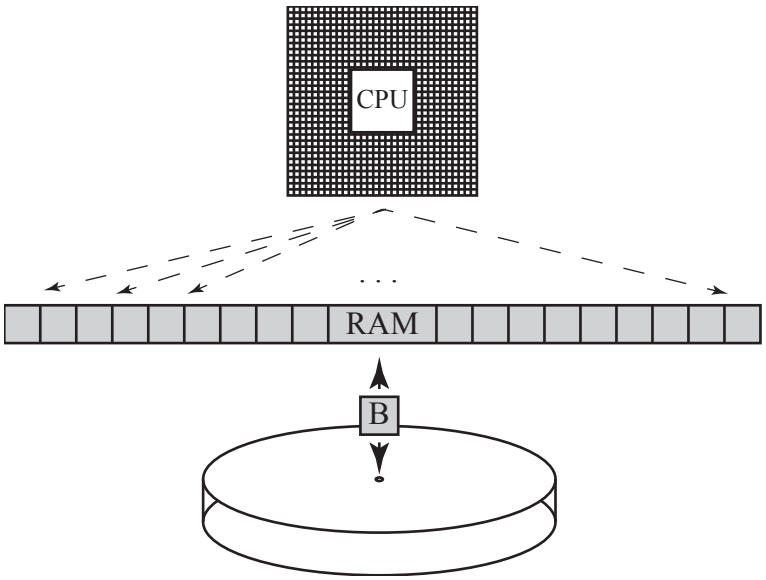


Figure 2.2.: External Memory model

2.2. External Memory Model

In contrast to abstract RAM the main memory of a real-world architecture is not only bounded, but is also much more complex than assumed by the cost model of RAM. Thus, a current microprocessor has a file of *registers* supporting several parallel accesses per clock cycle, *first level (L1)* cache memory with 1 or 2 accesses per clock, *second level (L2)* cache with a latency of 10 clock cycles [105]. Besides that some microprocessors have a *third level cache (L3)* that is made of fast static random access memory cells. The *main memory* consists of dynamic random access memory cells with access latency up to dozens of nanoseconds. Finally, a computer system have a *hard drive*, which is often used by the operating system to swap data that does not fit into the main memory. The access time latency of a hard drive of up to 10 ms is 10^7 higher than the access time to a register. However, both, main memory and a hard drive, are optimized for block-wise accesses. Thus, accessing 1 byte from the main memory can be only twice slower than reading a block of 16 contiguous bytes. And as soon as the hard drive starts reading contiguous data the transfer rate reaches about 50–70 MBs per second. Hence, instead of a single level uniform cost main memory assumed by RAM, we have a multilevel memory hierarchy with large variety of latencies and nontrivial strategies of migrating data between different levels [105].

To reflect the limited amount of the main memory, different access latencies between different memory hierarchy levels and block-wise access pattern Aggarwal, Vitter and Shriver [3, 169] extended RAM and introduced a notion of an *external memory model*. The model considers only two levels of the hierarchy, but these can be any: as, for instance, cache/main memory as main memory/hard drive. The latter configuration has significantly more pronounced effect, since accessing data in main memory is orders of magnitude faster than reading data from a hard drive.

In contrast to RAM there is a limited M words of fast *internal* (main in terms of RAM)

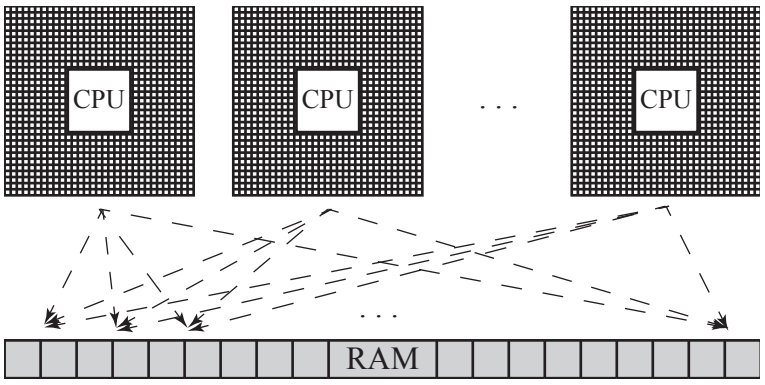


Figure 2.3.: PRAM model

memory. Processing of data in internal memory is similar to RAM. Besides internal memory there is slow *external memory*. Transfers of data between internal and external memory happen in blocks of B contiguous words and are called input/output operations, or I/Os for short. The input data is assumed to be considerably larger than the size of internal memory and objective is to minimize the number of I/Os besides internal memory operations.

Accessing data on a hard drive consists of seek time t_s to position the hard drive's head and t_B transfer time to move the data. Though, by choosing an appropriate B, we can eliminate t_s from the model. Let B be the number of words, such that the time t_B to transfer them into the main memory is approximately t_s. Now, if we access less than B words and count it is as one I/O, that is as accessing the whole block of B contiguous words, we account for at most factor two more time than needed. If we access $D > B$ contiguous words and count it as $\lceil D/B \rceil$ I/Os, we are also at most factor two off from what we really spent.

Though being pretty young compared to RAM, this model proved to produce accurate results in practice. There is a wide range of scientific papers from purely theoretical results to implementations that are successfully used in practical applications, see an overview by Vitter [168]. Trying to further extend the model and introduce further parameters such as, for instance, hard drive caches, dependencies of the seek time and the current position of the reading head would make the model much more complicated and, thus, would be probably counter productive. One would need to adapt parameters for different manufactures and take care of complex hardware details that are even sometimes not publicly available.

2.3. Parallel Random Access Machine

In time when a parallel processor is a commonplace, the need for a model capturing growing parallelism becomes essential for algorithm design. Interestingly, an extension of RAM *called parallel random access machine* (PRAM), stems from late 70s [51, 60, 149], when parallel architectures were rather exotic. PRAM consists of several sequential processors, each having its own private memory, accompanied with *global memory* shared among processors for communication. In one unit of time each processor can access either its private memory or global memory. There exist several PRAM models differentiating by the mean they handle

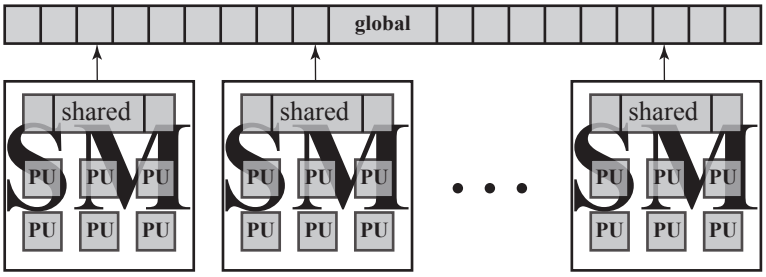


Figure 2.4.: CUDA architecture

parallel accesses to common global memory locations. Thus, EREW PRAM does not allow parallel reads or writes from the same memory location. CREW PRAM allows parallel reads, but no parallel writes. While CRCW PRAM allows both, as parallel reads as parallel writes.

The way PRAM resolves write conflicts leads to further classification. In the *common* model all processors writing simultaneously to the same memory cell write the same value *Arbitrary* model only one of the processors participating in parallel write succeeds. The algorithm should work correctly independently on the processor that succeeds. *Priority* model assumes that there exists a linear ordering on processors, and the smallest processor succeeds in a simultaneous write. Despite this diversity different models do not differ significantly in parallel performance. Thus, an algorithm for the strongest CRCW PRAM in the priority model can be simulated by the weakest EREW PRAM at the cost of $\mathcal{O}(\log p)$ time, where p is the number of processors. Analogously, as long as the number of processors is large enough, PRAM in common model can be simulated by PRAM in priority model with no asymptotical overhead.

Unfortunately, the growth of memory and number of processors causes increase in memory access time, thus making PRAM impossible to realize in practice. Nevertheless it serves a working horse for many algorithmists since decades and is probably the most widely accepted model for parallel processing.

2.4. General Purpose Computation on GPUs

With the growth of parallelism available in emerging architectures, *graphical processing units* (GPUs) in particular, the necessity for further refinement of existing models became apparent.

In 2008 NVidia introduced Tesla, a unified graphic and computing architecture, along with a programming model codenamed *compute unified device architecture* (CUDA).

In the same year Apple Inc. in cooperation with another CPU, GPU, embedded-processor and software companies like AMD, IBM, Quallcomm, Intel and Invidia proposed an alternative technology called Open Computing Language (OpenCL for short). OpenCL is not limited to GPU, but targets a wide range of shared memory parallel architectures.

In the following we will describe mainly CUDA, since it is the main technology we use in this thesis. Though most of the high-level CUDA components showed in Figure 2.4 are also present in OpenCL.

Though there is no established theoretical model for designing algorithms for GPUs, physical design and optimization recommendations allow us to draw a parallel to some of the features and objectives inherent to the existing models like PRAM or external memory.

Analogous to PRAM's processors we have p *thread blocks* in CUDA that share global memory for communication between them. In contrast to PRAM, each thread block is not a sequential processor, but a t-threaded processor, where threads behave in a SIMD fashion executing the same single instruction in one unit of time. Wether a thread block is able to realize the full parallelism depends on the availability of data in thread's private registers at the point of time the instruction is ready to be executed. To achieve maximum bandwidth, threads must read continuous chunks of global memory. Thus, we can assume that each thread block has to load the data in blocks of size t, which make it similar to external memory model. Besides global memory, each thread block possess a limited private memory, called *shared memory*. In terms of external memory model, the global memory can be viewed as external and shared as internal memory. Though, in contrast to external memory model, the processing of data in shared memory should be done in SIMD fashion. Namely, one should ensure that (a) the execution paths of different threads do not diverge and (b) parallel load of data by t threads into registers should happen from t independent shared memory banks. As long as several threads request different data from the same bank these requests get serialized and thus the hardware is not able to realize full SIMD width parallelism.

Implementation The hardware implements this model as following. Current Kepler architecture features up to 15 *streaming multiprocessors* (SM for short) each having 192 *processing units* (PUs for short) or in other notation *cores*, see Figure 2.4. A CUDA thread block virtualizes streaming multiprocessor, while threads virtualize processing units. There can be up to $2^{32} - 1$ thread blocks each having up to 2048 threads. The hardware schedules threads in groups of size 32, called warps, thus physical SIMD width is 32 only. There is also 32 shared memory banks respectively. Thus, as an optimization strategy, one should optimize I/O operations and SIMD instructions for groups of threads of size 32, but not for the whole thread block's width. The number of threads in a thread block should be at least the number of physical cores or more. Hyper-threading helps the hardware to fully utilize available parallelism and hide memory latency.

Unfortunately, one cannot directly translate results from PRAM or external memory model. PRAM does not account for fast shared memory and SIMD fashion processing. Since the ratio between accessing GPU's global memory in random fashion and consecutively is much smaller than accessing the hard drive and CPU's main memory, pure external memory algorithms are also of limited use for GPUs. For instance, sorting the data and further accessing it continuously is still slower than random gather from global memory. Thus, in algorithm design, we can not limit ourselves to a single optimization criteria. If we optimize for I/O operations and ignore SIMD behaviour, our algorithm easily becomes compute bound due to high serialization rate. If we ignore I/Os, threads wait for data, and become memory bandwidth bound. Finding the right balance between compute load and memory bandwidth for a particular problem is probably the most difficult task in this model.

CHAPTER 3

Algorithm Engineering Infrastruture

In this chapter we describe an "infrastructure" we use throughout the thesis for design, implementation and experimental evaluation of our results.

3.1. Algorithm Engineering Methodology

We follow *algorithm engineering* methodology, probably most thoroughly described by Sanders [140]. One of its most important aims is to supply a faster transfer of new algorithmic results into real applications.

Historically, the algorithm design was limited to developing theoretical solutions with worst-case boundaries for some particular machine model. Some of the problems with this approach include:

- machine model might not reflect real hardware;
- theoretical boundaries might include large constant factors;
- worst-cases might be rare in practice;
- the algorithm might be too difficult to implement.

The algorithm engineering cycle (see Figure 3.1) suggests that the design and analysis of algorithms using realistic machine models should get continues feedback from algorithm's implementation using inputs stemming from real-world applications. Moreover, each part of the cycle is equally important for obtaining the final solution. Thus, for instance, we should not limit ourselves to the widely-used RAM Model (see Section 2.1) and take the hard-drive into consideration as soon as our input does not fit into the main memory of the machine (see Section 2.2). We should not hide constant factor under the $\mathcal{O}(\cdot)$ notation, since it may have great impact on the performance. Real-world inputs is the main source

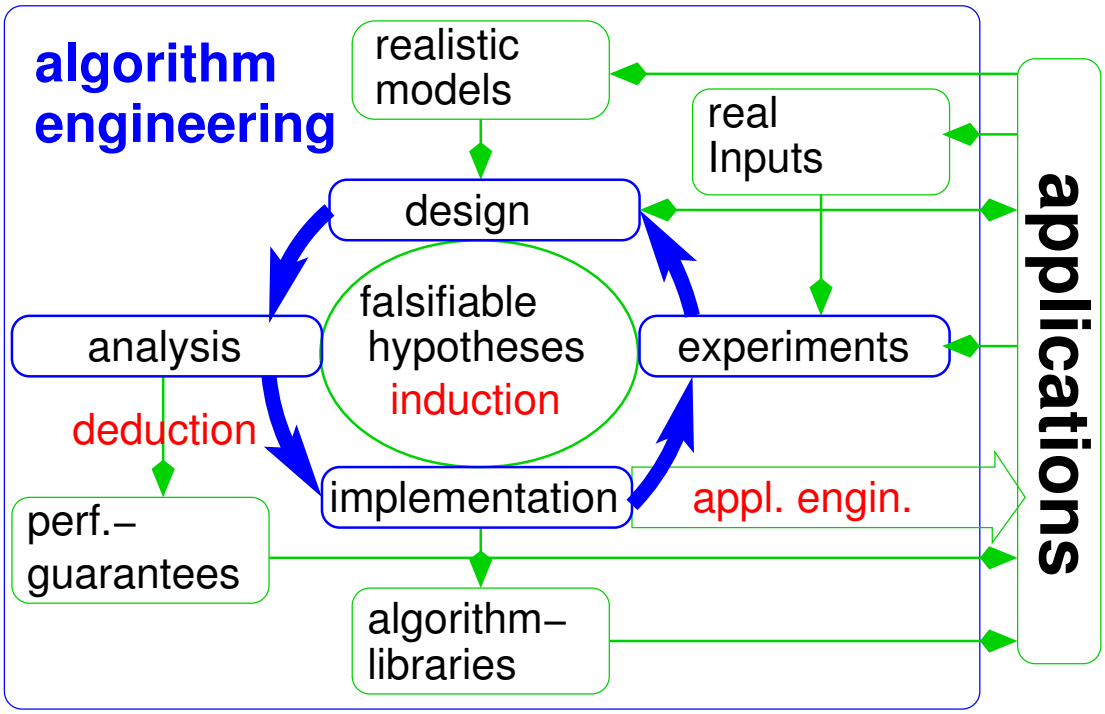


Figure 3.1.: Algorithm Engineering cycle [140]

of the performance evaluation. The synthetical inputs are not enough. While designing the algorithm and especially reusing existing results, we should keep the algorithm simple enough to be implementable.

Throughout the thesis we mostly show the last iteration of this cycle, omitting unsatisfactory or unsuccessful iterations.

3.2. Machine Configurations

Throughout the thesis, we use the following machines to do the performance study of our implementations. We keep the details in Table 3.1 and refer to the corresponding configuration names in the sections, where we use them.

3.3. Data Analysis

There several ways to analyze the performance data we obtain thorough the experimental study. One can aggregate data within the main benchmark application. The problem with this approach is, that if we want to redo experiments for some part of the input data it is often difficult and error-prone to update the aggregated numbers.

Another popular approach is to output per instance performance data into separate text data files or one large file. And than run a script, that aggregates the data that we need. This approach allows to recompute part of the experiments, since it does not hide per instance

Name	CPU/ GPU	Frequency [GHz]	Multi- proccesors	Cores per MP	RAM [GB]	Sections
A	Intel Q6600	2.4	1	4	8	4.3.7
B	NVidia Tesla C1060		30	8	4	4.3.7
C	Intel i7 920	2.67	1	4	6	5.5.2, 9.4
D	NVidia Fermi GTX 480		15	32	1.5	5.5.2, 4.3.8
E	Intel Xeon X5355	2.66	2	4	16	5.6.4
F	Intel Xeon E5-2670	2.6	2	8	32	5.6.4
G	AMD Opteron 270	2.0	1	2	3	6.2.2, 7.2.1
H	AMD Opteron 2350	2.0	2	4	16	8.3.5

Table 3.1.: Machine configurations we used throughout the thesis

data in the aggregated values. The problem with this approach is, that the one need to write our own scripts to aggregate the data. As any other application scripts may contain errors. Another problem is that one usually does not see individual per instance data, since it is scattered in within one large or many different files. This way it is possible to overlook instances that demonstrate erroneous behaviour.

We use, in our opinion, the most flexible the least error-prone database based approach. We output all per-instance results into a database. All aggregation operations we perform in SQL using built-in operations, thus, minimizing the probability of making an error in the script. We are also able to detect erroneous behaviour by sorting results in the database by the value in question to see the outliers. This approach proved to be very flexible and efficient throughout experiments we performed in the thesis.

As for visualisation, we can export results from the database into a comma-separated text file, which is easily read by most of the plotting applications. In our experiments we use Gnuplot - a widely-used open source graphing utility that is available for many operations systems including Linux, Windows, OSX etc.

CHAPTER 4

Sorting

Sorting is one of the most widely researched computational problems in computer science. It is an essential building block for numerous algorithms, whose performance depends on the efficiency of sorting. In particular we use sorting for suffix array construction in Chapter 5, graph algorithms for constructing minimum spanning trees and matchings (Chapters 8 and 9). Practically all nontrivial external memory algorithms (for instance Breadth First Search form Chapter 6 and Single Source Shortest Path from Chapter 7) are based on I/O efficient sorting. It is also an internal primitive utilized by database operations, and therefore, any application that uses a database may benefit from an efficient sorting algorithm. Geographic information systems, computational biology, and search engines are further fields that involve sorting. Hence, it is of utmost importance to provide efficient sort primitives for nowadays architectures, which exploit architectural attributes, such as increased parallelism that were not available before.

References. The contents of this chapter is based on the joint work with Nikolaj Leischner and Peter Sanders [92] and PEPPHER project reports [131]. Most of the wording of the original publication is preserved.

4.1. Divide and Conquer Approaches

Due to vast research in the area, there is too much work done to review it here. Therefore, we mainly focus on sorting algorithms for parallel architectures that are most relevant to our work.

Until recently, refined versions of quicksort were considered among the fastest *comparison-based*, that is requiring a comparison function on keys only, sorting algorithms for single core machines used in practice [117]. However, the emergence of current generation CPUs featuring several cores, large caches and an SIMD instruction set, turned the focus on more cache efficient divide-and-conquer approaches that were able to expose a higher level of par-

allelism. Indeed, to our knowledge there is no efficient quicksort implementation, which exploits SIMD instructions. Moreover, despite having perfect spatial locality, quicksort requires at least $\log(n/M)$ scans until subproblems fit into a cache of size M.

A general divide-and-conquer technique can be described in three steps: the input is recursively split into k tiles while the tile size exceeds a fixed size M, individual tiles are sorted independently and merged into the final sorted sequence. Most divide-and-conquer algorithms are based either on a k-way distribution or a k-way merge procedure. In the former case, the input is split into tiles that are delimited by k ordered splitting elements. The sorted tiles form a sorted sequence, thus making the merge step superfluous. As for a k-way merge procedure, the input is evenly divided into $\log_k n/M$ tiles, that are sorted and k-way merged merged together. In contrast to two-way quicksort or merge sort, multi-way approaches perform $\log_k n/M$ scans through the data (in expectation for k-way distribution).

This general pattern gives rise to several efficient manycore algorithms varying only in the way they implement individual steps. For instance, in a multicore gcc sort routine [159], each core gets an equal-sized part of the input (thus k is equal to the number of cores), sorts it using introsort [117], and finally, cooperatively k-way merges the intermediate results.

Another recently published multicore algorithm following the same pattern additionally uses SIMD instructions [148]. For a CPU cache of size M it divides the input into n/M equal-sized parts, sorts them using bitonic sort and SIMD instructions in cache [19], and finally multi-way merges results. To our knowledge this algorithm is the fastest published multicore sorting approach at least for the key types reported in the paper.

4.2. Sorting on GPU

Though GPUs are generally better suited for computational rather than combinatorial problems, new architectural capabilities of graphics processors brought considerable attention to sorting on GPUs.

For example, Sengupta et al. [154] developed an efficient *scan* (prefix sum) primitive – an essential building block for data parallel computation with numerous applications. By reducing counting sort to a number of scan primitives, Satish et al. [147] were able to design an efficient radix sort algorithm. They also gave the first implementation of quicksort that was based on a segmented scan primitive [154]. However, high overhead induced by this approach led to a sort that was not competitive to an explicit partitioning scheme, that was used in an alternative implementation by Cederman and Tsigas [32].

One of the first GPU-based two-way merge sort algorithms appeared as the second phase of a two step approach by Sintorn and Assarsson [160]. The algorithm divides the input into $n/4$ tiles, sorts all of them and merges the chunks in $\log(n/4)$ iterations by assigning one thread to each pair of sorted sequences. To improve parallelism in the last iterations, it initially partitions the input into sufficiently many tiles assuming that the keys are uniformly distributed. Another recent approach is bbsort [35] based on initial partitioning similar to that of hybrid sort.

As for comparison-based sorting algorithms on GPUs in general, the fastest algorithm currently described in the literature is a two-way merge sort by Satish et al. [147]. It divides

the input into $n/256$ tiles, sorts them using odd-even merge sort [19], and two-way merges the results in $\log_2(n/256)$ iterations. In contrast to hybrid sort, several threads can work cooperatively on merging two sequences, therefore eliminating the need for prior partitioning. Recently, Satish et.al [148] further improved their merge sort performance.

In the same work [147], Satish et al. presented a very efficient variant of radix sort, which was until recently superior to all other GPU and CPU sorting algorithms, at least for 32-bit integer keys and key-value pairs.

Merrill and Grimshaw implemented an efficient scan routine that operated at the GPUs memory bandwidth [108], which they later used for an efficient implementation of radix sort that outperformed all available radix sort implementations by a large margin [107].

In the following subsections, we present the design of a sample sort algorithm for manycore GPUs. Despite being one of the most efficient comparison-based sorting algorithms for distributed memory architectures, its performance on GPUs was previously unknown. We show that it is the best current comparison-based sorting algorithm for GPUs.

In our original publication [92], we described the design and implementation of sample sort for Nvidia GPUs using CUDA. We showed that sample sort was also robust to the commonly accepted set of distributions used for experimental evaluation of sorting algorithms [64], and performed equally well for the whole range of input sizes. Our experimental study demonstrated that our implementation was faster than all previously published comparison-based GPU sorting algorithms, and outperformed the state-of-the-art radix sort from the Thrust library on 64-bit integers and some nonuniform distributions of 32-bit integers. One of the main reasons for a better performance of sample sort over quicksort and two-way merge sort is that it needs less accesses to global memory, since it processes the data in a few multi-way phases rather than in a larger number of two-way phases.

Based on these results, we continued our work on sample sort in the following directions

- a new implementation that is particularly efficient for smaller distribution degrees, exploiting coalesced writing for speeding up the distribution phase
- for the smallest distribution degrees 16 and 32 we adopt a distribution phase of radix sort [107]
- a better implementation of the sorting routine for small inputs

We show that

- our implementation outperforms a new tuned variant of merge sort [148] for large uniformly distributed inputs on Nvidia's Tesla architecture
- sample sort scales well and achieves the best published performance in comparison-based sorting on Nvidia's Fermi architecture

4.3. GPU Sample Sort

Sample sort is considered to be one of the most efficient comparison-based algorithms for distributed memory architectures. Its sequential version is probably best described in pseu-

Algorithm 1: Serial sample sort

```
1  SampleSort(e = ⟨e_1, ..., e_n⟩, k)
2  begin
3      if n < M then return SmallSort(e)
4
5      choose a random sample S = S_1, ..., S_{ak-1} of e
6      Sort(S)
7      ⟨s_0, s_1, ..., s_k⟩ = ⟨−∞, S_a, ..., S_{a(k-1)}, ∞⟩
8      for 1 ≤ i ≤ n do
9          find j ∈ {1, ..., k}, such that s_{j-1} ≤ e_i ≤ s_j
10         place e_i in bucket b_j
11         return Concatenate(SampleSort(b_1, k), ..., SampleSort(b_k, k))
12     end
13 end
```

docode, see Algorithm 1. The oversampling factor a trades off the overhead for sorting the splitters and the accuracy of partitioning, which is crucial for load balance.

The splitters partition the input elements into k buckets delimited by successive splitters. Each bucket is then sorted recursively, and the concatenation forms the sorted output. If M is the size of the input when `SmallSort` is applied, the algorithm requires $\mathcal{O}(\log_k n/M)$ k-way distribution phases, expectedly, until the whole input is split into n/M buckets. Using quicksort as sorter for the small cases leads to an expected execution time of $\mathcal{O}(n \log n)$.

Satish et al. [147] pointed out that their main reason for favoring merge sort over sample sort was its implicit load balancing. They considered it more beneficial than sample sort's avoidance of interprocessor communication. Since the bucket sizes heavily depend on the quality of the splitters, this argument is certainly true. However, sufficiently large random samples yield provably good splitters independent of the input distribution. Therefore, one should not overestimate the impact of load balancing on the performance.

On the other hand, due to the high cost of global memory accesses on GPUs, multi-way approaches are more promising than two-way: Each k-way distribution phase requires only $\mathcal{O}(n)$ memory accesses. Expected $\log_k(n/M)$ passes are needed until buckets fit into fast GPU shared memory M. Thus, we can expect $\mathcal{O}(n \log_k (n/M))$ global memory accesses instead of $\mathcal{O}(n \log_2 (n/M))$ required by two-way merge sort [147]. This asymptotic behavior motivated our work in the direction of k-way sorting algorithms, and sample sort in particular.

Before we describe the design of GPU sample sort, we should mention that hybrid sort [160] and bbsort [35] involve a distribution phase, assuming that the keys are uniformly distributed. The uniformity assumption simplifies partitioning, but makes these approaches not competitive to sample sort on nonuniform distributions, see Subsection 4.3.7. Moreover, although such distribution approaches are suitable for numerical keys, they are not comparison-based.

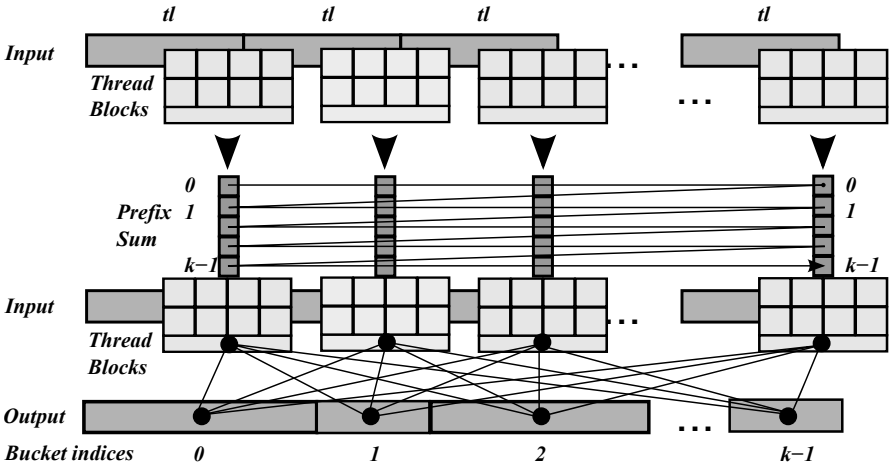


Figure 4.1.: An iteration of k-way distribution

4.3.1. Algorithm Design

A high-level design of a sample-sort's distribution phase, while the bucket size exceeds a fixed size M, can be described in 4 phases corresponding to individual GPU kernel launches, see Figure 4.1.

We divide the input into $p = \lceil n/(t \cdot \ell) \rceil$ tiles of $t \cdot \ell$ elements and assign one block of t threads to each tile, thus each thread processes ℓ elements sequentially.

Phase 1. Choose splitters as in Algorithm 1.

Phase 2. Each thread block computes the bucket indices for all elements in its tile, counts the number of elements in each bucket, and stores this per-block k-entry histogram in global memory.

Phase 3. Perform a prefix sum over the $k \times p$ histogram tables stored in a column-major order to compute global bucket offsets in the output, e. g. use the Thrust implementation [154].

Phase 4. Each thread block finally stores elements at their proper output positions using the global offsets computed in the previous step.

For buckets of size less than M, one can use any GPU sorting algorithm. In our original implementation [92], we chose to use an adaptation of quicksort by Cederman and Tsigas [32]. The improved implementation described in the next Subsection also uses sample sort again on the highest level, see Subsection 4.3.5 for details.

In the following subsections, we give a detailed description of each phase, including design choices we made motivated by architectural attributes, and the performance guidelines reviewed in Section 2.4. The subsections highlight the differences between the original variant presented in [92], and the improved one presented here.

4.3.2. Implementation Details for the Original Algorithm

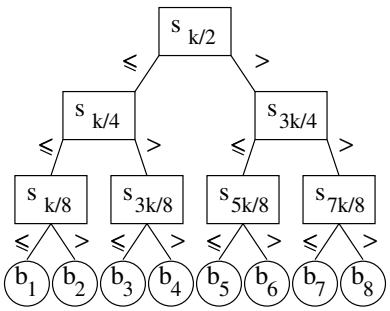
For the original algorithm, we fixed ℓ and t, making p depend on the input size.

Phase 1. We take a random sample S of $a \cdot k$ input elements using a simple linear-congruential random number generator for GPUs, which takes its seed from the CPU-based Mersenne Twister [102]. Then we sort the sample, and place each a-th element of S in the array of splitters b such that they form a complete binary search tree with $b[1] = s_{k/2}$ as the root. The left child of $b[j]$ is placed at position $2j$ and the right child at position $2j + 1$, see Algorithm 2.

Phase 2. To speed up the traversal of the search tree and save accesses to global memory, each block loads b into its shared memory.

To find the target bucket index for an element, we adopt a technique that was originally used to prevent branch mispredictions, impeding instruction-level parallelism on commodity CPUs [146]. In our case, it allows avoiding conditional branching of threads while traversing the search tree. Indeed, a conditional increment in the loop is replaced by a predicated increment. Therefore, threads concurrently traversing the search tree do not diverge, avoiding serialization. Since all possible values for k are known at compile time, the compiler can unroll the loop, which further improves the performance.

Algorithm 2: Serial search tree traversal



$b = \langle s_{k/2}, s_{k/4}, s_{3k/4}, s_{k/8}, s_{3k/8}, s_{5k/8}, s_{7k/8} \ldots \rangle$

```
TraverseTree(e_i)
begin
    j := 1
    repeat log_2 k times
        j := 2j + (e_i > b[j]) // left or right child?
    j := j − k + 1 // bucket index

end
```

Having computed the bucket indices for its elements, each thread block counts the sizes of the resulting buckets by incrementing counters in its shared memory. For correctness,

we use atomic add instructions here. Since many threads may increment the same counter, we lower contention by splitting threads into groups and use individual counter arrays per group. We found $r = 8$ arrays to be a good compromise between overhead for handling several arrays and a lack of parallelism when only one array is used. On hardware that does not support atomic operations, we can explicitly avoid conflicts by using a single thread per group for counting.

Per-block k-entry histograms in global memory resulting from the vector sum computation on the bucket size arrays form the output of this phase. Note that the larger the tiles, i.e., the larger the parameters t and ℓ, the less histogram data is produced by this phase, reducing the number of global memory accesses. On the other hand side, large tiles decrease parallelism. Therefore, the parameter ℓ allows a flexible trade-off between these two effects.

Phase 4. This phase basically repeats the second phase. The difference is that instead of counting the number of elements in the buckets, each block computes local offsets within each bucket for its elements. Hence, by using results of the prefix sum in the previous phase, it computes the final element positions and stores the elements there.

At first glance, it seems to be inefficient to do some work twice in phases 2 and 4, namely computing the bucket indices. However, we found out that storing the bucket indices in global memory in-between (as in [146]) was not faster than just recomputing them, i.e., since Phase 4 is completely memory-bandwidth-bound, the hardware is capable of hiding computation behind memory-intensive operations.

4.3.3. Implementation Details for the Improved Algorithm

This time, we fix (dependent on architecture, see Subsection 4.3.6) the number of thread blocks p and the number of threads per blocks, and vary ℓ accordingly.

Phase 1. As in the original algorithm from Subsection 4.3.2.

Phase 2. Bucket indices of input elements and bucket sizes are computed similarly to the original algorithm. An important difference is that every block of t input elements is ordered by the bucket index, and written out as such to global memory. In addition to the $p \cdot k$-sized histogram table as in the original algorithm, we store bucket sizes per each such block of t elements, which are used again in Phase 4. A benefit of the locally ordered layout, especially for small k, is a significantly more efficient Phase 4, which may coalesce writes of elements belonging to the same bucket.

Phase 4. Loads the bucket size information per block of t elements from global memory, and recomputes the bucket indices of the block elements as in Phase 2. By using results of the prefix sum in the Phase 3, it is able to compute element positions in the output. Since every block of t elements is now presorted by the bucket indices, threads within one warp are likely to write their elements into consecutive positions of the same bucket in global memory. The hardware is thus capable of combining such requests into one memory transaction, which

is significantly more efficient. The efficiency of such an approach heavily depends on k, the warp size, and the maximum number of threads per block. Indeed, the smaller k and the larger t is, the more likely it is that the writes performed by threads within one warp are combined into one memory transaction. The choice of concrete parameter values for different architectures is outlined in Subsection 4.3.6.

4.3.4. Special Case: 16- and 32-way Distribution

A recent work by Merrill and Grimshaw [107] describes a very efficient implementation of the radix sort. It is in turn based on the, to our knowledge, most efficient implementation of a scan (prefix sum) primitive by the same authors [108]. The sorting rate reaches up to one billion 32-bit integers per second on the NVidia Fermi architecture. That is at least 2x faster compared to the implementations available before.

Radix sort iterates over the d-bit digit places of the k-bit integer keys from the least-significant to the most significant. In each pass the algorithm performs a stable distribution of keys into 2^d buckets. Thus, the radix sort requires k/d scans to sort the input.

Our interest in this radix sort implementation is due to the efficiency of its distribution pass. Although it was originally designed for integers only, we generalized the implementation for the usage of arbitrary keys and comparison functions.

Such small distribution degree allows fine-grained optimizations on the thread block level. In Phase 2 it allows contention free updates of bucket counters, therefore we do not need atomic operations anymore. Moreover, we do not order elements in Phase 2 as in improved algorithm in Subsection 4.3.3, but rather do it in shared memory in Phase 4, before distributing elements to the global memory. In contrast to the improved algorithm we do not need to write anything to global memory in Phase 2, and do not need to load bucket sizes from global memory in Phase 4. In the same time due to ordering in shared memory we benefit from coalesced writing in Phase 4 as in improved algorithm. Unfortunately, due to the limited amount of shared memory, such optimizations are possible for very small distribution degrees only.

Thus, our sample sort implementation currently uses two adapted radix distribution passes ($d = 4$ or 5) in order to distribute keys into a sufficient number of buckets and then sorts the buckets using our small case sorter, which we described in detail the next subsection.

4.3.5. Sorting Small Buckets.

We delay the sorting of small buckets until the whole input is partitioned into buckets of size at most M. Since the number of buckets grows with the input size, it is much larger than the number of SMs in the usual cases. Therefore, we can use a single thread block per bucket without sacrificing exploitable parallelism. To improve load-balancing we schedule the buckets for sorting ordered by size. In the original algorithm, we employed GPU quicksort by Cederman and Tsigas [32] for sorting the buckets. GPU quicksort does not cause any serialization of work, except for pivot selection and stack operations. Additionally, its consumption of registers and shared memory is modest.

	hardware parameter	value Tesla/Fermi
o	overload factor recommended for architecture	5/5
T	hardware limit on threads per block	512/1024
S	number of streaming multiprocessors	15/30
E	size of shared memory (in number of elements)	16 KB/48 KB

Table 4.1.: The given hardware parameters and recommendations.

In the improved version, we use an implementation of sample sort that is run by a single thread block. It has a fixed small distribution degree k and falls back to GPU quicksort when bucket sizes drop below $M' \ll M$. This approach allows us to proceed with the parallel sorting of buckets as soon as the distribution phase produces enough of them to saturate the hardware. See Subsection 4.3.6 for the concrete set of parameters.

For sequences that fit into shared memory $M'' \ll M' \ll M$, the quicksort routines switches to an odd-even merge sorting network [19]. In our experiments, we found it to be faster than the bitonic sorting network and other approaches like a parallel merge sort.

In summary, we have four stages of sorting in the improved algorithm. Distribution for large inputs parallelized over all SMs, then local distribution parallelized over one thread block, then two-way partitioning in quicksort, then odd-even merge sort.

4.3.6. Tuning Architecture-Dependent Parameters

By well-chosen tuning parameter values, we try to achieve good performance for a wide range of input and hardware characteristics. In the following, we will derive dependencies and formulas that allow us to calculate good parameter values. Table 4.2 summarizes the affected parameters, while Table 4.1 states the hardware parameters given as input.

Original algorithm on Tesla architecture. Since Phase 4 of the original algorithm is not optimized for memory coalescing, the distribution time is significantly slower than the improved variant and does not scale that well with k, given that the work increases only logarithmically (see Figure 4.2). Therefore, to minimize the overall runtime of the algorithm we minimized the number of rounds performed by the distribution. We chose a maximum k such that $r \cdot k$ (where r is the replication factor, see Subsection 4.3.2) integers fit into shared memory, and we have at least o active blocks per multiprocessor, providing a reasonable parallelism as suggested by [108]. We set M by experimentally trading-off the nonuniformity of bucket sizes produced by the k-way distribution, against the better performance of quicksort on small instances. This way, we achieve an almost uniform sorting rate throughout the whole input size range. The concrete values are $k = 128$ and $M = 2^{17}$.

We choose the oversampling factor as to produce a good quality sample, but still to allow sorting the samples in the second phase to be completely performed in shared memory, thus inducing almost no overhead compared to a smaller oversampling factor. In practice, this amounts to $a = 30$ for 32-bit integers, and $a = 15$ for 64-bit integers.

	parameter / formula	GPU	value for original	value for improved
k	distribution degree	Tesla	128	variable
	$k \leq E/(ro)$	Fermi	128	variable
t	number of threads per block	Tesla	256	512
	$t \leq T$	Fermi	256	1024
ℓ	number of elements per thread	Tesla	8	$n/(t \cdot p)$
	$\ell := n/(t \cdot p)$	Fermi	8	$n/(t \cdot p)$
p	number of thread blocks	Tesla	$n/t \cdot \ell)$	150
	$p := oS$	Fermi	$n/(t \cdot \ell)$	75
M	fallback to small case sorter	Tesla	2^{17}	2^{16}
	determined experimentally	Fermi	2^{17}	2^{16}
M''	fallback to odd-even merge sort	Tesla	2^{10}	2^{10}
	$M'' \leq E/o, M'' \in 2^{\mathbb{N}}$	Fermi	2^{10}	2^{11}
r	histogram replication factor	Tesla	8	8
	determined experimentally	Fermi	8	8

Table 4.2.: The deduced tuning parameters for the two algorithm variants, on Tesla and Fermi.

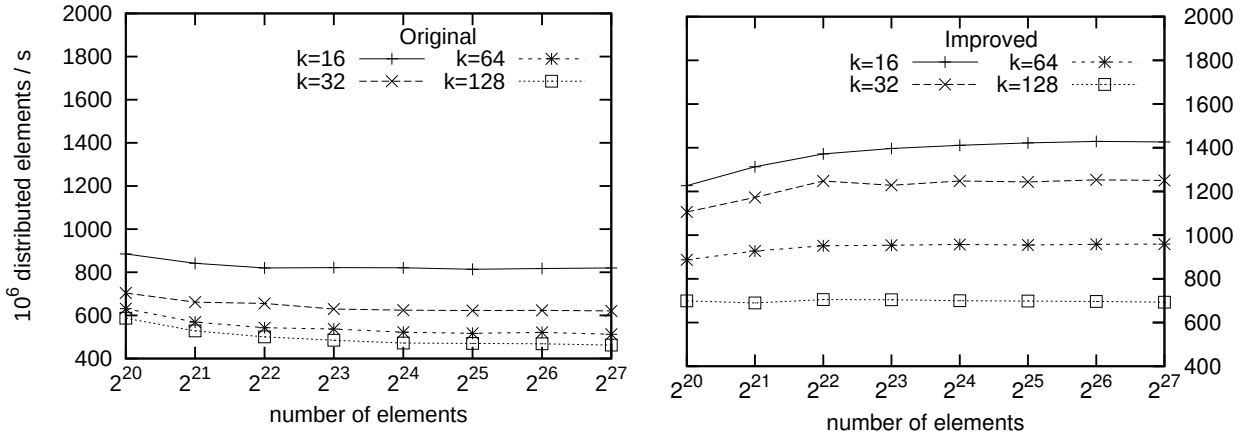


Figure 4.2.: Processing rate of the distribution phase of the original (left) and the improved (right) algorithm on Tesla. The input consists of uniformly distributed 32-bit integers.

When choosing the numbers t (threads per block) and ℓ (elements per thread), we have to achieve a compromise between the parallelism exposed by the algorithm, the amount of data ($(n \cdot k)/(t \cdot \ell)$ indices) written in the second phase, and memory latency in the fourth phase. We choose $t = 256$ and $\ell = 8$.

As stated here for the original algorithm, we made good guesses on the parameter values, based on personal experience. For the improved algorithm, we will describe how to do this in a more systematic way.

Improved algorithm on Tesla and Fermi architecture. Due to a better performance of the improved algorithm's distribution stage, we found it beneficial to vary the distribution degree depending on the input size. We use the following optimization rule: (1) Determine the bucket size M that maximizes the performance of the small case sorting algorithm (by processing rate, i. e. comparisons per time unit); (2) Depending on the input size, choose the distribution degree such that after at most two rounds, the distribution produces enough buckets p of approximate size M to saturate the hardware. We assume that the sizes of the buckets produced in one distribution round are approximately equal. Indeed, we observed [92] that our sample sort performance does not depend significantly on the distribution of input keys, as long as the oversampling factor is large enough.

We determine M experimentally by running the small case sorter on p buckets of size M, and settle it to 2^{16} for both Tesla and Fermi, the two architectures coincide here. As seen in Figure 4.3, the number of comparisons per time unit has a local maximum for this M for Fermi, and going much larger would result in too few buckets, having a negative effect on the load-balancing over the streaming multiprocessors. This value is also good choice for Tesla. Choosing the peak value $M = 2^{14}$ for Tesla threatens performance, since the bucket size will not be the exact value, but spread around the target bucket size. For 2^{13}, though, the performance would drop significantly.

In contrast to the original algorithm, we do not fix the sequential work ℓ per thread, but rather the number of thread blocks p. This shrinks the histogram table produced in Phase 2, and therefore the runtime of scan primitive in Phase 3. As recommended by the vendor, we found p equaling $o = 5$ times the number of multiprocessors to be enough to saturate the hardware. Therefore, for Tesla we choose $p = 150$, while for Fermi $p = 75$, directly deduced from their number of SMs S.

To decrease the amount of data output in Phase 2 and improve coalesced writing in Phase 4, it is profitable to set the number of threads per block to the maximum allowed by the hardware, i. e. $t = 512$ for Tesla $t = 512$ and $t = 1024$ for Fermi.

As for the choice of M'', we settle it to the maximum power of 2 such that at each SM can run o thread blocks. That is, $o \cdot M''$ elements should fit into shared memory of an SM.

Overall, we focus on the distribution phase for selecting the parameters via formulas. For sorting buckets, we fix M'', but then-on rely on experimental results. This retains the possibilities to replace the small case sorter in a black-box fashion.

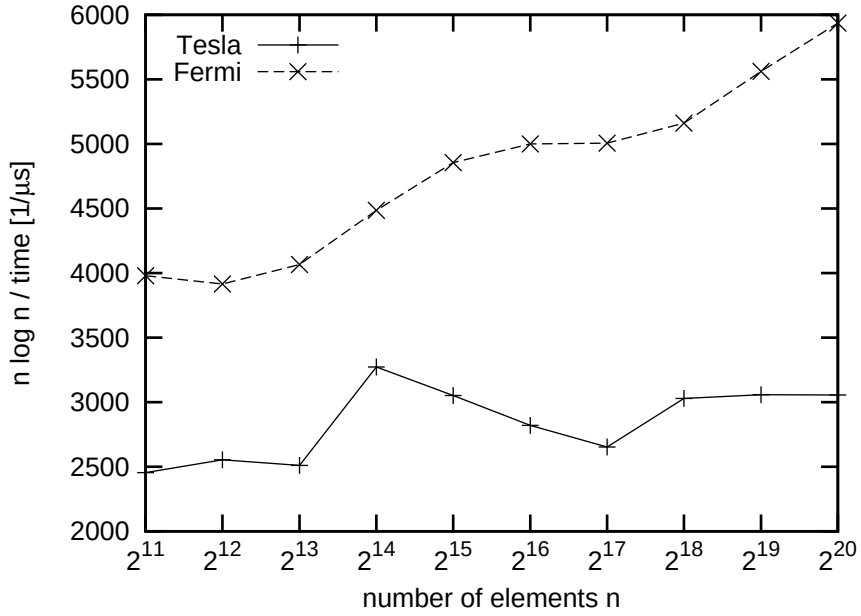


Figure 4.3.: Performance of the small case (SM-local) sorter, depending on the input size, for both platforms.

4.3.7. Experimental Study of Original Algorithm on Tesla

We report on experimental results of our original sample sort implementation on sequences of floats, 32-bit and 64-bit integers, and key-value pairs where both keys and values are 32-bit integers. We compare the performance of our algorithm to a number of existing GPU implementations including: Thrust and CUDPP radix sorts, and Thrust merge sort [147], as well as quicksort [32], hybrid sort [160] and bbsort [35]. Since most of the other algorithms do not accept arbitrary key types, we omit them for the inputs they were not implemented for. We have not included approaches based on graphics APIs in our benchmark, bitonic sort in particular [62], since they are not competitive to the CUDA-based implementations listed above.

Our experimental machine has **configuration A** and the GPU has **configuration B**, see Section 3.2. In comparison to commodity Nvidia cards, the Tesla C1060 has a larger memory of 4GB, that allows a better scalability evaluation. We compiled all implementations using CUDA 2.3 and GCC 4.3.2 on 64-bit Suse Linux 11.1 with optimization level -O3.

We do not include the time for transferring the data from host CPU memory to GPU memory, since sorting is often used as a subroutine for large-scale GPU computations.

For the performance analysis we used a commonly accepted set of distributions motivated and described in [64].

Uniform. A uniformly distributed random input in the range $[0, 2^{32} - 1]$.

Gaussian. A Gaussian distributed random input approximated by setting each value to an average of 4 random values.

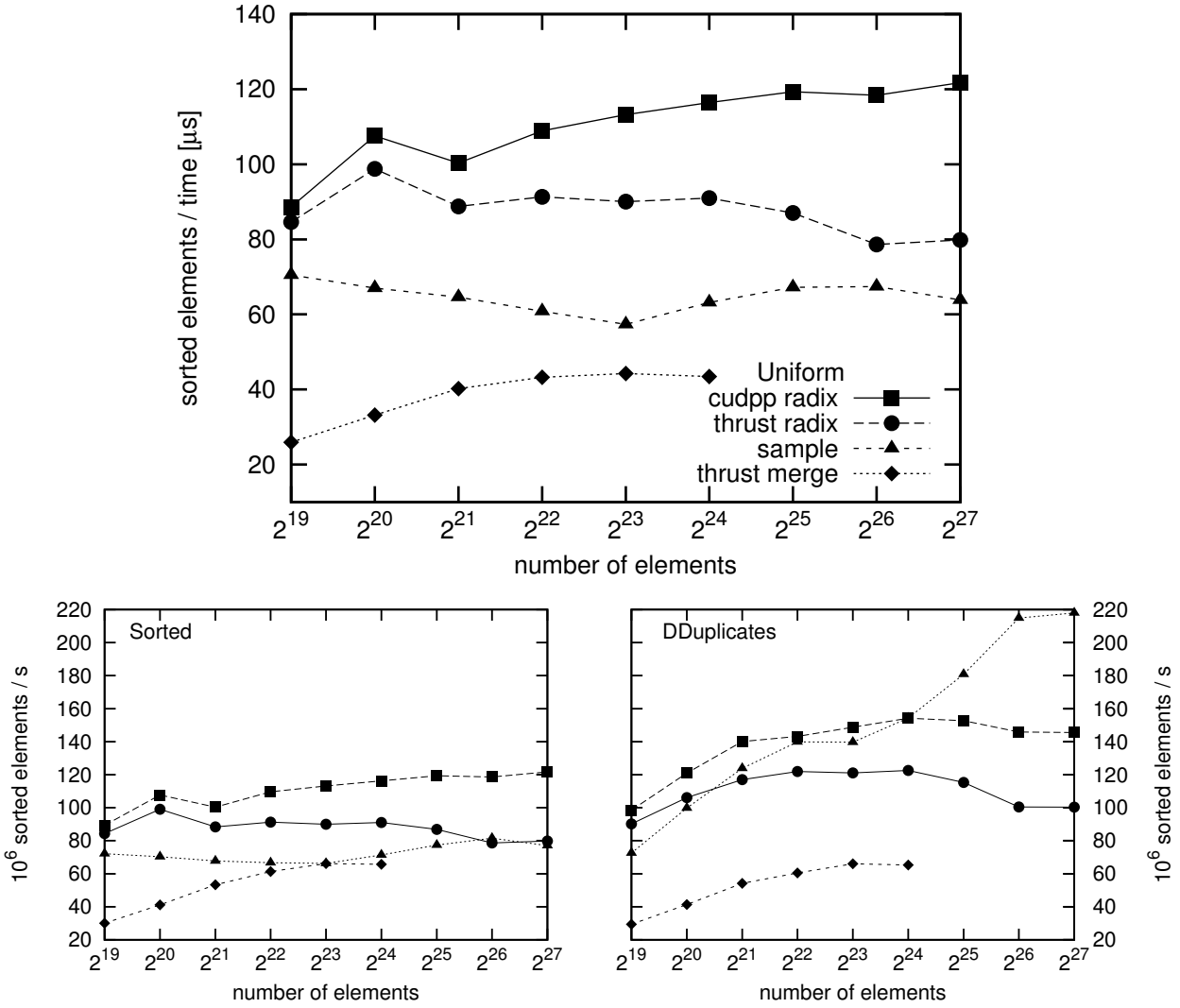


Figure 4.4.: Sorting rates on key-value pairs.

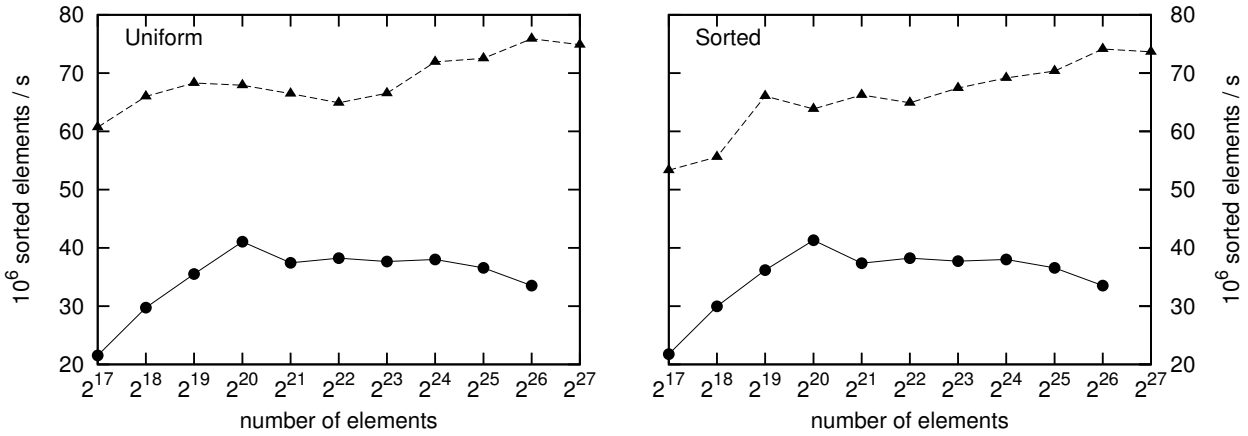


Figure 4.5.: Sorting rates on 64-bit integer keys.

Bucket Sorted. For $p \in \mathbb{N}$, the input of size n is split into p blocks, such that the first n/p^2 elements in each of them are random numbers in $[0, 2^{31}/p - 1]$, the second n/p^2 elements in $[2^{31}/p, 2^{32}/p - 1]$, and so forth.

Staggered. For $p \in \mathbb{N}$, the input of size n is split into p blocks such that if the block index is $i \leq p/2$ all its n/p elements are set to a random number in $[(2i-1)2^{31}/p, (2i)(2^{31}/p-1)]$.

Deterministic Duplicates. For $p \in \mathbb{N}$, the input of size n is split into p blocks, such that the elements of the first $p/2$ blocks are set to $\log_2 n$, the elements of the second $p/4$ processors are set to $\log_2(n/2)$, and so forth.

We use $p = 240$ (the number of scalar processors of a Tesla C1060).

Key-value pairs. Since the best comparison-based sorting algorithm, Thrust merge sort, is designed for key-value pairs only, we can fairly compare it to our sample sort only on this input type. On uniformly distributed keys, our sample sort implementation is at least 25% faster, and on average achieves a 68% higher performance than Thrust merge sort. For key-value pairs, we do not depict all distributions, but rather mention only the behavior of our implementation on uniform inputs, and worst-case sorted sequences. Sample sort is at least as fast as Thrust merge sort, and still is 30% better on average, see Figure 4.4.

Similarly to radix sort on commodity CPUs, CUDPP radix sort is considerably faster than the comparison-based sample and merge sort on 32-bit integer keys. However, on low-entropy inputs, such as Deterministic Duplicates, see Figure 4.4, even for such short-length key types, radix sort is outperformed by sample sort.

64-bit integer keys. With the growth of the key length, radix sort's dependence on the binary key representation renders Thrust radix sort (the only implementation accepting 64-bit keys) uncompetitive to sample sort. On uniformly distributed keys, our sample sort is at least 63%, and on average 2 times faster than Thrust radix. On a sorted sequence, for which

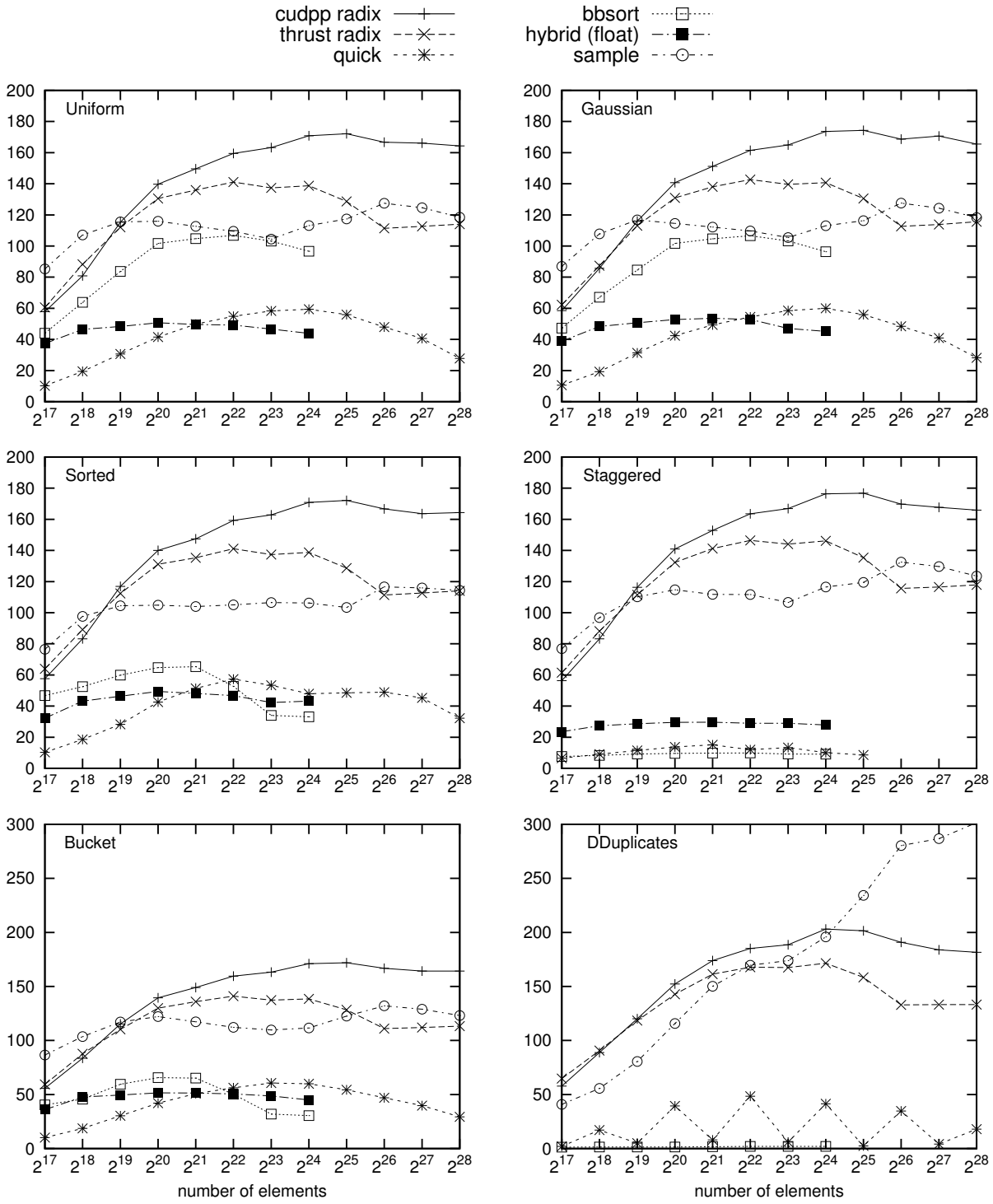


Figure 4.6.: Sorting rates on 32-bit integers.

our implementation performs worst, its sorting rate does not deviate significantly from the uniform case, see Figure 4.5.

32-bit integer keys. Since the majority of GPU sorting implementations are able to sort 32-bit integers, we report sample sort's behavior on all distributions listed above, referring to Figure 4.6. We include hybrid sort results on floats, since it is the only key type accepted by this implementation, and the sorting rates of other algorithms on floats are similar to the ones on integer inputs.

The short length of the key type allows both implementations of radix sort to outperform all algorithms, similar to the 32-bit integer key-value pairs case. Sample sort demonstrates the fastest and still robust performance over all other approaches except for the radix sorts. In particular, it is on average more than 2 times faster than quicksort and hybrid sort, for a uniform distribution. Due to the uniformity assumption, and hence, a reduced computational cost involved as we mentioned in Section 4.3, bbsort is competitive, but still outperformed by our implementation. On the other hand, the performance of bbsort as well as hybrid sort on Bucket and Staggered distributions significantly degrades when compared to the uniform case. Moreover, on the Deterministic Duplicates input, bbsort becomes completely inefficient, while hybrid sort crashes.

Sample sort is robust with respect to all tested distributions and performs almost equally well on all of them. It demonstrates a sorting rate close to constant, i.e., scales almost linearly with the input size. A higher level of parallelism, and hence, a better possibility of hiding memory latency on large inputs, dominate the logarithmic factor in the runtime complexity.

Exploring bottlenecks. Figure 4.7 reports sorting rates of CUDPP and Thrust radix sorts as well as Thrust merge sort and our sample sort on two different GPUs: Nvidia Tesla C1060 and Zotac GTX 285 (both Tesla architecture). These GPUs have the same number of scalar processors, but the GTX 285 is clocked at 1.476 GHz and Tesla at 1.296 GHz, i.e., 13% slower. The measured memory bandwidth of GTX 285 is 124.7 GB/s, while Tesla's is 70% slower, only 73.3GB/s. The average improvements of CUDPP and Thrust radix sorts on the GTX 285 are 30% and 25% respectively, while Thrust merge and sample sorts improve just by 18%. This indicates that none of the algorithms is solely computationally or memory bandwidth bounded. However, the larger improvement for both radix sorts suggests that they are rather memory bandwidth bounded, while merge and sample sort are more computationally bounded.

4.3.8. Experimental Study of Algorithms on Fermi and Performance Portability

Overall Comparison of Algorithms on Both Platforms. Our Fermi's **configuration D** is listed in Section 3.2. In Figure 4.8, we show performance results for the original and the improved algorithm, for both platforms respectively. The improved algorithm is significantly better in most cases, in particular for large inputs, and for Tesla, by up to 40%. The best

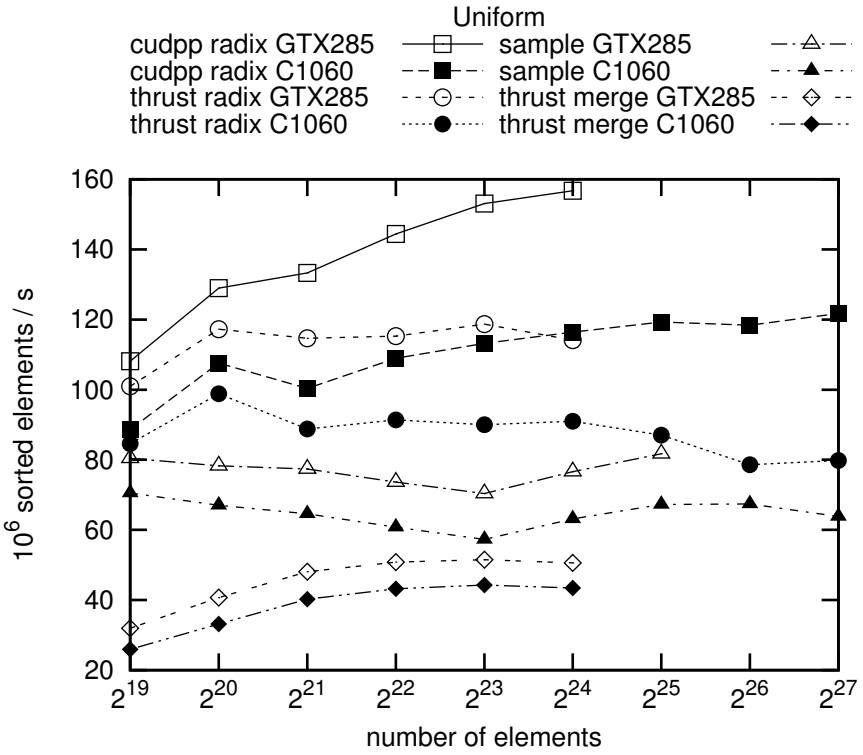


Figure 4.7.: Sorting rates on uniform key-value pairs on Tesla C1060 (filled dots) and Zotac GTX285 (hollow dots).

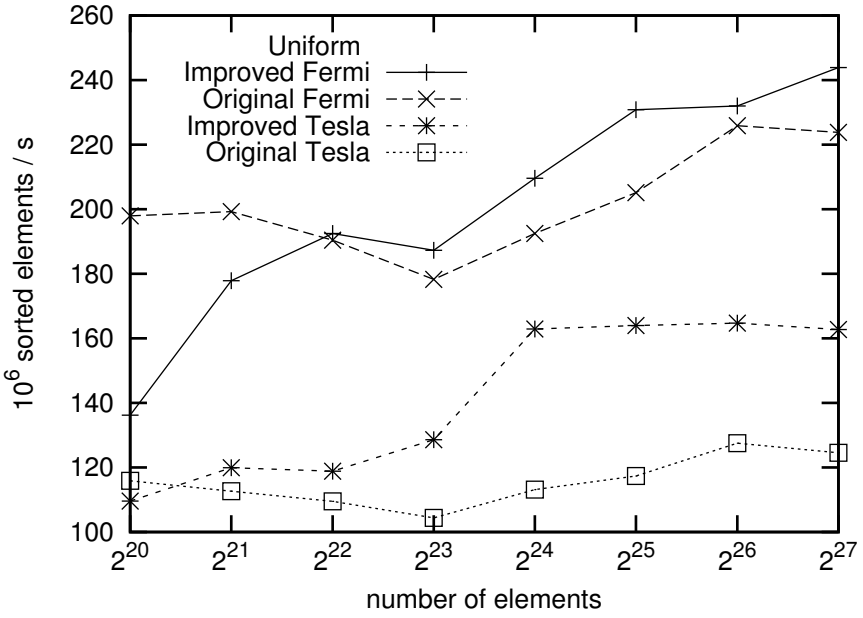


Figure 4.8.: Comparison of the original and the improved algorithm on both platforms.

reported performance for GPU comparison-based sorting is by Satish et. al. [147] achieves on average 176 million elements per second. However, for the largest input, it lies around 150–160 on GTX 280. Our implementation sorting rate is 164 millions elements per second on Tesla C1060, and thus is superior for larger input sizes. As mentioned in Subsection 4.3.7, on a GTX 285, which is comparable to the GTX 280, our algorithm achieves even 18% better performance than on C1060. Thus, we expect an even larger margin over our competitors.

Generally, Fermi is about 50% faster than Tesla. For very small inputs, though, the performance of the improved algorithm on Fermi drops significantly. This is probably due to too little work per thread ℓ. It might be better to have a lower bound for ℓ.

Tuning Parameter Quality. Next, we will analyze in detail the quality of the systematically chosen tuning parameters[1]. The corresponding performance results are shown in Figure 4.9.

For the distribution degree k, we compare against a fixed $k = 128$ (as in the original algorithm), and against doubling or halving the determined k. For Tesla, in fact, the calculated k is usually best, or close to the best. The only input size where it is significantly worse than the doubled k, is for 2^{23} elements. For large inputs, the chosen k clearly wins, and the curve is overall monotonic and quite smooth. Things look even better for Fermi, where the other choices for k lead to highly erratic or embarrassingly bad (fixed k) performance.

The number p of thread blocks actually does not influence the performance that much, at least in the range we tried. For both platforms, the chosed p is among the best values.

The number of threads per block is strictly limited the hardware. A large number of threads, though, increases parallelism and thus the possibility of hiding memory latency. On the other hand, the number of concurrent threads limits the number of registers usable by a single thread. For Tesla in fact, choosing the maximum number gives the best performance. For Fermi, we actually lose performance for $t = 1024$ instead of $t = 512$. However, this loss is not dramatic, and the choice is better than $t = 256$, since it is dominated for large inputs.

M'' determines the maximum size for which the small case sorter finally uses odd-even merge sort. For Tesla, choosing it too large ($M'' = 2048$) gives a serious performance hit, and the chosen M'' actually gives the best results. The results for all tried M'' are very similar, and for the maximum input, we see that the systematically made choice is actually best.

Overall, in most cases, the systematically derived values give the best or nearly best performance. Erratic behavior and bad performance penalties are usually avoided, when compared to simpler or different choices. This shows that the chosen approach carries over to new architectures.

4.3.9. Evaluation of a Special-case Treatment for 16-and 32-way Distribution

In this subsection we study an impact of one further optimization of the improved algorithm, our adaptation of the radix sort[107] distribution pass we described in Subsection 4.3.4.

[1] For many of the tuning parameters, we only consider powers of 2, because we except smaller changes to have only little influence.

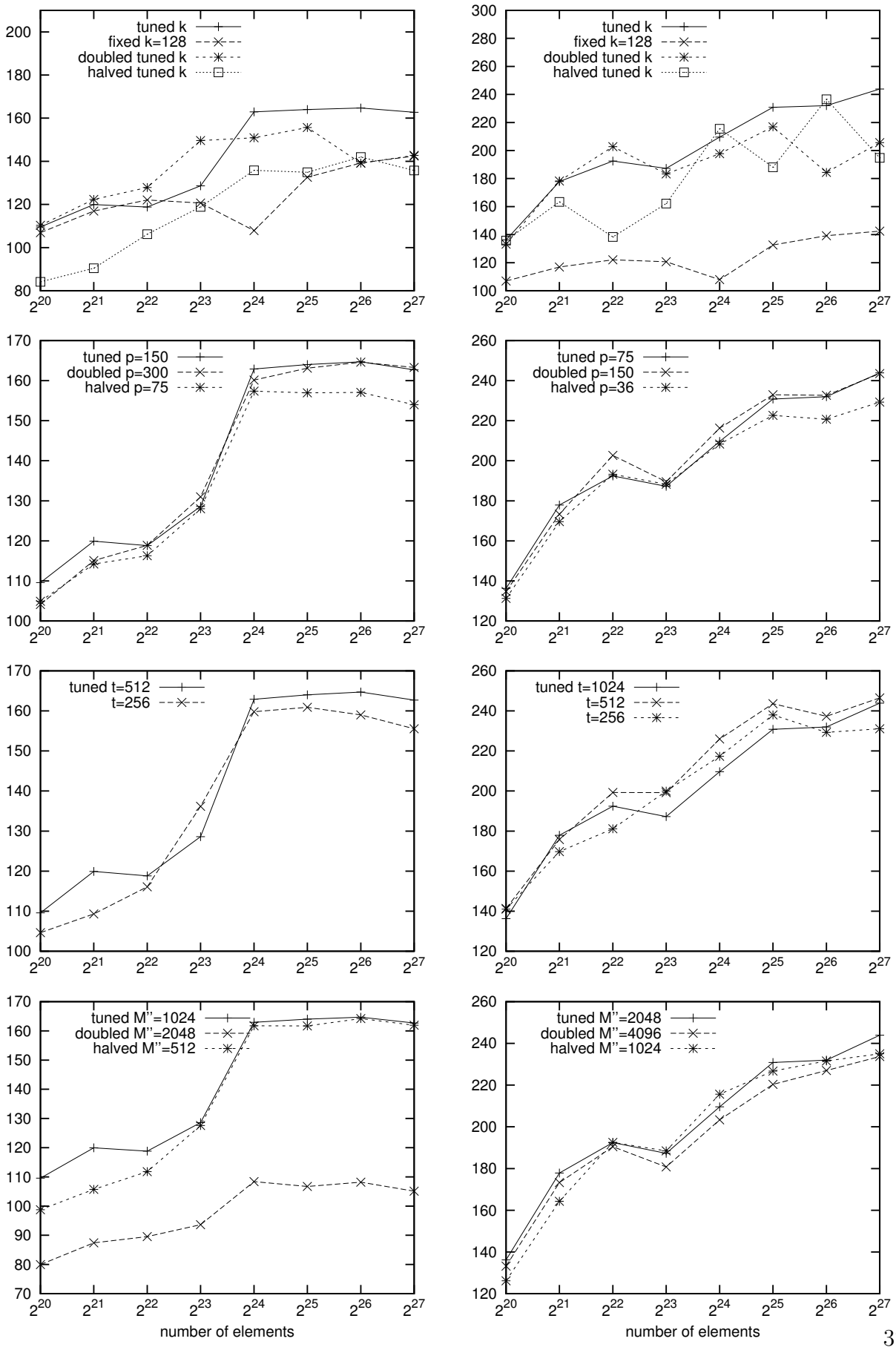


Figure 4.9.: Comparing varied parameter values against the systematically chosen ones, for Tesla (left) and Fermi (right).

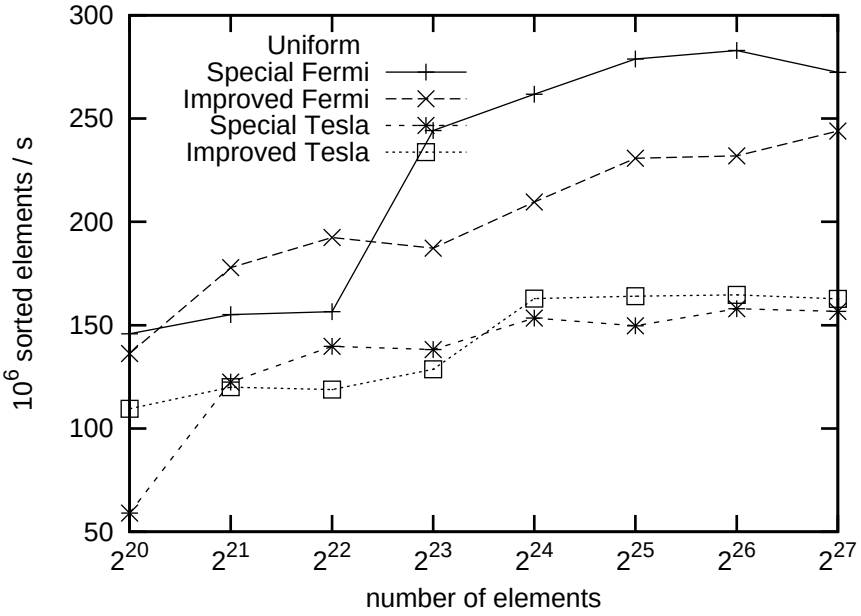


Figure 4.10.: Sorting rate of the improved sample sort ("Improved") and the variant involving adaptation of radix sort distribution ("Special") on 32-bit uniform integer inputs for NVidia Tesla and Fermi architectures

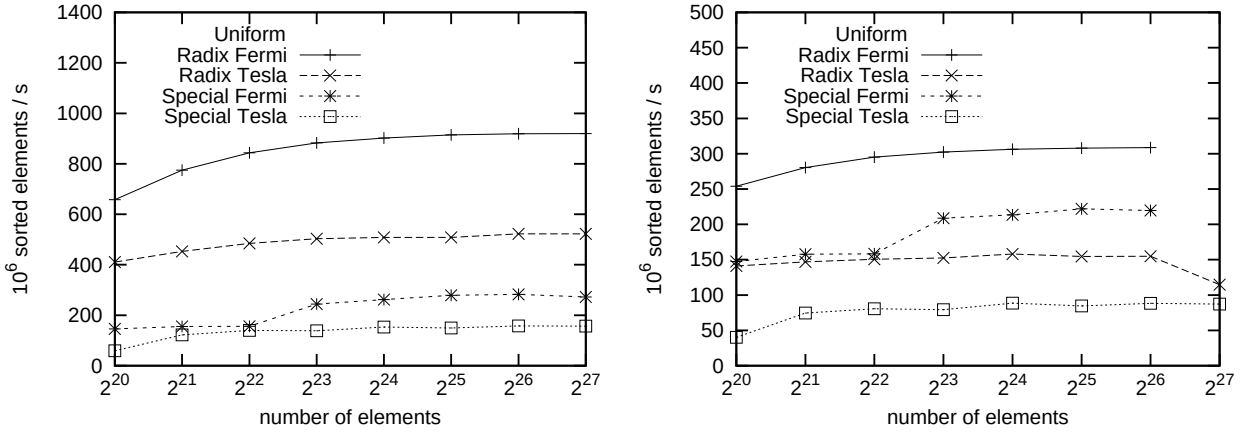


Figure 4.11.: Sorting rate of the variant involving adaptation of radix sort distribution ("Special") and state-of-the-art radix sort on 32-bit (left) and 64-bit (right) integer inputs for NVidia Tesla and Fermi architectures

As seen on the Figure 4.10, the overall performance of the sample sort implementation degraded slightly on the Tesla architecture, but gained a significant performance boost on the Fermi architecture.

It is also of independent interest to compare the general purpose comparison based sample sort implementation with the somewhat limited (due to its binary key representation dependency), but very efficient state-of-the-art radix sort [107] on integer inputs across different key lengths and architectures. Figure 4.11 compares the performance of our sample sort implementation and the radix sort implementation on uniformly distributed 32-bit (left) and 64-bit (right) integers for Fermi and Tesla architectures. As we can see, the performance of the radix sort for 32-bit integers on the Tesla architecture outperforms sample sort by a factor of 2.5 and 3.5 on Fermi. For 64-bit integers the difference is not that significant though. In particular, on the Fermi architecture radix sort outperforms sample sort by 40% only. As the key length increases, radix sort needs to perform more passes through the data, while sample sort's number of passes does not change. Thus, as the key length increases, sample sort also becomes competitive with radix sort for sorting integers.

4.4. Conclusion

In this chapter, we improved the design and implementation of our sample sort algorithm such that it outperforms all previous comparison-based approaches. Our experimental study demonstrates that our implementation is robust to different input types, thus disproving a previous conjecture that the performance of sample sort on GPUs is highly dependent on the distribution of keys [147].

This is an indication that multi-way algorithms in general and the sample sort's multi-way distribution in particular are superior on GPUs to two-way approaches, which require more passes to process the data. In this respect, we believe that the performance of other multi-way approaches should be explored further.

Though we implemented the algorithm using GPUs, our techniques should also suit other manycore architectures well.

By systematically choosing tuning parameters, we have transferred the algorithm's excellent performance to the new Fermi architecture.

CHAPTER 5

Suffix Sorting

Besides general-purpose sorting, many applications require specialized sorting of strings or string's suffixes. If one uses a comparison-based sorting algorithm to sort all suffixes of a string of length n, it would require $\mathcal{O}(n^2 \log n)$ time, since in general case the time to compare two suffixes is $\mathcal{O}(n)$. Suffix sorting is a base of a so called *suffix array construction problem*. Therefore, it appears surprising that there exist suffix array construction algorithms (SACAs) that solve this problem in linear time [76, 87, 121]. Even more surprising is that despite of wide usage in text processing, compression, web search, biology and extensive research over two decades there is a lack of efficient algorithms that are able to exploit shared memory parallelism in practice. We try to close this gap and develop the first approach exposing shared memory parallelism that significantly outperforms the state-of-the-art existing implementations for sufficiently large strings.

For large data sets stemming from such applications as genome processing or web search the bottleneck in most cases is I/O efficiency rather than computation. For such instances, we develop a new SACA in external memory model. Our algorithm outperforms the previous best external memory implementations [42] by a factor of about two in time and I/O-volume.
References. The contents of this chapter is based on the paper [125] and on the joint work with Timo Bingmann and Johannes Fischer [22]. Most of the wording of the original publications is preserved.

5.1. Preliminaries

To simplify the description of the algorithms and make it easier for a direct comparison to the existing listings, we use the notation of the overview paper by Puglisi et al. [137].

Let $T = t_1 t_2 \dots t_n$ be a finite nonempty string of length n, where letters belong to an indexed alphabet Σ. A notion of an *indexed alphabet* can be defined as follows

- λ_j, $j = 1, 2, \dots, \sigma \in \Sigma$ are ordered: $\lambda_1 < \lambda_2 < \dots < \lambda_\sigma$;

- we can define an array $A[\lambda_1 \dots \lambda_\sigma]$, such that accessing $A[\lambda_j]$, $j \in 1 \dots \sigma$ can be done in constant time;
- $\lambda_\sigma - \lambda_1 \in \mathcal{O}(n)$.

This means, that Σ can be mapped to an integer alphabet of a limited range. This restriction though is commonly accepted and non-restrictive for a computational string processing.

A substring $T_i = t_i \dots t_n$ of T is called ith *suffix* of T. Since i ranges from 1 to n there are n suffixes of T.

Thus, by a given string T our goal is to compute a *suffix array* SA_T, or SA for short. $\mathsf{SA}[1 \dots n]$ is an integer array, where $\mathsf{SA}[j] = i \Leftrightarrow T_i$ is the jth suffix in ascending lexicographical order. For convenience, we denote $T_i = t_i \dots t_n$ as a suffix i and append the string with a sentinel \$, which we assume to be less than any letter $\lambda \in \Sigma$.

In the course of the algorithm description, we will need a notion of an *inverse suffix array* denoted as ISA_T, or ISA for short. It is an integer array $\mathsf{ISA}[1 \dots n]$, such that

$$\mathsf{ISA}[i] = j \Leftrightarrow \mathsf{SA}[j] = i$$

For example for $\Sigma = \{a, b, c, d, r, \$\}$ and string

		1	2	3	4	5	6	7	8	9	10	11	12
T	=	a	b	r	a	c	a	d	a	b	r	a	\$
SA	=	12	11	8	1	4	6	9	2	5	7	10	3
ISA	=	4	8	12	5	9	6	10	3	7	11	2	1

Most SACAs proceed by ordering suffixes by their prefixes of increasing length $h \geq 1$, the process that we call h-sorting. The obtained partial order is denoted as *h-ordering* of suffixes into *h-order*. Suffixes that are equal in h-order are called *h-equal*. They have the same *h-rank* and belong to the same *h-group* of h-equal suffixes. If h-sort is stable, then the h-groups for a larger h "refine" the h-groups for a smaller h.

To store a partial h-order, we use an *approximate suffix array* denoted as SA_h or an *approximate inverse suffix array* denoted as ISA_h.

5.2. Related Work

The *suffix tree* of a string is a compact trie of all its suffixes. The *suffix array* and methods for constructing it were proposed by Manber and Myers in 1990 [98] as a simple and space efficient alternative to suffix trees. It is simply the lexicographically sorted array of the suffix indices of a string. Suffix tree and methods for its construction were involved in hundreds of papers over the last two decades. The milestones of this research include the following results:

- there exist space-efficient linear time (in the string length) SACAs [76, 87, 121];
- there exist supralinear time algorithms that are faster in practice than their linear time counterparts [99, 101, 115, 137];

- if a problem is solvable using a suffix tree its asymptotic complexity does not change when the suffix array is used instead [2]

There are three basic techniques for constructing suffix arrays [137] that we informally denote by *prefix-doubling*, *induced copying* and *recursion*. In short, *prefix-doubling* approaches iteratively sort the suffixes by their prefixes that double in length in each iteration, see Section 5.3. *Induced copying* algorithms sort a sample of the suffixes and use them to induce the order of the non-sample suffixes, see Section 5.4 *Recursion* methods recursively reduce the input string length in each iteration. Thus, the existing algorithms can be implicitly divided into three classes according to the technique they exploit. Besides those that can be classified into a single class there exist hybrid approaches that combine at least two of the basic techniques.

On the theoretical side both the induced copying and the recursion class contain linear time algorithms. The Ko and Aluru (KA) [87], Kärkkäinen and Sanders (KS) [76] as well as the recent Nong et al. (SAIS) [121] algorithms can be referred to the hybrid recursive approaches that use the induced copying technique. Though, the underlying ideas behind inducing are different. KA and SAIS use a sample of input-dependent suffixes (*SL-inducing*), while KS's choice of the sample is input-independent and is merely based on the regular suffix positions in the input string (*DC-inducing*).

In practice algorithms based on SL-inducing outperform their DC-inducing counterparts [137]. Moreover, for real-world instances supralinear $\mathcal{O}(n^2 \log n)$ algorithms are often even faster [137]. As long as we are concerned with practical performance and do not insist on linearity, the $\mathcal{O}(n \log n)$ Larsson and Sadakane (LS) [91] algorithm becomes competitive. LS is based on the original prefix-doubling Manber and Myers (MM) [98] algorithm with a powerful filtering criterion on top that makes it significantly (by a factor of 10 or so [137]) faster in practice.

5.3. Prefix-Doubling Algorithms

Algorithms belonging to the prefix-doubling class construct from a given h-order, $h \geq 1$ a $2h$-order of the suffixes of T in $\mathcal{O}(n)$ time. They require at most $\log n$ iterations to build a SA of T, and therefore the overall runtime is $\mathcal{O}(n \log n)$.

Initially prefix doubling algorithms construct SA_1 using linear time bucket sort and proceed further by employing the following idea due to Karp et al. [78]:

Observation 1. *Suppose that* SA_h *and* ISA_h *have been computed for some* $h > 0$*, where* $i = \mathsf{SA}_h[j]$ *is the* j*th suffix in* h*-order and* h*-rank*$[i] = \mathsf{ISA}_h[i]$*. Then, a sort using the integer pairs*

$$(\mathsf{ISA}_h[i], \mathsf{ISA}_h[i+h])$$

as keys, $i + h \leq n$*, computes a* $2h$*-order of the suffixes* i*. (Suffixes* $i > n - h$ *are necessarily fully sorted)*

The MM algorithm applies Observation 1 in the following way. By scanning suffixes $i = \mathsf{SA}_h[j]$ from left to right (that is in h-order) the algorithm necessarily scans the suffixes

$i - h > 0$ within their h-groups in $2h$-order. It also maintains the invariant that the h-rank of the suffix i is the leftmost position (*head*) of its h-group in SA_h. Conceptually, MM is described in Algorithm 3.

Algorithm 3: MM algorithm [98]

```
1  initialize SA_1 by sorting suffixes by their first character
2  initialize ISA_1[i] as the head of i's 1-group in SA_1
3  h = 1
4  while exist non-singleton h-groups do
       /* h-group(i) is the h-group the suffix i belongs to          */
       /* head(i) is the first position in SA_h of the h-group i     */
       /* offset(i) tracks the current position within the h-group i */
5      offset(h-group(·)) ← head(h-group(·))
6      for 1 ≤ j ≤ n do
7          i = SA_h[j] − h
8          if i > 0 then
9              q ← offset(h-group(i))
10             SA_2h[q] ← i
11             if (q = head(h-group(i))) ∨ (h-group(i) ≠ h-group(SA_2h[q − 1]) then
12                 2h-group(i) ← q
13                 head(2h-group(i)) ← q
14             else
15                 dh-group(i) ← dh-group(SA_2h[q − 1])
16             end
17             increase offset(h-group(i))
18         end
19     end
20     compute ISA_2h by ISA[i] ← 2h-group(i)
21     h = h · 2
22 end
```

LS algorithm differs from MM in the following aspects:

- LS explicitly sorts suffixes i ($i \leq n - h$, suffixes $i > n - h$ are necessarily sorted) within each of h-group in SA_h by $\mathsf{ISA}_h[i + h]$ using ternary-split quicksort [21];
- LS avoids rescanning singleton $h - groups$ in the loop 6–19, see Algorithm 3.

5.4. Induced Sorting

Following previous work [121], we classify all suffixes into one of the two *types*: `S` and `L`. For suffix T_i the type(i) is `S` if $T_i < T_{i+1}$, and `L` otherwise. Suffix T_n is fixed as type `S`. Furthermore, we distinguish the "left-most" occurrences of either type as `S`* and `L`*; more

precisely, T_i is $\mathtt{S}^*$ if T_i is $\mathtt{S}$-type and T_{i-1} is $\mathtt{L}$-type. Symmetrically, T_i is $\mathtt{L}^*$-type if T_i is $\mathtt{L}$-type and T_{i-1} is $\mathtt{S}$-type. The last suffix $T_n = [\$]$ is always $\mathtt{S}^*$, while the first suffix is never $\mathtt{S}^*$ or $\mathtt{L}^*$. Sometimes we also say the character t_i is of type(i).

Using these classifications, one can identify subsequences within the suffix array. The range of suffixes starting with the same character c is called the c-bucket, which itself is composed of a sequence of $\mathtt{L}$-suffixes followed by $\mathtt{S}$-suffixes. We also define the *repetition count* for a suffix T_i as $\text{rep}(i) := \max_{k \in \mathbb{N}_0}\{t_i = t_{i+1} = \cdots = t_{i+k}\}$; then the $\mathtt{L}/\mathtt{S}$ subbuckets can further be decomposed into ranges of equal repetition counts, which we call *repetition buckets*.

The principle behind *induced sorting* is to deduce the lexicographic order of unsorted suffixes from a set of already ordered suffixes. Many fast suffix sorting algorithms incorporate this principle in one way or another [137] and we call them SL-inducing based. They are built on the following *inducing lemma* [87]:

Lemma 1. *If the lexicographic order of all* $\mathtt{S}^*$*-suffixes is known, then the lexicographic order of all* $\mathtt{L}$*-suffixes can be induced iteratively smallest to largest.*

Proof. We start with $\mathcal{L} := S^*$ as the lexicographically ordered set of $\mathtt{S}^*$-suffixes. Iteratively, choose the unsorted $\mathtt{L}$-suffix $T_i \notin \mathcal{L}$ that, among all unsorted $\mathtt{L}$-suffixes, has smallest first character t_i and smallest rank of suffix T_{i+1} within $\mathcal{L}$, such that T_{i+1} is already in $\mathcal{L}$. From these properties, $T_i < T_j$ for all $T_j \in \mathcal{L} \setminus \{T_i\}$ follows due to the transitive ordering of $\mathtt{L}$-suffix chains, and T_i can be inserted into $\mathcal{L}$ as the next larger $\mathtt{L}$-suffix. This procedure ultimately sorts all $\mathtt{L}$-suffixes, because each has an $\mathtt{S}^*$-suffix to its right. □

Analogously, the order of all $\mathtt{S}$-suffixes can be induced iteratively largest to smallest, if the relative order of all $\mathtt{L}^*$-suffixes is known. Therefore, it remains to find the relative order of $\mathtt{S}^*$-suffixes.

For each $\mathtt{S}^*$-suffix T_i, we define the k-th $\mathtt{S}^*$*-substring* $[t_i, \ldots, t_j]$, where T_j is the next $\mathtt{S}^*$-suffix in the string. The last $\mathtt{S}^*$-suffix $[\$]$ is fixed to be an $\mathtt{S}^*$-substring by itself. We call the last character t_j of each $\mathtt{S}^*$-substring the *overlapping character*. $\mathtt{S}^*$-substrings are ordered lexicographically, with each component compared first by character and then by type, $\mathtt{L}$-characters being smaller than $\mathtt{S}$-characters in case of ties. This partial order allows one to apply *lexicographic naming* to $\mathtt{S}^*$-substrings [121]. By representing each $\mathtt{S}^*$-substring by its lexicographic name in the super-alphabet Σ^*, one can efficiently reduce the problem of finding the relative order of $\mathtt{S}^*$-suffixes to recursively suffix sorting the string of lexicographic names.

5.5. Parallel Suffix Array Construction for Shared Memory Architectures

Parallel suffix array construction solutions exist in the distributed [53, 89] as well as parallel external memory settings [20, 42]. The most efficient of them are based on KS algorithm (DC-inducing). As for shared memory parallel SACAs, we see almost no progress in the area. The main reasons for that are: (1) all the fastest practical sequential algorithms based on the

SL-inducing technique are difficult to parallelize; (2) the DC-inducing and prefix-doubling techniques involve large overheads making parallelization using a small number of cores of little (if at all) use. Thus, we either need better parallalizable approaches involving smaller overheads, or go beyond commodity multicore machines and design algorithm that would scale well with the number of cores and hence compensate for the increased overhead.

We pursue the second option and reduce the suffix array construction problem to a number of parallel primitives such as prefix sum, radix sorting and random gather/scatter from/to the memory. The performance of our approach depends merely on the scalability and the efficiency of these primitives on a particular architecture. In this section we conduct the performance study of our implementation on manycore NVidia GPUs, though the method itself can be easily transferred to another shared memory (such as multicores, IBM CELL or Intel Xeon Phi) or even distributed architecture, as long as the necessary primitives are available.

Due to a lack of parallel approaches exploiting SL-inducing technique, the choice of the algorithm that would suit a manycore architecture boils down to the prefix-doubling or DC-inducing based methods. The considerations that lead us to the final decision were mainly the asymptotic runtime, the performance on the real-world data and efficiency of the necessary parallel primitives. Analyzing the first two aspects we found out that the better asymptotic behavior of KS algorithm alone does not guarantee the better performance on real world data [137]. Moreover, practical implementation of KS algorithm, requires sorting of large tuples (up to five 32-bit integers) using comparison based sorting and merging afterwards [20, 42]. Though there exist efficient comparison based GPU sorting (see Chapter 4 and merging [147, 148] primitives, their performance is still inferior to that of GPU radix sorting [92, 107, 147, 148]. In contrast to KS, prefix-doubling algorithms (LS and MM) require radix sorting of (32-bit key, 32-bit value) pairs only.

Comparing MM and LS algorithms we found out that each of them has drawbacks with respect to parallelization. LS requires simultaneous sorting of a (possibly large) number of various-size chunks of data and thus needs load balancing. Comparison-based sorting is also considerably slower on GPU than radix sorting, while radix-sorting of small instances of (possibly) large numbers is suboptimal. Therefore, in our approach we perform an implicit $2h$-sort similar to MM. On the other hand, MM induces large overheads by re-sorting suffixes whose final positions in the SA are already defined. Unfortunately, we cannot skip singleton h-groups i while scanning SA_h as LS does. Though being fully sorted themselves, these suffixes still can be important for determining $2h$-rank of suffixes $i-h$. A simple modification of this rule allows formulation of the filtering criterion that we employ in our approach, see Algorithm 3

Observation 2. *If in the k-th iteration of the MM algorithm*

1. *suffix $i = SA_{2^k}[j]$ forms a singleton 2^k-group,*
2. *$i < 2^{k+1}$ or suffix $i - 2^{k+1}$ also forms a singleton 2^k-group*

then for all further iterations $j > k$ either $i < 2^j$ or suffix $i - 2^j$ forms a singleton 2^j-group.

Proof. By induction on j. The base of induction $j = k + 1$ is due to the second condition. Assume that for $j < k + t, t > 1$ the observation is true. If $i - 2^{k+t} > 0$ then the suffix $i - 2^{k+t-1}$ forms a singleton 2^{k+t-1}-group at iteration $k + t - 1$ by the induction hypothesis. Hence, due to the condition in line 11 the suffix $i - 2^{k+t-1} - 2^{k+t-1} = i - 2^{k+t}$ forms a singleton 2^{k+t}-group at the end of this iteration. □

Therefore, we can skip singleton h-groups in the loop 6–19 (see Algorithm 3) in the following iterations as soon as conditions of Observation 2 get fulfilled. We should mention that running time was our primary goal. Therefore, our implementation is not particular lightweight in memory consumption and requires for a string of length n a total of $32n$ byte storage in GPU memory.

For completeness we should mention that Sun and Ma [163] attempted to design a SACA for GPUs. They implemented the original MM algorithm and compared it to its CPU counterpart on random strings. Though their GPU implementation demonstrated a speedup of up to 10 for sufficiently large inputs, the significance of the result is questionable since for real world data MM is proven to be more than an order of magnitude slower than the currently fastest SACAs [137]. Moreover, random strings having an average longest common prefix of length 4 are easy instances for MM.

5.5.1. Algorithm Design

In order to simplify the description of the algorithm, we do not give its full parallel version. We present a "parallel ready" serial description. Having that, it is not particularly difficult to fill in details of the parallel solution. In the following, we discuss parallelization of each code block of Algorithm 4.

Initialization (lines 1 – 3). We initialize SA_h and ISA_h by sorting suffixes by their first h characters. In contrast to the original MM algorithm we initially use $h = 4$ characters. This is exactly the maximum number of one byte characters that can be combined into one 32-bit *key* integer that we use for 4-sorting a corresponding suffix.

The GPU implementation of the algorithm 4-sorts the suffixes using radix sorting of $(key, \mathsf{suffix\_id})$ pairs. We compute a head of each 4-group by determining the starting position of a corresponding sequence of equal keys. $\mathsf{ISA}_4[i]$ is initialized with the 4-rank of the suffix i, which equals the head of its 4-group in SA_4. In parallel it can be done by, firstly, comparing every two consecutive keys and storing either the position of the key if it differs from the previous one or 0 otherwise in an auxiliary array aux. And, secondly, computing inclusive $\mathsf{prefix\_sum}(aux, max)$ over aux using max binary operator. We initialize ISA_4 by scattering aux using $\mathsf{suffix\_id}$ as ISA_4 offsets and making singleton 4-groups negative.

Inducing (lines 9 – 11) and Filtering (lines 13 – 15) . Analogously to MM Algorithm 3 (lines 7–10) suffixes i in h-order induce the $2h$-order of suffixes $i-h$. Unfortunately, we cannot use implicit inducing in the parallel version since insertion of suffixes into newly constructed $2h$-groups would result in a lot of contention. Therefore, we explicitly accompany suffixes $i - h$ with their h-ranks (see lines 10 and 14 of Algorithm 4) in order to track in which

Algorithm 4: "parallel-ready" description of the algorithm

initialize SA_4 by sorting suffixes by their first 4 characters
initialize $\mathsf{ISA}_4[i]$ with the 4-rank of i = head of i's 4-group in SA_4
mark singleton 4-groups by making corresponding entries in ISA_4 negative
Size $\leftarrow n$
while Size > 0 **do**
    /* $\mathsf{SA}_h[i]$ contains ith suffix in h-order */
    /* 2h-rank$[i]$ is the $2h$-rank of the suffix $\mathsf{SA}_h[i]$

$$\text{h-rank}[i] = \begin{cases} -\text{h-rank}(\mathsf{SA}_h[i]) & \text{if } \mathsf{SA}_h[i]+h \text{ is a singleton h-group} \\ \text{h-rank}(\mathsf{SA}_h[i]+h) & \end{cases}$$

$$\mathsf{ISA}_h[i] = \begin{cases} \text{h-rank(i)} & \\ -\text{h-rank(i)} & \text{if } i \text{ is a singleton } h\text{-group} \end{cases}$$

    */
    $s \leftarrow 0$
    for $0 \le j <$ Size **do**
        $i \leftarrow \mathsf{SA}_h[j] - h$
        if $(i > 0) \wedge (\mathsf{ISA}_h[i] > 0)$ **then**
            $(\mathsf{SA}_{2h}, \text{2h-rank}, \text{h-rank})[s{+}{+}] \leftarrow (i, \mathsf{ISA}_h[i], \mathsf{ISA}_h[\mathsf{SA}_h[j]])$
        end
        $i \leftarrow \mathsf{SA}_h[j]$
        if $(\mathsf{ISA}_h[i] < 0) \wedge (i - 2h) > 0 \wedge (\mathsf{ISA}_h[i-2h] > 0)$ **then**
            $(\mathsf{SA}_{2h}, \text{2h-rank}, \text{h-rank})[s{+}{+}] \leftarrow (i, \mathsf{ISA}_h[i], -\mathsf{ISA}_h[i])$
        end
    end
    radix_sort(key(2h-rank), $value$(SA_{2h}, h-rank))
    $head \leftarrow 0$ /* correcting 2h-ranks */
    for $1 \le j < s$ **do**
        if 2h-rank$[j] \neq$ 2h-rank$[head]$ **then**
            $head \leftarrow j$
        else
            if h-rank$[j] \neq$ h-rank$[head]$ **then**
                2h-rank$[j] \leftarrow$ 2h-rank$[j] + j - head$
                $head \leftarrow j$
            else
                2h-rank$[j] \leftarrow$ 2h-rank$[head]$
            end
        end
    end
    $\mathsf{ISA}_{2h}[\mathsf{SA}_{2h}[i]] \leftarrow$ 2h-group$[\mathsf{SA}_{2h}[i]]$ and make singleton $2h$-groups negative
    Size $\leftarrow s$, $h \leftarrow h \cdot 2$
end

h-group they are inserted. By *stably* sorting these suffixes using their h-ranks we get exactly the same $2h$-order as the MM Algorithm.

Moreover, in order to obtain new $2h$-groups we need to determine pairs of consecutive h-equal suffixes whose $2h$-order is induced by the suffixes that are not h-equal (condition in line 11 of Algorithm 3). Therefore, we include the h-rank of inducing suffixes also (lines 10 and 14 of Algorithm 4). Thus, we need to construct triples containing the h-rank of suffix $i-h$, $i-h$ itself and the h-rank of i.

We additionally employ Observation 2 in order to avoid re-sorting of singleton h-groups. As soon as the conditions of Observation 2 are fulfilled the algorithm excludes the corresponding suffixes from further processing.

In order to implement this phase in parallel, the algorithm performs two scans through the data. In the first scan we define which elements can be skipped by storing 0s in the corresponding entries of auxiliary array aux. After running $\mathsf{prefix\_sum}(aux, +)$ over aux each triple gets its position in the compacted array and therefore can be placed in the correct position in the second pass through the data in parallel.

Sorting (line 17) We *stably* radix sort triples by h-ranks of the suffixes to construct SA_{2h}.

Computing $2h$-ranks (lines 18 – 30). The $2h$-rank of a suffix i is equal to its h-rank incremented by the offset of the leftmost h-equal suffix j (possibly i itself), such that $2h$-order of i and j is induced by h-equal suffixes.

The parallel implementation consists of two phases. In the first phase, we compute for each suffix the position of the leftmost h-equal suffix from it. This can be done by (1) defining suffixes that lie in the beginning of the group (each element is compared with its predecessor), (2) storing the positions of such suffixes or 0 otherwise in an auxiliary array aux and (3) computing $\mathsf{prefix\_sum}(aux, \max)$. Thus, aux contains the computed positions for corresponding suffixes. In the second phase for suffixes that lie in the beginning of new $2h$-groups (conditions in lines 20 and 23 are verified for each two consecutive elements) we store in an auxiliary array $\widetilde{aux}$ their old h-rank incremented by the difference of their position and corresponding value in aux. For the rest of suffixes we store 0. By computing $\mathsf{prefix\_sum}(\widetilde{aux}, \max)$ we obtain new $2h$-ranks in $\widetilde{aux}$.

We update ISA_{2h} by scattering values of $\widetilde{aux}$ and making ISA_{2h} values corresponding to singleton $2h$-groups negative (line 31).

Implementation Details. Thus, we reduced our "parallel-ready" approach in Algorithm 4 to a number of parallel primitives. In particular prefix sum, radix sort, scatter/gather to/from the memory. In our implementation, we use memory-bandwidth optimal GPU prefix sum and the most efficient GPU radix sort by Merrill and Grimshaw [107, 108].

Memory Consumption. We mainly targeted performance, therefore our implementation is not particularly lightweight in memory consumption. We need to sort integer triples requiring $6n$ integers and maintain ISA and SA itself. Thus, the overall memory consumption is $8n$ integers.

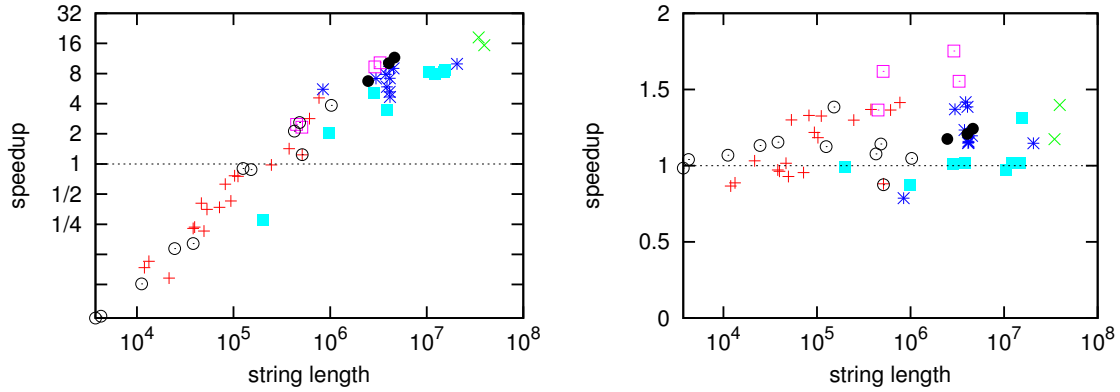


Figure 5.1.: The relative speedup of GPU SACA compared to serial LS algorithm (left) and 4-core divsufsort compared to its sequential version (right)

5.5.2. Experimental Study

Our experimental machine has **configuration C** and the GPU **configuration D**, see Section 3.2. We compiled all implementations using CUDA 4.1 RC 2 and Microsoft Visual Studio 2010 on 64-bit Windows 7 Enterprise with maximum optimization level.

We do not include the time for transferring the data from host CPU memory to GPU memory as suffix array construction is often a subroutine in a more complex algorithm. Therefore, we expect applications to reuse the constructed data structure for the further processing on GPU.

We performed the performance analysis on widely used benchmark sets of files including Calgary Corpus, Canterbury Corpus, Large Canterbury Corpus, Manzini's Large Corpus, Maximum Compression Test Files, Protein Corpus and The Gauntlet [115]. Due to the GPU memory capacity and the memory requirements of our implementation we include into the benchmark strings of size at most 45 MB.

In Figure 5.1 (left) we show the relative speedup of our implementation over the original LS Algorithm [91]. For instances under 10^5 characters the usage of a GPU is inferior to simple serial implementation. Such short instances are not capable to saturate the hardware and efficiently exploit available parallelism. On the other hand, the CPU is able to realize the full potential of its cache that fits the whole input.

Though for larger instances our implementation achieves a considerable speedup of up to 18 over its sequential counterpart. Sufficiently small fluctuations in speedup for approximately equally sized instances suggests that the behavior of our MM variant is similar to LS. Hence, our filtering criterion, see Observation 2, also effectively prevents resorting of fully sorted suffixes. This way we avoid an order of magnitude performance drop inherent to the original MM algorithm when compared to LS in practice [137].

We also compare the performance of our implementation to Yuta Mori's highly tuned CPU implementation divsufsort 2.01 [115], which also makes use of available parallelism by exploiting OpenMP constructs, see Figure 5.2. We should mention that divsufsort scales pretty bad with the number of processors, since algorithmically it is not a parallel solution,

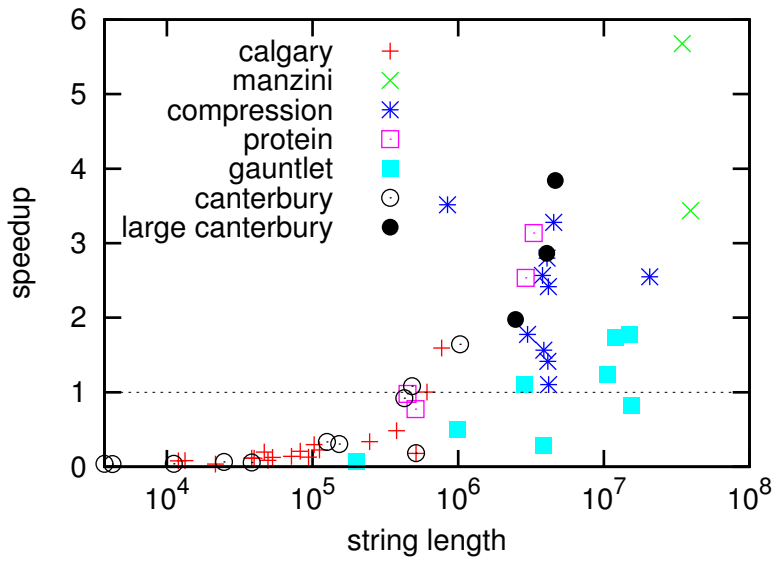


Figure 5.2.: The relative speedup of GPU SACA compared to 4-core divsufsort

see Figure 5.1 (right). Nevertheless, we enabled the OpenMP option in order to show the performance one would be able to obtain using available shared memory CPU solutions and compare it with the performance of our GPU implementation.

The relative speedup fluctuates significantly depending on the instance. This is due to different techniques that are used in the algorithms. For example, three instances that are simply multiple concatenation of some string from the Gauntlet set are still faster on a CPU. The reason is, that these are the most difficult inputs for prefix doubling algorithms. The filtering criterion is also of little help here, since most of the suffixes get fully sorted only on the last few iterations of the algorithm. While for the class of induced copying algorithms, which includes divsufsort, this kind of instances are not particularly hard.

Nevertheless, our implementation achieves a speedup of up to 6 for the majority of significantly large instances.

5.5.3. Discussion

In this section we demonstrated the design of a suffix array construction algorithm for parallel architectures. We reduced a well known MM algorithm to a number of parallel primitives such as prefix sum, radix sorting, scatter/gather to/from the memory. We proposed a new simple filtering criterion, that allows MM to avoid extensive resorting of fully sorted suffixes similar to the LS Algorithm. We implemented the proposed approach for manycore NVidia Fermi architecture, though the same method can be applied for any shared or even distributed memory architecture as long as the necessary parallel primitives are available. We showed a speedup of up to 18 over sequential LS algorithm, and demonstrated the efficiency of our filtering criterion. We compared the performance of our parallel implementation on a set of widely used instances and showed a significant speedup of up to 6 for majority of sufficiently

large instances over the state-of-the-art OpenMP assisted divsufsort algorithm.

In this work we used massive parallelism to compensate for inherent overheads induced by parallelizable approaches, though it would be of immediate interest to find asymptotically optimal algorithms that are, firstly, parallelizable and, secondly, have better constant factors in runtime and thus induce less overheads in practice.

5.6. Suffix Array Construction in External Memory

While being very fast, our GPU implementation is limited by the available GPU memory, which is currently rather limited. Thus, applications involving massive data processing like genome sequencing or search engines are out of scope of our GPU solution. To tackle this problem we turned to the External Memory (EM) model, see Section 2.2

In 2009 Nong et al. [121] presented an extremely elegant linear time algorithm called SAIS (based on the SL-inducing principle [70, 87], see Section 5.4) that was on par with the best superlinear algorithms on all *practical* inputs. Its worst-case guarantees also imply that it has a similar behavior on *all* inputs, while for all engineered superlinear algorithms [99, 101, 151, etc.] there exist worst-case inputs where their running time shoots up by several orders of magnitude.

Unfortunately, being inherently sequential, this algorithm is not suitable for our shared memory parallel solution. On the other hand, motivated by the superior performance of the SAIS algorithm over other suffix array construction algorithms in internal memory, we investigate if the SL-inducing principle can be exploited also in the EM model.

We make the first comparative study of suffix arrays in EM model that includes the induced sorting principle, since all previous studies [17, 42] were conducted before the advent of SAIS. We show that SAIS is suitable for the EM model by reformulating the original algorithm such that it uses only scanning, sorting, merging and a priority queue - three primitives and a data structure that are efficient in terms of EM model as in theory [9] as in practice [41, 141]. We make careful implementation decisions in order to keep the I/O-volume low. As a result, our new algorithm, called eSAIS, is about two times faster than the EM-implementation of DC3 [42]. The I/O volume is reduced by a similar factor. Since we do not use parallel components in our implementation we do not make direct comparison to a more recent DC3 implementation involving shared memory parallel components and parallel asynchronous I/Os [20]. The speedup reported in the paper [20] of up to 1.63 over sequential version [42] suggests that even the parallel version would be outperformed by our implementation of SAIS.

5.6.1. Algorithm Design

Our first goal is to design an EM algorithm based on the induced sorting principle that runs in sorting complexity and has a lower constant factor than DC3 [42]. The basis for this algorithm is an efficient EM priority-queue (PQ) [41], as suggested by the proof of Lemma 1. Since it is derived from RAM-based SAIS, we call our new algorithm eSAIS (*External Suffix Array construction by Induced Sorting*). We first comment on details of the

Algorithm 5: eSAIS description in tuple pseudo-code

```
1  eSAIS(T = [t_0 ... t_{n-1}]) begin
2      Scan T back-to-front, create [ (s*_k) | k ∈ [0,K) ] for K S*-suffixes, and sort
       S*-substrings:
           P := Sort_{S*}[ ([t_i ... t_j], i, type(j)) | (i,j) = (s*_k, s*_{k+1}), k ∈ [0,K) ]   // with
       s*_K := n − 1
3      N = [ (n_k, i) ] := Lexname_{S*}(P)   // choose lexnames n_k ∈ [0,K) for S*-substrings
4      R := [ n_k | (n_k, i) ∈ Sort(N by 2nd component) ]   // sort lexnames to string order
5      if the lexnames in N are not unique then
6          SA_R := eSAIS(R)                                   // recursion with |R| ≤ |T|/2
7          ISA_R := [ r_k | (k, r_k) ∈ Sort[ (SA_R[k], k) | k ∈ [0,K) ] ]   // invert permutation
8      else                          // (Sort sorts lexicographically unless stated otherwise.)
9          ISA_R := R                                         // ISA_R has been generated directly
10     end
11     S* := [ (t_j, S, ISA_R[k], [t_{j-1} ... t_i], j) | (i,j) = (s*_{k-1}, s*_k), k ∈ [0,K) ]   // s*_{-1} := 0
12     ρ_L := 0,   Q_L := CreatePQ(S* by (t_i, y, r, [t_{i-1} ... t_{i-ℓ}], i))
                                                  /* induce from next S*- or L-suffix */
13     while (t_i, y, r, [t_{i-1} ... t_{i-ℓ}], i) = Q_L.extractMin() do
14         if y = L then A_L.append((t_i, i))                 // save i as next L-type in SA
15         if t_{i-1} ≥ t_i then Q_L.insert(t_{i-1}, L, ρ_L++, [t_{i-2} ... t_{i-ℓ}], i − 1) // T_{i-1} is L-type?
16         else L*.append( (t_i, L, ρ_L++, [t_{i-1} ... t_{i-ℓ}], i) )   // T_{i-1} is S-type
17     end
18     Repeat lines 11–15 and construct A_S from L* array with inverted PQ order and
       ρ_S--.
19     return
       [ i | (t,i) ∈ Merge((t_i, i) ∈ A_L and (t_j, j) ∈ A_S.reverse() by first component) ]
20 end
```

pseudocode shown as Algorithm 5, which is a simplified variant of eSAIS. Subsection 5.6.2 is then devoted to complications that arise due to large S*-substrings.

Let R denote the reduced string consisting of lexicographic names of S*-suffixes. The objective of lines 2–9 is to create the inverse suffix array ISA_R, containing the ranks of all S*-suffixes in T. In line 2, the input is scanned back-to-front, and the type of each suffix i is determined from t_i, t_{i+1}, and type$(i+1)$. Thereby, S*-suffixes are identified, and we assume there are K S*-suffixes with $K-1$ S*-substrings between them, plus the sentinel S*-substring. For each S*-substring, the scan creates one tuple. These tuples are then sorted as described at the end of Section 5.4 (note that the type of each character inside the tuple can be deduced from the characters and the type of the overlapping character). After sorting, in line 3 the S*-substring tuples are lexicographically named with respect to the S*-substring ordering, and the output tuple array N is naturally ordered by names $n_k \in [0, K)$. The names must be sorted back to string order in line 4. This yields the reduced string R, wherein each character represents one S*-substring. If the lexicographic names are unique, the lexicographic ranks

of S*-substrings are simply the names in R (lines 8–9). Otherwise the ranks are calculated recursively by calling eSAIS and inverting SA_R (lines 5–7).

With ISA_R containing the ranks of S*-suffixes, we apply Lemma 1 in lines 10–15. The PQ contains quintuples $(t_i, y, r, [t_{i-1}, \ldots, t_{i-\ell}], i)$ with (t_i, y, r) being the sort key, which is composed of character t_i, indicator $y = \text{type}(i)$ with L < S and relative rank r of suffix T_{i+1}. To efficiently implement Lemma 1, instead of checking *all* unsorted L-suffixes, we design the PQ to create the relative order of S*- and L-suffixes as described in the proof. Extraction from the PQ always yields the smallest unsorted L-suffix, or, if all L-suffixes within a c-bucket are sorted, the smallest S*-suffix i with unsorted preceding L-suffix at position $i-1$ (hence $t_{i-1} > c$). Thus diverging slightly from the proof, the PQ only contains L-suffixes T_i where T_{i+1} is already ordered, plus all S*-suffixes where T_{i-1} has not been ordered; so at any time the PQ contains at most K items. In line 11, the PQ is initialized with the array S^*, which is built in line 10 by reading the input back-to-front again, re-identifying S*-suffixes and merging with ISA_R to get the rank for each tuple. Notice that the characters of S*-substrings are saved in *reverse* order. The while loop in lines 12–15 then repeatedly removes the minimum item and assigns it the next relative rank as enumerated by ρ_L; this is the *inducing* process. If the extracted tuple represents an L-suffix, the suffix position i is saved in A_L as the next L-suffix in the t_i-bucket (line 13). Extracted S*-suffixes do not have an output. If the preceding suffix T_{i-1} is L-type, then we shorten the tuple by one character to represent this suffix, and reinsert the tuple with its relative rank (line 14). However, if the preceding suffix T_{i-1} is S-type, then the suffix T_i is L*-type, and it must be saved for the inducing of S-suffixes (line 15). When the PQ is empty, all L-suffixes are sorted in A_L, and L^* contains all L*-suffixes ranked by their lexicographic order.

With the array L^* the while loop is repeated to sort all S-suffixes (line 16). This process is symmetric with the PQ order being reversed and using ρ_S-- instead of incrementing. If $t_{i-1} > t_i$ occurs, the tuple can be dropped, because there is no need to recreate the array S^* (as all L-suffixes are already sorted). When both A_L and A_S are computed, the suffix array can be constructed by merging together the L- and S-subsequences bucket-wise (line 17). A_S has to be reversed first, because the S-suffix order is generated largest to smallest. Note that in this formulation the alphabet Σ is only used for comparison.

5.6.2. Splitting Large Tuples.

After the detailed description of Algorithm 5, we must point out two issues that occur in the EM setting. While S*-substrings are usually very short, at least three characters long and on average four, in pathological cases they can encompass nearly the whole string. Thus in line 2–3 of Algorithm 5, the tuples would grow larger than an I/O block B, and one would have to resort to long string sorting [12]. More seriously, in the special case of [$] being the only S*-suffix, the while-loop in lines 12–15 inserts $\frac{n(n+1)}{2}$ characters, which leads to quadratic I/O volume. Both issues are due to long S*-substrings, but we will deal with them differently.

Long string sorting in EM can be dealt with using lexicographic naming and doubling [12, Sect. 4]. However, instead of explicitly sorting long strings, we integrate the doubling procedure into the suffix sorting recursion and ultimately only need to sort short strings in line 2 of Algorithm 5. This is done by dividing the S*-substrings into *split substrings* of

Algorithm 6: Inducing step with $\mathtt{S}^*$-substrings split by D_0 and D, replacing lines 10–15 of Algorithm 5

/* *split positions, with* $s^*_{-1} = 0$ */

$\mathcal{D} := \{\, s^*_k - D_0 - \nu{\cdot}D \mid \nu \in \mathbb{N},\, s^*_k - D_0 - \nu{\cdot}D > s^*_{k-1},\, k \in [0, K) \,\}$

$S^* := \mathtt{Sort}[\,(t_j, \mathsf{ISA}_R[k], [t_{j-1} \dots t_i], j, \mathbb{1}_{i\in\mathcal{D}}) \mid j = s^*_k,\, i = \max(s^*_{k-1}, j - D_0),\, k \in [0, K)\,]$

$L := \mathtt{Sort}[\,(t_j, \mathrm{rep}(j), j, [t_{j-1} \dots t_i], \mathbb{1}_{i\in\mathcal{D}}) \mid j \in \mathcal{D},\, i = \max(s^*_{k-1}, j - D),\, t_j \text{ is } \mathtt{L}\text{-type}\,]$

$S := \mathtt{Sort}[\,(t_j, \mathrm{rep}(j), j, [t_{j-1} \dots t_i], \mathbb{1}_{i\in\mathcal{D}}) \mid j \in \mathcal{D},\, i = \max(s^*_{k-1}, j - D),\, t_j \text{ is } \mathtt{S}\text{-type}\,]$

$\rho_L := 0, \quad a := \bot, \quad r_a = 0, \quad S^* := \mathtt{Stack}(S^*),$
$Q_L := \mathtt{CreatePQ}(\emptyset \text{ by } (t_i, r, [t_{i-1} \dots t_{i-\ell}], i, c))$

while $Q_L.\mathtt{NotEmpty}()$ or $S^*.\mathtt{NotEmpty}()$ **do**
  while $Q_L.\mathtt{Empty}()$ or $t < Q_L.\mathtt{TopChar}()$ with $(t, r, [t_{i-1} \dots t_{i-\ell}], i, c) = S^*.\mathtt{Top}()$ **do**
    $Q_L.\mathtt{insert}(t_{i-1}, \rho_L\mathtt{++}, [t_{i-2} \dots t_{i-\ell}], i - 1, c), \quad S^*.\mathtt{Pop}()$ // *induce from* $\mathtt{S}^*$
  end
  /* *next a-repetition bucket* */
  $a' := a, \quad a := Q_L.\mathtt{TopChar}(), \quad r_a := (r_a + 1)\mathbb{1}_{a'=a}, \quad m := \rho_L, \quad M := \emptyset$ **while** $Q_L.\mathtt{TopChar}() = a$ and $Q_L.\mathtt{TopRank}() < m$ **do** // *induce from* L*-suffixes*
    $(t_i, r, [t_{i-1} \dots t_{i-\ell}], i, c) = Q_L.\mathtt{extractMin}(), \quad A_L.\mathtt{append}((t_i, i))$ // *save* i *as next* L*-type*
    if $\ell > 0$ **then**
      if $t_{i-1} \geq t_i$ **then** $Q_L.\mathtt{insert}(t_{i-1}, \rho_L\mathtt{++}, [t_{i-2} \dots t_{i-\ell}], i - 1, c)$ // T_{i-1} *is* L
      else $L^*.\mathtt{append}(\,(t_i, \rho_L\mathtt{++}, [t_{i-1} \dots t_{i-\ell}], i, c)\,)$ // T_{i-1} *is* S
    else if $\ell = 0$ and $c = 1$ **then** $M.\mathtt{append}(i, \rho_L\mathtt{++},)$ // *need continuation?*

  end
  foreach
  $\mathtt{Merge}([\,(a, r_a, i, r) \mid (i, r) \in \mathtt{Sort}(M)\,] \text{ with } (a, r_a, i, [t_{i-1}, \dots, t_{i-\ell}], c) \in L)$ **do**
    if $t_{i-1} \geq t_i$ **then** $Q_L.\mathtt{insert}(t_{i-1}, r, [t_{i-2} \dots t_{i-\ell}], i - 1, c)$ // T_{i-1} *is* L*-type*
    else $L^*.\mathtt{append}(\,(a, r, [t_{i-1} \dots t_{i-\ell}], i, c)\,)$ // T_{i-1} *is* S*-type*
  end
end

length at most B, starting at the *beginning*, and lexicographically naming them along with all other substrings. Thereby, a long $\mathtt{S}^*$-substring is represented by a sequence of lexicographic names in the reduced string. The corresponding split tuples are formed in the same way as $\mathtt{S}^*$-substring tuples in P, they also overlap by one character, except that this character need not be $\mathtt{S}^*$-type. After the recursive call, long $\mathtt{S}^*$-substrings are correctly ordered among all other $\mathtt{S}^*$-substring due to suffix sorting, and split tuples can easily be discarded in line 10 as they do not correspond to any $\mathtt{S}^*$-suffix. The *d-critical* version of SAIS [122, Sect. 4] is a similar approach.

The second issue arises due to repeated re-insertions of payload characters into the PQ in line 14, possibly incurring quadratic I/O volume. This again is handled by splitting the $\mathtt{S}^*$-substrings, now starting at the *end*, into chunks of size D_0 or D ($D_0 \geq D$ indicating when to split at all, and $D \geq 1$ being the actual split length). Lines 10–15 of Algorithm 5 have to be replaced by Algorithm 6, which we will describe in the following. Let $\mathcal{D}$ be the set of splitting positions, counting first D_0 and then D characters backwards starting at each $\mathtt{S}^*$-suffix until the preceding $\mathtt{S}^*$-suffix is met. As before, for each $\mathtt{S}^*$-substring a tuple is stored in the S^* array, except that only the initial D_0 payload characters are copied. We call these items *seed tuples*. If an $\mathtt{S}^*$-substring consists of more than D_0 characters, a *continuation tuple* is stored in one of the two new arrays L or S in lines 3–4, depending on the type of its overlapping character. This overlapping character t_i will later be used together with its *repetition count* $\mathrm{rep}(i)$ to efficiently match continuation tuples with preceding tuples (see Section 5.4 for the definition of repetition counts); $\mathrm{rep}(i)$ is easily calculated while reading the text back-to-front. Along with both seed and continuation tuples we save a flag $\mathbb{1}_{i\in\mathcal{D}}$ marking whether a continuation exists.

Differing from Algorithm 5, in line 5 the PQ is initialized as empty and S^* will be processed as a stack. This modification separates the while loop into inducing from $\mathtt{S}^*$-suffixes in lines 7–8 and inducing from $\mathtt{L}$-suffixes in lines 10–15. The two induction sources are alternated between, with precedence depending on their top character: $Q_L.\mathtt{TopChar}() = t_i$ with $(t_i, r, \tau, i, c) = Q_L.\mathtt{Top}()$. Since $\mathtt{L}$-suffixes are smaller than $\mathtt{S}^*$-suffixes if they start with the same character, the while loop in 7–8 may only induce from $\mathtt{S}^*$-suffixes with the first character being smaller than $Q_L.\mathtt{TopChar}()$; otherwise, the while loop in 10–15 has precedence. When line 9 is reached, the loop in 10–15 extracts all suffixes from the PQ starting with a, after which the S^* stack must be checked again. In lines 11–14 the extracted tuple is handled as in Algorithm 5, however, when there is no preceding character t_{i-1} in the tuple and the continuation flag c is set, the tuple *underruns* and the matching continuation must be found. For each underrun tuple, the required position i and its assigned rank ρ_L is saved in the buffer M, which will be sorted and merged with the L array in line 16. Matching of the continuation tuple can be postponed up to the smallest rank at which a continued tuple may be reinserted into the PQ. This earliest rank is $m = \rho_L$, as set in line 9, because any reinsertion will have $r \geq \rho_L$, and thus the while loop 10–15 extracts exactly the r_a-th repetition bucket of a. Because continuation tuples must only be matched exactly once per repetition bucket, the continuation tuples are sorted by $(t_j, \mathrm{rep}(j), j)$, whereby L can be sequentially merged with M if M is kept sorted by the first component and L scanned as a stack.

In Subsection 5.6.3 we compute the optimal values for D_0 and D, and analyze the resulting

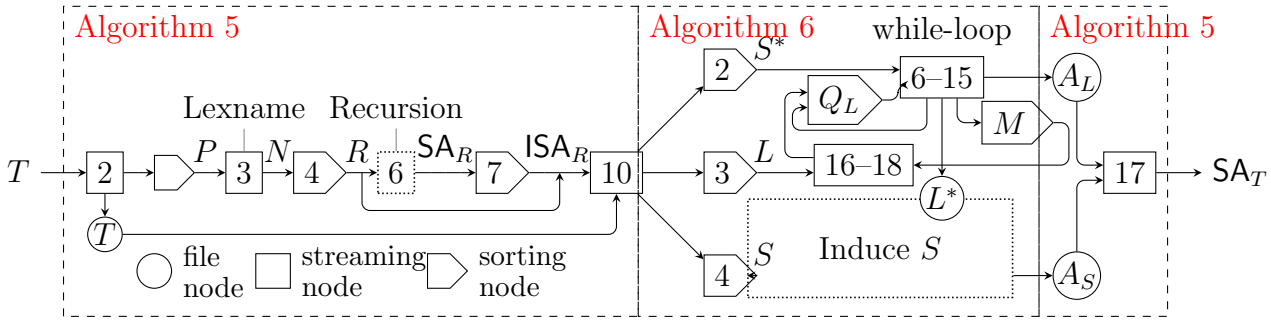


Figure 5.3.: Data flow graph of the algorithm; numbers refer to the line numbers of Algorithm 5 and Algorithm 6, respectively. The input T is read and saved to a file (2), while creating tuples. Sorting these tuples yields P, whose entries are lexicographically named in N (3) and sorted again by string index, resulting in R (4). If names are not unique in R, the algorithm calls itself recursively (6) to calculate SA_R. The suffix array is inverted into ISA_R (7) and resulting ranks are merged with T to create seed and continuation tuples (10), which are distributed into sorters (2,3,4) in Algorithm 6. The main while-loop (6–15) reads from array S^* and priority-queue Q_L. Depending on the calculation, the while-loop outputs final L-suffix order information into A_L, stores merge requests to M when tuples underrun, reinserts a shortened tuple, or it saves $\mathtt{L}^*$-tuples. Merge requests are handled by matching tuples from M and L (16–18) and reinserting into Q_L. When the while-loop for inducing L-suffixes finishes, the process is repeated with seed tuples from $\mathtt{L}^*$ and continuation tuples from S, yielding the final S-suffix order values in A_S. The output suffix array is constructed by merging A_L and A_S (17).

I/O volume.

5.6.3. I/O Analysis of eSAIS with Split Tuples

We now analyze the overall I/O performance of our algorithm and find the best splitting parameters D_0 and D, both under practical assumptions. We will focus on calculating the I/O volume processed by `Sort` in lines 2–4 and 16, and by the PQs.

For simplicity, we assume that there is only one elementary data type, disregarding the fact that characters can be smaller than indices, for instance. Thus a tuple is composed of multiple elements of equal size. We write $\textsc{Sort}(n)$ or $\textsc{Scan}(n)$ as the number of I/Os needed to sort or scan an array of n elements. For our practical experiments we assume $n \leq \frac{M^2}{B}$, and thus can relate $\textsc{Sort}(n) = 2\,\textsc{Scan}(n)$, which is equivalent to saying that n elements can be sorted with one in-memory merge step. With parameters $M = 2^{30}$ (1 GiB) and $B = 2^{10}$ (1 MiB), as used in our experiments, up to 2^{50} (1 PiB) elements can be sorted under this assumption. Furthermore, we also assume that the PQ has amortized I/O complexity $\textsc{Sort}(n)$ for sorting n elements, an assumption that is supported by preliminary experiments.

In the analysis we denote the length of $\mathtt{S}^*$-substrings *excluding* the overlapping character, thus the sum of their lengths is the string length. For further simplicity, we assume that line 15 of Algorithm 6 always stores continuation requests in M, and unmatched requests

are later discarded. Thus our analysis can ignore the boolean continuation variables.

For a broader view of the algorithm, we abstracted Algorithm 5 (including Algorithm 6) into a pipelined data flow graph in Figure 5.3.

Lemma 2. *To minimize I/O cost Algorithm 6 should use* $D = 3$ *and* $D_0 = 8$ *for splitting* $\mathtt{S}^*$*-strings.*

Proof. We first focus on the number of elements sorted and scanned by the algorithm for one $\mathtt{S}^*$-substring of long length $\ell = kD$ for $k \in \mathbb{N}_1$ when splitting by period D and set $D_0 := D$. In this proof we count amortized costs $\textsc{Sort}(1)$ per element sorted and $\textsc{Scan}(1)$ per element scanned. This is possible, as all $\frac{n}{\ell}$ $\mathtt{S}^*$-substrings are processed by the algorithm sequentially.

For one $\mathtt{S}^*$-substring the algorithm incurs $\textsc{Sort}(D{+}3)$ for sorting $\mathtt{S}^*$ (line 2) and $\textsc{Sort}((\frac{\ell}{D}{-}1){\cdot}(D{+}3))$ for sorting L and S (lines 3–4). In Q_L and Q_S a total of $\textsc{Sort}(\frac{\ell}{D}(\frac{1}{2}D(D{+}1)){+}\ell{\cdot}3)$ occurs due to repeated reinsertions into the PQs with decreasing lengths. The buffer M (line 16) requires at most $\textsc{Sort}((\frac{\ell}{D}-1)\cdot 2)$, while reading from L and S is already accounted for. Additionally, at most $\textsc{Scan}((D{-}1)+3)$ occurs when switching from Q_L to Q_S via L^*, as at least the first $\mathtt{S}$-character was removed. Overall, this is $\textsc{Sort}(\frac{\ell}{D}(\frac{1}{2}D^2+\frac{9}{2}D+5)-2)+\textsc{Scan}(D+2)$, which is minimized for $D = \sqrt{10} \approx 3.16$, when assuming $\textsc{Sort} = 2\,\textsc{Scan}$. Taking $D = 3$, we get at most $\textsc{Sort}(\frac{23}{3}\ell - 3) + \textsc{Scan}(5)$ per $\mathtt{S}^*$-substring.

Next, we determine the value of D_0 (as the length at when to start splitting by D). This offset is due to the base overhead of using continuations over just reinserting into the PQ. Given an $\mathtt{S}^*$-substring of length ℓ, repeated reinsertions without continuations would incur $\textsc{Sort}(\frac{1}{2}\ell(\ell+1)+\ell\cdot 3))$. By putting this quadratic cost in relation to the one with splitting by $D = 3$, we get that at length $\ell \approx 7.7$ the cost in both approaches is balanced. Therefore, we choose to start splitting at $D_0 = 8$. □

Theorem 3. *For a string of length* n *the I/O volume of Algorithm 5 is bounded by* $\textsc{Sort}(17n){+}\textsc{Scan}(9n)$*, when splitting with* $D = 3$ *and* $D_0 = 8$ *in Algorithm 6.*

Proof. To bound the I/O volume, we consider a string that consists of $\frac{n}{\ell}$ $\mathtt{S}^*$-substrings of length ℓ, and determine the maximum volume over all $2 \le \ell \le n$, where $\ell = 2$ is the smallest possible length of $\mathtt{S}^*$-substrings, due to exclusion of the overlapping character. Algorithm 5 needs $\textsc{Scan}(2n)$ to read T twice (in lines 2 and 10) and $\textsc{Sort}(n + \frac{n}{\ell}\cdot 2)$ to construct P in line 2, counting the overlapping character and excluding the boolean type, which can be encoded into i. In this $\textsc{Sort}$ the I/O volume of $\mathtt{Lexname}_{S^*}$ is already accounted for. Creating the reduced string R requires sorting of N, and thus $\textsc{Sort}(2\cdot\frac{n}{\ell})$ I/Os. Then the suffix array of the reduced string R with $|R| \le \frac{n}{\ell}$ is computed recursively and inverted using $\textsc{Sort}(2\cdot\frac{n}{\ell})$, or the names are already unique. After creating ISA_R, Algorithm 6 is used with the parameters derived in Lemma 2, incurring the amortized I/O cost calculated there for all $\frac{n}{\ell}$ $\mathtt{S}^*$-substrings. The final merging of A_L and A_S (line 17) needs $\textsc{Scan}(2n)$. In sum this is

$$\begin{aligned} V(n) \le\ & \textsc{Scan}(2n) + \textsc{Sort}(n+\tfrac{n}{\ell}\cdot 2) + \textsc{Sort}(\tfrac{n}{\ell}\cdot 2) \\ & + V(\tfrac{n}{\ell}) + \textsc{Sort}(\tfrac{n}{\ell}\cdot 2) + \textsc{Scan}(2n) \\ & + \tfrac{n}{\ell}\cdot \min\{\textsc{Sort}(\tfrac{23}{3}\ell - 3) + \textsc{Scan}(5)\,, \\ & \qquad \textsc{Sort}(\tfrac{1}{2}\ell(\ell+1)+\ell\cdot 3) + \textsc{Scan}(\tfrac{\ell}{2})\}\,. \end{aligned}$$

Maximizing $V(n,\ell)$ for $2 \le \ell \le n$ by $\ell = 2$, we get $V(n,\ell) \le V(n,2) \le \text{SORT}(8.5n) + \text{SCAN}(4.5n) + V(\frac{n}{2})$ and, solving the recurrence, $V(n,\ell) \le \text{SORT}(17n) + \text{SCAN}(9n)$. In Subsection 5.6.4 a worst-case string is constructed with $\mathtt{S}^*$-substrings of length $\ell = 2$. □

5.6.4. Experimental Study

We implemented the eSAIS algorithm in C++ using the external memory library STXXL [41]. This library provides efficient external memory sorting and a priority queue that is modeled after the design for cached memory [141]. Note that in STXXL all I/O operations bypass the operating system cache; therefore the experimental results are not influenced by system cache behavior. Our implementation and selected input files are available from http://tbingmann.de/2012/esais/.

Before describing the experiments, we highlight some details of the implementation. Most notably, STXXL does not support variable length structures, nor are we aware of a library with PQ that does. Therefore, in the implementation the tuples in the PQ and the associated arrays are of fixed length, and superfluous I/O transfer volume occurs. Due to fixed length structures, the results from the I/O analysis for the tuning parameter D cannot directly be used. We found that $D = D_0 = 3$ are good splitting values in practice. All results of the algorithms were verified using a suffix array checker [42, Sect. 8]. We designed the implementation to use an implicit sentinel instead of '$,' so that input containing zero bytes can be suffix sorted as well. Since our goal was to sort large inputs, the implementation can use different data types for array positions: usual 32-bit integers and a special 40-bit data type stored in five bytes. The input data type is also variable, we only experimented with usual 8-bit inputs, but the recursive levels work internally with the 32/40-bit data type. When sorting ASCII strings in memory, an efficient in-place radix sort [75] is used. Strings of larger data types are sorted in RAM using `g++` STL's version of introsort. The initial sort of short strings into P was implemented using a variable length tuple sorter.

We chose a wide variety of large inputs, both artificial and from real-world applications:

Wikipedia is an XML dump of the most recent version of all pages in the English Wikipedia, which is obtainable from http://dumps.wikimedia.org/; our dump is dated `enwiki-20120601`.

Gutenberg is a concatenation of all ASCII text documents from http://www.gutenberg.org/robot/harvest as available in September 2012.

Human Genome consists of all DNA files from the UCSC human genome assembly "hg19" downloadable from http://genome.ucsc.edu/. The files were stripped of all characters but $\{\mathtt{A}, \mathtt{G}, \mathtt{C}, \mathtt{T}, \mathtt{N}\}$ and normalized to upper-case. Note that this input contains very long sequences of unknown `N` placeholders, which influences the LCPs.

Pi are the decimals of π, written as ASCII digits and starting with "3.1415."

Skyline is an artificial string for which eSAIS has maximum recursion depth. To achieve this, the string's suffixes must have type sequence `LSLS`...`LS` at each level of recursion. Such a string can be constructed for a length $n = 2^p$, $p \ge 1$, using the alphabet $\Sigma = [\,\mathtt{\$}, \sigma_1, \ldots, \sigma_p\,]$ and the grammar $\{S \to T_1\mathtt{\$},\ T_i \to T_{i+1}\sigma_i T_{i+1}$ for $i = 1, \ldots, p-1$ and $T_p \to \sigma_p\}$. For $p = 4$ and $\Sigma = [\,\mathtt{\$}, \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\,]$, we get `dcdbdcdadcdbdcd$`; for the test runs we replaced `$` with σ_0.

The input Skyline is generated depending on the experiment size, all other inputs are cut to size.

Our main experimental **platform E**, see Section 3.2, was a cluster computer, with one node exclusively allocated when running a test instance. In all tests only one core of the processor is used. Each node has 850 GiB of available disk space striped with RAID 0 across four local disks of size 250 GiB; the rest is reserved by the system. We limited the main memory usage of the algorithms to 1 GiB of RAM, and used a block size of 1 MiB. The block size was optimized in preliminary experiments.

Due to the limited local disk space in the cluster computer, we chose to run some additional, larger experiments on **platform F**, see Section 3.2. The main memory usage was limited to 4 GiB RAM, we kept the block size at 1 MiB and up to six local SATA disk with 1 TB of local space were available. Programs on both platforms were compiled using `g++` 4.4.6 with `-O3` and native architecture optimization.

As noted in the introduction, the previously fastest EM suffix sorter is DC3 [42]. We adapted and optimized the original source code[1], which is already implemented using STXXL, to our current setup and larger data types. An implementation of DC7 exists that is reported to be about 20% faster in the special case of human DNA [172], but we did not include it in our experiments.

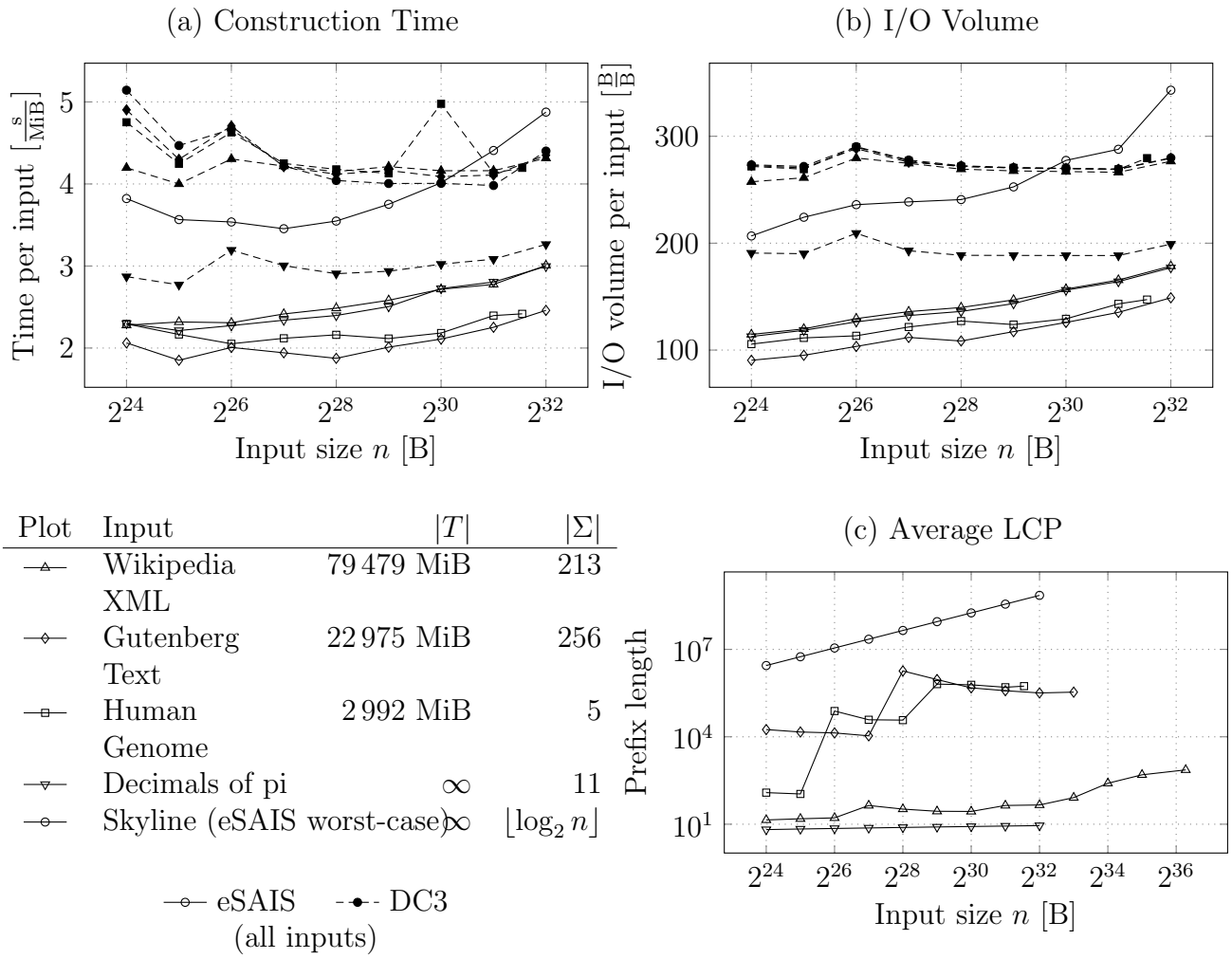
Figure 5.4 shows the construction time and I/O volume of eSAIS and DC3 on platform E using 32-bit keys. The two algorithms eSAIS (open bullets) and DC3 (filled bullets) were run on prefixes $T[0, 2^k)$ of all five inputs, with only Skyline being generated specifically for each size. In total these plots took 3.2 computing days and over 16.8 TiB of I/O volume, which is why only one run was performed for each of the 90 test instances.

For all real-world inputs eSAIS's construction time is about half of DC3's. The I/O volume required by eSAIS is also only about 60% of the volume of DC3. The two artificial inputs exhibit the extreme results they were designed to provoke: Pi is random input with short LCPs, which is an easy case for DC3. Nevertheless, eSAIS is still faster, but not twice as fast. The results from eSAIS's worst-case Skyline show another extreme: eSAIS has highest construction time on its worst input, whereas DC3 is moderately fast because Skyline can efficiently be sorted by triples. The high I/O volume of eSAIS for Skyline is due to its maximum recursion depth, reducing the string only by $\frac{1}{2}$ and filling the PQ with $\frac{n}{2}$ items on each level. The PQ implementation requires more I/O volume than sorting, because it recursively combines short runs to keep the arity of mergers in main memory small. Even though DC3 reduces by $\frac{2}{3}$, the recursion depth is limited by $\log_3 n$ and sorting is more straightforward.

Besides the basic eSAIS algorithm, we also implemented a variant which "discards" sequences of multiple unique names from the reduced string prior to recursion [42, 138]. However, we discovered that this optimization has much smaller effect in eSAIS than in other suffix sorters (see Figure 5.5 (a)-(d)). This is probably due to the induced sorting algorithm already adapting very efficiently to the input string's characteristics.

Figure 5.5 (a)-(d) shows the results of all three variants of the algorithms on the real-world inputs run on platform E.

[1] http://algo2.iti.kit.edu/dementiev/esuffix/docu/



Plot	Input	$\lvert T\rvert$	$\lvert\Sigma\rvert$
—△—	Wikipedia XML	79 479 MiB	213
—◇—	Gutenberg Text	22 975 MiB	256
—□—	Human Genome	2 992 MiB	5
—▽—	Decimals of pi	∞	11
—○—	Skyline (eSAIS worst-case)	∞	$\lfloor \log_2 n \rfloor$

Figure 5.4.: The first row shows construction time and I/O volume of eSAIS (open bullets) and DC3 (filled bullets) on experimental platform E. The second row shows selected characteristics of the input strings.

To exhibit experiments with building large suffix arrays, we configured the algorithms to use 40-bit positions on platform E. Figure 5.5 (c)-(d) show results for the Wikipedia and Gutenberg input only up to 2^{33}, because larger instances require more local disk space than available at the node of the cluster computer. On average over all tests instances of Wikipedia, calculation using 40-bit positions take about 33% more construction time and the expected 25% more I/O volume.

The size of suffix arrays that can be built on platform E was limited by the local disk space; we therefore determined the maximum disk allocation required. Table 5.1 shows the average maximum disk allocation measured empirically over our test inputs for 32-bit and 40-bit offset data types.

On platform F we had the necessary 4 TiB disk space required to process the full Wikipedia instance, and these results are shown in Figure 5.6. The maximum size of the in-memory

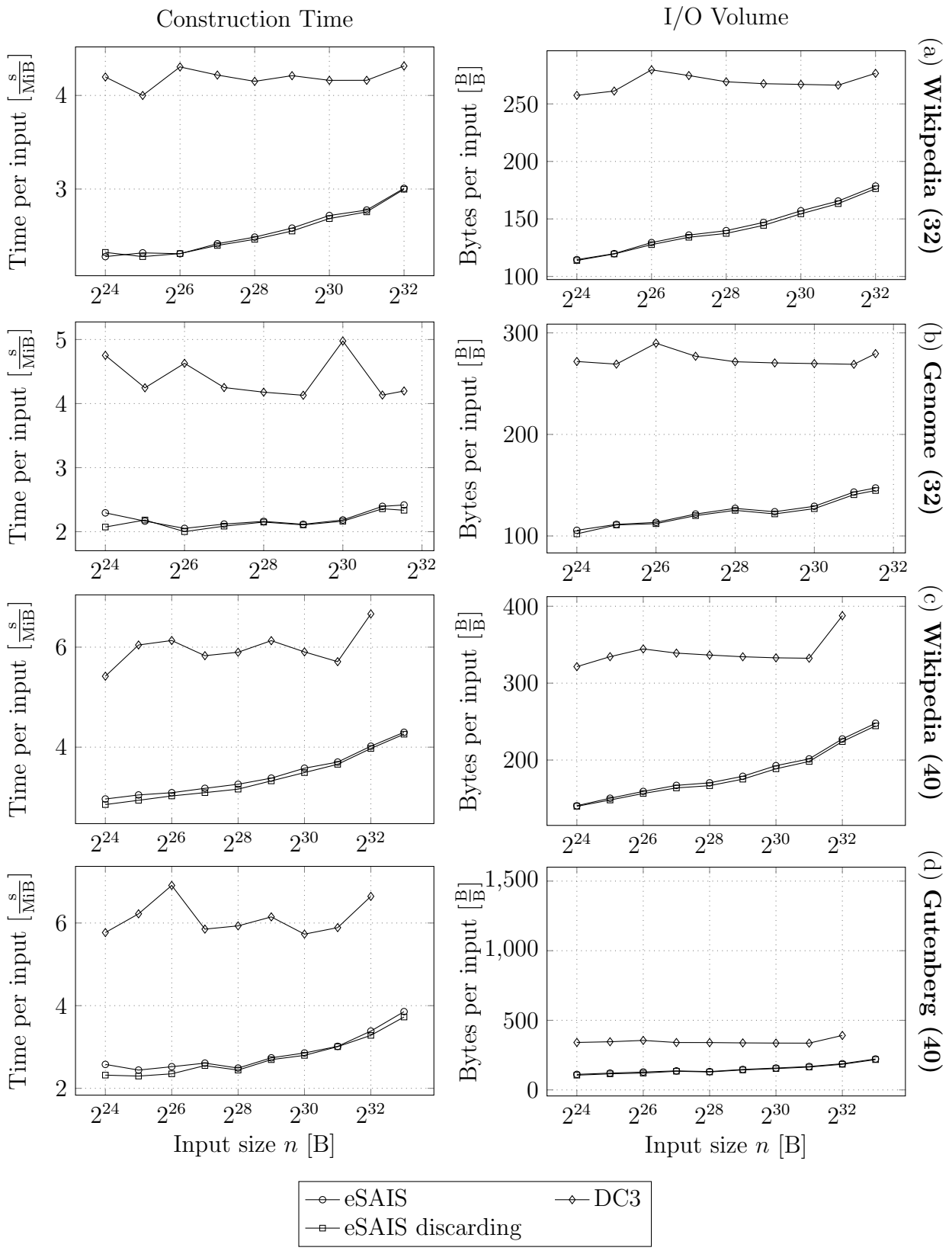


Figure 5.5.: Subfigures (a)-(d) show construction time and I/O volume of all three implementations run on platform E for three different inputs. Subfigures (a)-(b) use 32-bit positions, while (c)-(d) runs with 40-bit. On the right hand side, $\frac{\text{B}}{\text{B}}$ indicates I/O volume in bytes per input byte.

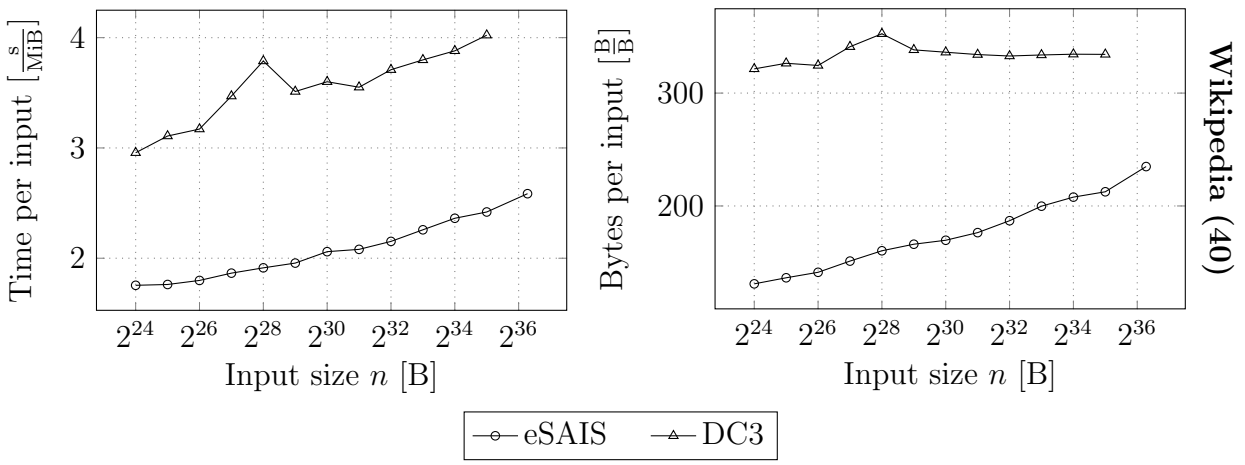


Figure 5.6.: Measured construction time and I/O volume of two implementations is shown for the largest test instance Wikipedia run on platform F using 40-bit positions

	eSAIS	DC3
32-bit	$25n$	$46n$
40-bit	$28n$	$58n$

Table 5.1.: Maximum disk allocation in bytes required by the algorithms, averaged and rounded over all our inputs

RMQ structure was only about 12 MiB. Sorting of the whole Wikipedia input with eSAIS took 2.4 days and 18 TiB I/O volume.

CHAPTER 6

Breadth First Search on Massive Graphs

Breadth first search (BFS) is a fundamental graph traversal strategy. It decomposes the input graph $G = (V, E)$ of n nodes and m edges into at most n levels where level i comprises all nodes that can be reached from a designated source s via a path of i edges, but cannot be reached using less than i edges.

Large graphs arise naturally in many applications and very often we need to traverse these graphs for solving optimization problems. Typical real-world applications of BFS on large graphs (and some of its generalizations like shortest paths or A^*) include crawling and analyzing the WWW [118, 156], route planning using small navigation devices with flash memory cards [59], state space exploration [46], etc.

While modern processor speeds are measured in GHz, average hard disk latencies are in the range of a few milliseconds. Hence, the cost of accessing a data element from the hard-disk (an I/O) is around a million times more than the cost of an instruction, see Section 2.2 for details. Therefore, it comes as no surprise that the I/Os dominate the runtimes of even basic graph traversal strategies like BFS on large graphs, making their standard implementations non-viable. One way to ease this problem can be to represent the graph [25, 26] in a more compact way that minimizes the I/Os required by the standard algorithms. However, such approaches work only for graphs with good separators. The other approach that we consider in this chapter relies on new algorithmic ideas capturing the I/Os into the performance metric of the computation model. In order to do so, we need to look beyond the traditional RAM model which assumes an unbounded amount of memory with unit cost access to any location.

Ajwani et al. [5] showed that the randomized variant of the $o(n)$ I/O algorithm of Mehlhorn and Meyer [104] (MM_BFS) can compute the BFS level decomposition for large graphs (around a billion edges) in a *few hours* for small diameter graphs and a *few days* for large diameter graphs. We improve upon their implementation of this algorithm by reducing the overhead associated with each BFS level, thereby improving the results for large diameter graphs which are more difficult for BFS traversal in external memory. Also, we present

the implementation of the deterministic variant of MM_BFS and show that in most cases, it outperforms the randomized variant. The running time for BFS traversal is further improved with a heuristic that preserves the worst case guarantees of MM_BFS. Together, they reduce the time for BFS on large diameter graphs from *days* shown in [5] to *hours*. In particular, on line graphs with random layout on disks, our implementation of the deterministic variant of MM_BFS with the proposed heuristic is more than 75 times faster than the previous best result for the randomized variant of MM_BFS in [5].

References. The contents of this chapter is based on the joint work with Ulrich Meyer and Deepak Ajwani, who has also literally used parts of the original publication in his PhD thesis [4, 6]. Most of the wording of the original publication is preserved.

6.1. Algorithms

BFS is well-understood in the RAM model. There exists a simple linear time algorithm [38] (hereafter refered as IM_BFS) for the BFS traversal in a graph. IM_BFS keeps a set of appropriate candidate nodes for the next vertex to be visited in a FIFO queue Q. Furthermore, in order to find out the unvisited neighbours of a node from its adjacency list, it marks the nodes as either visited or unvisited. Unfortunately as reported in [5], even when half of the graph fits in the main memory, the running time of this algorithm deviates significantly from the predicted RAM performance (*hours* as compared to *minutes*) and for massive graphs, such approaches are simply non-viable. As discussed before, the main cause for such a poor performance of this algorithm on massive graphs is the number of I/Os it incurs. Remembering visited nodes needs $\Theta(m)$ I/Os in the worst case and the unstructured indexed access to adjacency lists may result in $\Theta(n)$ I/Os.

The algorithm by Munagala and Ranade [116] (refered as MR_BFS) ignores the second problem but addresses the first by exploiting the fact that the neighbours of a node in BFS level i are all in BFS levels $i+1$, i or $i-1$. Thus, the set of nodes in level $i+1$ can be computed by removing all nodes in level i and $i-1$ from the neighbours of nodes in level i. The resulting worst-case I/O-bound is $O(n + \textsc{Sort}(n+m))$.

Mehlhorn and Meyer suggested another approach [104] (MM_BFS) which involves a preprocessing phase to restructure the adjacency lists of the graph representation. It groups the vertices of the input graph into disjoint clusters of small diameter and stores the adjacency lists of the nodes in a cluster contiguously on the disk. Thereafter, an appropriately modified version of MR_BFS is run. MM_BFS exploits the fact that whenever the first node of a cluster is visited then the remaining nodes of this cluster will be reached soon after. By spending only one random access (and possibly, some sequential accesses depending on cluster size) in order to load the whole cluster and then keeping the cluster data in some efficiently accessible data structure (pool) until it is all used up, on sparse graphs the total amount of I/O can be reduced by a factor of up to $\sqrt{B}$. The neighbouring nodes of a BFS level can be computed simply by scanning the pool and not the whole graph. Though some edges may be scanned more often in the pool, unstructured I/O in order to fetch adjacency lists is considerably reduced, thereby saving the total number of I/Os. The preprocessing of MM_BFS comes in two variants: randomized and deterministic (referred as MM_BFS_R

and MM_BFS_D, respectively). In the randomized variant, the input graph is partitioned by choosing master nodes independently and uniformly at random with a probability μ and running a BFS like routine with joint adjacency list queries from these master nodes "in parallel".

The deterministic variant first builds a spanning tree for G and then constructs an Euler tour $\mathcal{T}$ for the tree. Next, each node v is assigned the rank in $\mathcal{T}$ of the first occurrence of the node (by scanning $\mathcal{T}$ and a sorting step). We denote this value as $r(v)$. $\mathcal{T}$ has length $2V-1$; so $r(v) \in [0; 2V-2]$. Note that if for two nodes u and v, the values $r(v)$ and $r(u)$ differ by d, then d is an upper bound on the distance between their BFS level. Therefore, we chop the Euler tour into chunks of $\sqrt{B}$ nodes and store the adjacency lists of the nodes in the chunk consecutively as a cluster.

The randomized variant incurs an expected number of $O(\sqrt{n \cdot (n+m) \cdot \log(n)/B} + \textsc{Sort}(n+m))$ I/Os, while the deterministic variant incurs $O(\sqrt{n \cdot (n+m)/B} + \textsc{Sort}(n+m) + ST(n,m))$ I/Os, where $ST(n,m)$ is the number of I/Os required for computing a spanning tree of a graph with n nodes and m edges. Arge et al. [10] show an upper bound of $O((1 + \log\log(D \cdot B \cdot n/m)) \cdot \textsc{Sort}(n+m))$ I/Os for computing such a spanning tree.

Brodal et al. [30] gave a cache oblivious algorithm for BFS achieving the same worst case I/O bounds as MM_BFS_D. Their preprocessing is similar to that in MM_BFS_D, except that it produces a hierarchical clustering using the cache oblivious algorithms for sorting, spanning tree, Euler tour and list ranking. The BFS phase uses a data-structure that maintains a hierarchy of pools and provides the set of neighbours of the nodes in a BFS level efficiently.

The other external memory algorithms for BFS are restricted to special graphs classes like trees [31], grid graphs [11], planar graphs [97], outer-planar graphs [95], and graphs of bounded tree width [96].

6.1.1. Related Work

Ajwani et al. [5] showed that the usage of the two external memory algorithms MR_BFS and MM_BFS_R along with disk parallelism and pipelining can alleviate the I/O bottleneck of BFS on many large sparse graph classes, thereby making the BFS viable for these graphs. Even with just a single disk, they computed a BFS level decomposition of small diameter large graphs (around 256 million nodes and a billion edges) in a few *hours* and moderate and large diameter graphs in a few *days*, which otherwise would have taken a few *months* with IM_BFS. As for their relative comparison, MR_BFS performs better than MM_BFS_R on small-diameter random graphs saving a few *hours*. However, the better asymptotic worst-case I/O complexity of MM_BFS helps it to outperform MR_BFS for large diameter sparse graphs (computing in a few *days* versus a few *months*), where MR_BFS incurs close to its worst case of $\Omega(n)$ I/Os.

Independently, Christiani [37] gave a prototypical implementation of MR_BFS, MM_BFS_R as well as MM_BFS_D and reached similar conclusions regarding the comparative performance between MR_BFS and MM_BFS_R. Though their implementation of MR_BFS and MM_BFS_R is competetive and on some graph classes even better than [5], their experiments were mainly carried out on smaller graphs (up to 50 million nodes). Since their main goal was to design cache oblivious BFS, they used cache oblivious algorithms for sorting,

minimum spanning tree and list ranking even for MM_BFS_D. As we discuss later, these algorithms slow down the deterministic preprocessing in practice, even though they have the same asymptotic I/O complexity as their external memory counterparts.

6.1.2. Our Contribution

Our contributions are the following:

- We improve upon the MR_BFS and MM_BFS_R implementation described in [5] by reducing the computational overhead associated with each BFS level, thereby improving the results for large diameter graphs.
- We discuss the various choices made for a fast MM_BFS_D implementation. This involved experimenting with various available external memory connected component and minimum spanning tree algorithms. Our partial re-implementation of the list ranking algorithm of [157] adapting it to the STXXL framework outperforms the other list ranking algorithms for the sizes of our interest. As for the Euler tour in the deterministic preprocessing, we compute the cyclic order of edges around the nodes using the STXXL sorting.
- We conduct a comparative study of MM_BFS_D with other external memory BFS algorithms and show that for most graph classes, MM_BFS_D outperforms MM_BFS_R. Also, we compared our BFS implementations with Christiani's implementations [37], which have some cache-oblivious subroutines. This gives us some idea of the loss factor that we will have to face for the performance of cache-oblivious BFS.
- We propose a heuristic for maintaining the pool in the BFS phase of MM_BFS. This heuristic improves the runtime of MM_BFS in practice, while preserving the worst case I/O bounds of MM_BFS.
- Putting everything together, we show that the BFS traversal can also be done on moderate and large diameter graphs in a few *hours*, which would have taken the implementations of [5] and [37] several *days* and IM_BFS several *months*. Also, on low diameter graphs, the time taken by our improved MR_BFS is around one-third of that in [5]. Towards the end, we summarize our results (Table 6.13) by giving the state of the art implementations of external memory BFS on different graph classes.

6.1.3. Improvements of MR_BFS and MM_BFS_R

The computation of each level of MR_BFS involves sorting and scanning of neighbours of the nodes in the previous level. Even if there are very few elements to be sorted, there is a certain overhead associated with initializing the external sorters. In particular, while the STXXL stream sorter (with the flag DSTXXL_SMALL_INPUT_PSORT_OPT) does not incur an I/O for sorting less than B elements, it still requires to allocate some memory and does some computation for initialization. This overhead accumulates over all levels and for

large diameter graphs, it dominates the running time. This problem is also inherited by the BFS phase of MM_BFS. Since in the pipelined implementation of [5], we do not know in advance the exact number of elements to be sorted, we can't switch between the external and the internal sorter so easily. In order to get around this problem, we first buffer the first B elements and initialize the external sorter only when the buffer is full. Otherwise, we sort it internally.

In addition to this, we make the graph representation for MR_BFS more compact. Except the source and the destination node pair, no other information is stored with the edges.

6.2. Algorithm Design of MM_BFS_D

There are three main components for the deterministic variant of MM_BFS – sorting, connected components/ minimum spanning tree, and list ranking. The MM_BFS_D implementation of Christiani [37] uses the cache-oblivious lazy funnel-sort algorithm [28] (CO_sort). As Table 6.1 shows, the STXXL stream sort (STXXL_sort) proved to be much faster on external data. This is in line with the observations of Brodal et al. [29], where it is shown that an external memory sorting algorithm in the library TPIE [166] is better than their carefully implemented cache-oblivious sorting algorithm, when run on disk.

Regarding connected components and minimum spanning forest, Christiani's implementations [37] use the cache oblivious algorithm given in [1] (CO_MST). Empirically, we found that the external memory implementation of [43] (EM_MST)[1] performs better than the one in [1]. Table 6.2 shows the total time required for their deterministic preprocessing using CO_MST and EM_MST on low diameter random graphs and on high diameter line graphs.

n	CO_sort	STXXL_sort
256×10^6	21	8
512×10^6	46	13
1024×10^6	96	25

Table 6.1.: Timing in minutes for sorting n elements using CO_sort and with using STXXL_sort

As for list ranking, we found Sibeyn's algorithm in [157] promising as it has low constant factors in its I/O complexity. Sibeyn's implementation relies on the operating system for I/Os and does not guarantee that the top blocks of all the stacks remain in the internal memory, which is a necessary assumption for the asymptotic analysis of the algorithm. Besides, its reliance on internal arrays and swap space puts a restriction on the size of the lists it can rank. The deeper integration of the algorithm in the STXXL framework, using the STXXL stacks and vectors in particular, made it possible to obtain a scalable solution,

[1] Though [43] uses randomization of node order, we still denote this variant of preprocessing deterministic for consistency. Any other EM MST algorithm can be used instead.

Graph class	CO_MST	EM_MST
Random graph; $n = 2^{28}$, $m = 2^{30}$	107	35
Line graph with contiguous disk layout; (Simple Line) $n = 2^{28}$	38	16
Line graph with random disk layout (Random Line); $n = 2^{28}$	47	22

Table 6.2.: Timing in hours required by deterministic preprocessing by Christiani's implementation using CO_MST and EM_MST.

Graph class	n	m	Long clusters	Random clusters
Grid($2^{14} \times 2^{14}$)	2^{28}	2^{29}	51	28

Table 6.3.: Time taken (in hours) by the BFS phase of MM_BFS_D with long and random clustering

which could handle graph instances of the size we require while keeping the theoretical worst case bounds.

Christiani uses the algorithm in [36] for list ranking the Euler tour. While his cache oblivious list ranking implementation takes around 14.3 *hours* for ranking 2^{29} element random list using 3 GB RAM, our adaptation of Sibeyn's algorithm takes less than 40 *minutes* in the same setting.

To summarize, our STXXL based implementation of MM_BFS_D uses our adaptation of [157] for list ranking the Euler tour around the minimum spanning tree computed by EM_MST. The Euler tour is then chopped into sets of $\sqrt{B}$ consecutive nodes which after duplicate removal gives the requisite graph partitioning. The BFS phase remains similar to MM_BFS_R.

Quality of the spanning tree The quality of the spanning tree computed can have a significant impact on the clustering and the disk layout of the adjacency list after the deterministic preprocessing, and consequently on the BFS phase. For instance, in the case of grid graph, a spanning tree containing a list with elements in a snake-like row major order produces long and narrow clusters, while a "random" spanning tree is likely to result in clusters with low diameters. Such a "random" spanning tree can be attained by assigning random weights to the edges of the graph and then computing a minimum spanning tree or by randomly permuting the indices of the nodes. The nodes in the long and narrow clusters tend to stay longer in the pool and therefore, their adjacency lists are scanned more often. This causes the pool to grow external and results in larger I/O volume. On the other hand, low diameter clusters are evicted from the pool sooner and are scanned less often reducing the I/O volume of the BFS phase. Consequently as Table 6.3 shows, the BFS phase of MM_BFS_D takes only 28 hours with clusters produced by "random" spanning tree, while it takes 51 hours with long and narrow clusters.

6.2.1. A Heuristic for Maintaining the Pool

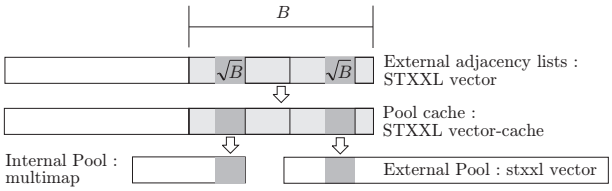


Figure 6.1.: Schema depicting the implementation of our heuristic

As noted in Section 6.1, the asymptotic improvement and the performance gain in MM_BFS over MR_BFS is obtained by decomposing the graph into low diameter clusters and maintaining an efficiently accessible pool of adjacency lists which will be required in the next few levels. Whenever the first node of a cluster is visited during the BFS, the remaining nodes of this cluster will be reached soon after and hence, this cluster is loaded into the pool. For computing the neighbours of the nodes in the current level, we just need to scan the pool and not the entire graph. Efficient management of this pool is thus, crucial for the performance of MM_BFS. In this subsection, we propose a heuristic for efficient management of the pool, while keeping the worst case I/O bounds of MM_BFS.

For many large diameter graphs, the pool fits into the internal memory most of the time. However, even if the number of edges in the pool is not so large, scanning all the edges in the pool for each level can be computationally quite expensive. Hence, we keep a portion of the pool that fits in the internal memory as a multi-map hash table. Given a node as a key, it returns all the nodes adjacent to the current node. Thus, to get the neighbours of a set of nodes we just query the hash function for those nodes and delete them from the hash table. For loading the cluster, we just insert all the adjacency lists of the cluster in the hash table, unless the hash table has already $O(M)$ elements.

Recall that after the deterministic preprocessing, the elements are stored on the disk in the order in which they appear on the Euler tour around a spanning tree of the input graph. The Euler tour is then chopped into clusters with $\sqrt{B}$ elements (before the duplicate removal) ensuring that the maximum distance between any two nodes in the cluster is at most $\sqrt{B} - 1$. However, the fact that the contiguous elements on the disk are also closer in terms of BFS levels is not restricted to intra-cluster adjacency lists. The adjacency lists that come alongside the requisite cluster will also be required soon and by caching these other adjacency lists, we can save the I/Os in the future. This caching is particularly beneficial when the pool fits in the internal memory. Note that we still load the $\sqrt{B}$ node clusters in the pool, but keep the remaining elements of the block in the pool-cache. For the line graphs, this means that we load the $\sqrt{B}$ nodes in the internal pool, while keeping the remaining $O(B)$ adjacency lists which we get in the same block, in the pool-cache, thereby reducing the I/O complexity for the BFS traversal on line graphs to the computation of a spanning tree.

We represent the adjacency lists of nodes in the graph as an STXXL vector. STXXL already provides a fully associative vector-cache with every vector. Before doing an I/O

for loading a block of elements from the vector, it first checks if the block is already there in the vector-cache. If so, it avoids the I/O loading the elements from the cache instead. Increasing the vector-cache size of the adjacency list vector with a layout computed by the deterministic preprocessing and choosing the replacement policy to be LRU provides us with an implementation of the pool-cache. Figure 6.1 depicts the implementation of our heuristic.

6.2.2. Experimental Study

Configuration. We have implemented the algorithms in C++ using the g++ 4.02 compiler (optimization level -O3) on the *GNU/Linux* distribution with a 2.6 *kernel* and the external memory library STXXL version 0.77. Our experimental **platform G**, see Section 3.2, has 250 GB Seagate Baracuda hard-disks [152]. These hard-disks have 8 MB buffer cache. The average seek time for read and write is 8.0 and 9.0 msec, respectively, while the sustained data transfer rate for outer zone (maximum) is 65 MByte/s. This means that for graphs with 2^{28} nodes, n random read and write I/Os will take around 600 and 675 hours, respectively. In order to compare better with the results of [5], we restrict the available memory to 1 GB for our experiments and use only one processor and one disk.

First, we show the comparison between improved MM_BFS_R and MR_BFS with the corresponding implementations in [5]. Then we compare our implementation of MM_BFS_D (without our heuristic) with Christiani's implementation based on cache-oblivious routines. Finally, we look at the relative performance of improved versions of MR_BFS, MM_BFS_R and MM_BFS_D. We summarize this section by highlighting the best algorithms for each graph class and its run-time. Note that some of the results shown in this section have been interpolated using the symmetry in the graph structure.

Graph classes. We consider the same graph classes as in [5] – Random, Grid, MR_worst graph, MM_worst graph, line graphs with different layouts and the webgraph. They cover a broad spectrum of different performances of external memory BFS algorithms.
Random graph: On a n node graph, we randomly select m edges with replacement (i.e., m times selecting a source and target node such that source $\neq$ target) and remove the duplicate edges to obtain random graphs.
MR_worst graph: This graph consists of B levels, each having $\frac{n}{B}$ nodes, except the level 0 which contains only the source node. The edges are randomly distributed between consecutive levels, such that these B levels approximate the BFS levels. The initial layout of the nodes on the disk is random. This graph causes MR_BFS to incur its worst case of $\Omega(n)$ I/Os.
Grid graph ($x \times y$): It consists of a $x \times y$ grid, with edges joining the neighouring nodes in the grid.
MM BFS worst graph: This graph causes MM_BFS_R to incur its worst case of $\Theta(n \cdot \sqrt{\frac{\log n}{B}} + \text{Sort}(n))$ I/Os.
Line graphs: A line graph consists of n nodes and $n-1$ edges such that there exist two nodes u and v, with the path from u to v consisting of all the $n-1$ edges. We took two different initial layouts – simple, in which all blocks consists of B consecutively lined nodes and the

random in which the arrangement of nodes on disk is given by a random permutation.
Web graph: As an instance of a real world graph, we consider an actual crawl of the world wide web [165], where an edge represents a hyperlink between two sites. This graph has around 130 million nodes and 1.4 billion edges. It has a core which consists of most of its nodes and behaves like a random graph.

Graph class	n	m	MM_BFS_R of [5]		Impr. MM_BFS_R	
			Phase 1	Phase 2	Phase 1	Phase 2
Random	2^{28}	2^{30}	5.1	4.5	5.2	3.8
MM_worst	$\sim 4.3 \cdot 10^7$	$\sim 4.3 \cdot 10^7$	6.7	26	5.2	18
MR_worst	2^{28}	2^{30}	5.1	45	4.3	40
Grid ($2^{14} \times 2^{14}$)	2^{28}	2^{29}	7.3	47	4.4	26
Simple Line	2^{28}	$2^{28} - 1$	85	191	55	2.9
Random Line	2^{28}	$2^{28} - 1$	81	203	64	25
Webgraph	$\sim 1.4 \cdot 10^8$	$\sim 1.2 \cdot 10^9$	6.2	3.2	5.8	2.8

Table 6.4.: Timing in hours taken for BFS by the two MM_BFS_R implementations

Graph class	n	m	MM_BFS_R of [5]		Impr. MM_BFS_R	
			I/O wait	Total	I/O wait	Total
MM_worst	$\sim 4.3 \cdot 10^7$	$\sim 4.3 \cdot 10^7$	13	26	16	18
Grid ($2^{14} \times 2^{14}$)	2^{28}	2^{29}	46	47	24	26
Simple Line	2^{28}	$2^{28} - 1$	0.5	191	0.05	2.9
Random Line	2^{28}	$2^{28} - 1$	21	203	21	25

Table 6.5.: I/O wait time and the total time in hours for the BFS phase of the two MM_BFS_R implementations on moderate to large diameter graphs

Graph class	n	m	MR_BFS of [5]		Impr. MR_BFS	
			I/O wait	Total	I/O wait	Total
Random	2^{28}	2^{30}	2.4	3.4	1.2	1.4
Webgraph	$\sim 135 \times 10^6$	$\sim 1.18 \times 10^9$	3.7	4.0	2.5	2.6
MM_worst	$\sim 42.6 \times 10^6$	$\sim 42.6 \times 10^6$	25	25	13	13
Simple line	2^{28}	$2^{28} - 1$	0.6	10.2	0.06	0.4

Table 6.6.: Timing in hours taken for BFS by the two MR_BFS implementations

Comparing MM_BFS_R. Table 6.4 shows the improvement that we achieved in MM_BFS_R. As Table 6.5 shows, these improvements are achieved by reducing the computation time per level in the BFS phase. On I/O bound random graphs, the improvement is just around 15%,

Graph class	n	m	Christiani's implementation	Our implementation
Random graph	2^{28}	2^{30}	107	5.2
Random Line	2^{28}	$2^{28}-1$	47	3.2

Table 6.7.: Timing in hours for computing the deterministic preprocessing of MM_BFS by the two implementations of MM_BFS_D

Graph class	n	m	Christiani's implementation	Our implementation
Random graph	2^{28}	2^{30}	16	3.4
Random Line	2^{28}	$2^{28}-1$	0.5	2.8

Table 6.8.: Timing in hours for the BFS phase of MM_BFS by the two implementations of MM_BFS_D (without heuristic)

Graph class	MR_BFS	MM_BFS_R	MM_BFS_D
Random graph	1.4	8.9	8.7
Random Line	4756	89	3.6

Table 6.9.: Timing in hours taken by our implementations of different external memory BFS algorithms.

Graph class	n	m	Randomized	Deterministic
Random graph	2^{28}	2^{30}	500	630
Random Line	2^{28}	$2^{28}-1$	10500	480

Table 6.10.: I/O volume (in GB) required in the preprocessing phase by the two variants of MM_BFS

Graph class	n	m	Randomized	Deterministic
Random graph	2^{28}	2^{30}	5.2	5.2
Random Line	2^{28}	$2^{28}-1$	64	3.2

Table 6.11.: Preprocessing time (in hours) required by the two variants of MM_BFS, with the heuristic

while on computation bound line graphs with random disk layout, we improve the running time of the BFS phase from around 200 hours to 25 hours. Our implementation of the randomized preprocessing in the case of the simple line graphs additionally benefits from the

way clusters are laid out on the disk as this layout reflects the order in which the nodes are visited by the BFS. This reduces the total running time for the BFS phase of MM_BFS_R on simple line graphs from around 190 hours to 2.9 hours. The effects of caching are also seen in the I/O bound BFS phase on the grid ($2^{14} \times 2^{14}$) graphs, where the I/O wait time decreases from 46 hours to 24 hours.

Comparing MR_BFS. Improvements in MR_BFS are shown in the Table 6.6. On random graphs where MR_BFS performs better than the other algorithms, we improve the runtime from 3.4 hours to 1.4 hours. Similarly for the web-crawl based graph, the running time reduces from 4.0 hours to 2.6 hours. The other graph class where MR_BFS outperforms MM_BFS_R is the MM_worst graph and here again, we improve the performance from around 25 hours to 13 hours.

Penalty for cache obliviousness. We compared the performance of our implementation of MM_BFS_D (without the heuristic) with Christiani's implementation [37] based on cache-oblivious subroutines. Table 6.7 and Table 6.8 show the results of the comparison on the two extreme graph classes - random graphs and line graphs with random layout on disk - for the preprocessing and the BFS phase respectively. We observed that on both graph classes, the preprocessing time required by our implementation is significantly less than the one by Christiani. While pipelining helps the BFS phase of our implementation on random graphs, it becomes a liability on line graphs as it brings extra computation cost per level.

We suspect that these performance losses are inherent in cache-oblivious algorithms to a certain extent and will be carried over to the cache-oblivious BFS implementation.

Comparing MM_BFS_D with other external memory BFS algorithm implementations. Table 6.9 shows the performance of our implementations of different external memory BFS algorithms with the heuristic. While MR_BFS performs better than the other two on random graphs saving a few *hours*, our implementation of MM_BFS_D with the heuristic outperforms MR_BFS and MM_BFS_R on line graphs with random layout on disk saving a few *months* and a few *days*, respectively. Random line graphs are an example of a tough input for external memory BFS as they not only have a large number of BFS levels, but also their layout on the disk makes the random accesses to adjacency lists very costly. Also, on moderate diameter grid graph, MM_BFS_D which takes 21 hours outperforms MM_BFS_R and MR_BFS. It is interesting to note that Christiani [37] reached a different conclusion regarding the relative performance of MM_BFS_D and MM_BFS_R. As noted before, this is because of the cache oblivious routines used in their implementation.

On large diameter sparse graphs such as line graphs, the randomized preprocessing scans the graph $\Omega(\sqrt{B})$ times, incurring an expected number of $O(\sqrt{n \cdot (n+m) \cdot \log(n)/B})$ I/Os. On the other hand, the I/O complexity of the deterministic preprocessing is $O((1 + \log\log(D \cdot B \cdot n/m)) \cdot \textsc{Sort}(n+m))$, dominated by the spanning tree computation. Note that the Euler tour computation followed by list ranking only requires $O(\textsc{Sort}(m))$ I/Os. This asymptotic difference shows in the I/O volume of the two preprocessing variants (Table 6.10), thereby explaining the better performance of the deterministic preprocessing over the randomized one (Table 6.11). On low diameter random graphs, the diameter of

the clusters is small and consequently, the randomized variant scans the graph fewer times leading to less I/O volume.

As compared to MM_BFS_R, MM_BFS_D provides dual advantages: First, the preprocessing itself is faster and second, for most graph classes, the partitioning is also more robust, thus leading to better worst-case runtimes in the BFS phase. The later is because the clusters generated by the deterministic preprocessing are of diameter at most $\sqrt{B}$, while the ones by randomized preprocessing can have a larger diameter causing adjacency lists to be scanned more often. Also, MM_BFS_D benefits much more from our caching heuristic than MM_BFS_R as the deterministic preprocessing gathers neighbouring clusters of the graph on contiguous locations in the disk.

6.2.3. Results with Heuristic.

Graph class	n	m	MM_BFS_D	
			Phase1	Phase2
Random	2^{28}	2^{30}	5.2	3.4
Webgraph	$\sim 1.4 \cdot 10^8$	$\sim 1.2 \cdot 10^9$	3.3	2.4
Grid $(2^{21} \times 2^7)$	2^{28}	$\sim 2^{29}$	3.6	0.4
Grid $(2^{27} \times 2)$	2^{28}	$\sim 2^{28} + 2^{27}$	3.2	0.6
Simple Line	2^{28}	$2^{28} - 1$	2.6	0.4
Random Line	2^{28}	$2^{28} - 1$	3.2	0.5

Table 6.12.: Time taken (in hours) by the two phases of MM_BFS_D with our heuristic

Table 6.12 shows the results of MM_BFS_D with our heuristic on different graph classes. On moderate diameter grid graphs as well as large diameter random line graphs, MM_BFS_D with our heuristic provides the fastest implementation of BFS in the external memory.

6.3. Conclusion

Table 6.13 gives the current state of the art implementations of external memory BFS on different graph classes.

Our improved MR_BFS implementation outperforms the other external memory BFS implementations on low diameter graphs or when the nodes of a graph are arranged on the disk in the order required for BFS traversal. For random graphs with 256 million nodes and a billion edges, our improved MR_BFS performs BFS in just 1.4 hours. Similarly, improved MR_BFS takes only 2.6 hours on webgraphs (whose runtime is dominated by the short diameter core) and 0.4 hours on line graph with contigous layout on disk. On moderate diameter square grid graphs, the total time for BFS is brought down from 54.3 hours for MM_BFS_R implementation in [5] to 21 hours for our implementation of MM_BFS_D with heuristics, an improvement of more than 60%. For large diameter graphs like random line graphs, MM_BFS_D along with our heuristic computes the BFS in just about 3.6

Graph class	n	m	Current best results	
			Total time	Implementation
Random	2^{28}	2^{30}	1.4	Improved MR_BFS
Webgraph	$\sim 1.4 \cdot 10^8$	$\sim 1.2 \cdot 10^9$	2.6	Improved MR_BFS
Grid ($2^{14} \times 2^{14}$)	2^{28}	2^{29}	21	MM_BFS_D w/ heuristic
Grid ($2^{21} \times 2^7$)	2^{28}	$\sim 2^{29}$	4.0	MM_BFS_D w/ heuristic
Grid ($2^{27} \times 2$)	2^{28}	$\sim 2^{28} + 2^{27}$	3.8	MM_BFS_D w/ heuristic
Simple Line	2^{28}	$2^{28} - 1$	0.4	Improved MR_BFS
Random Line	2^{28}	$2^{28} - 1$	3.6	MM_BFS_D w/ heuristic

Table 6.13.: The best total running time (in hours) for BFS traversal on different graphs with the best external memory BFS implementations

hours, which would have taken the MM_BFS_R implementation in [5] around 12 *days* and MR_BFS and IM_BFS a few *months*, an improvement by a factor of more than 75 and 1300, respectively.

6.3.1. Discussion

We implemented the deterministic variant of MM_BFS and showed its comparative analysis with other external memory BFS algorithms. Together with the improved implementations of MR_BFS and MM_BFS_R and our heuristic for maintaining the pool, it provides viable BFS traversal on different classes of massive sparse graphs. In particular, we obtain an improvement factor between 75 and 1300 for line graphs with random disk layout over the previous external memory implementations of BFS.

Acknowledgements

We are grateful to Rolf Fagerberg and Frederik Juul Christiani for providing us their code. Also thanks are due to Dominik Schultes and Roman Dementiev for their help in using the external MST implementation and STXXL, respectively. The authors also acknowledge the usage of the computing resources of the University of Karlsruhe.

CHAPTER 7

Single Source Shortest Paths on Massive Graphs

Let $G = (V, E)$ be a graph with $|V| = n$ vertices and $|E| = m$ edges, let s be a vertex of G, called the *source vertex*, and let c be an assignment of non-negative lengths to the edges of G. The *single-source shortest-path* (SSSP) problem is to find, for every vertex $v \in V$, the distance, $dist(s, v)$, from s to v, that is, the length of a shortest path from s to v in G.

The classical SSSP-algorithm for general graphs is Dijkstra's algorithm [44]. Unfortunately, it performs poorly on massive graphs that do not fit into the main memory and are stored on disk. The reason is that Dijkstra's algorithm accesses the data in an unstructured fashion.

Much recent work has focused on algorithms for massive graphs, see [110, 168] for surveys. These algorithms are analyzed in the external memory model [168], see Section 2.2, which assumes that the computer has a main memory that can hold M vertices or edges and that the graph is stored on disk.

In this chapter we report on initial experimental results for a practical I/O-efficient Single-Source Shortest-Paths (SSSP) algorithm on general undirected sparse graphs where the ratio between the largest and the smallest edge weight is reasonably bounded (for example integer weights in $\{1, \ldots, 2^{32}\}$) and the realistic assumption holds that main memory is big enough to keep one bit per vertex.

While our implementation only guarantees average-case efficiency, i.e., assuming randomly chosen edge-weights, it turns out that its performance on real-world instances with non-random edge weights is actually even better than on the respective inputs with random weights.

Furthermore, compared to the currently best implementation for external-memory BFS [6] described in Chapter 6, which in a sense constitutes a lower bound for SSSP, the running time of our approach always stayed within a factor of five, for the most difficult graph classes the difference was even less than a factor of two.

We are not aware of any previous I/O-efficient implementation for the classic general SSSP in a (semi) external setting: in two recent projects [34, 139], Kumar/Schwabe-like SSSP

approaches on graphs of at most 6 million vertices have been tested, forcing the authors to artificially restrict the main memory size, M, to rather unrealistic 4 to 16 MBytes in order not to leave the semi-external setting or produce huge running times for larger graphs: for random graphs of 2^{20} vertices, the best previous approach needed over six hours. In contrast, for a similar ratio of input size vs. M, but on a 128 times larger and even sparser random graph, our approach was less than seven times slower, a relative gain of nearly 20. On a real-world 24 million node street graph, our implementation was over 40 times faster. Even larger gains of over 500 can be estimated for random line graphs based on previous experimental results for Munagala/Ranade-BFS.

Finally, we also report on early results of experiments in which we replace the hard disk by a solid state disk (flash memory).

References. The contents of this chapter is based on the joint work with Ulrich Meyer [109]. Most of the wording of the original publication is preserved.

7.1. Overview

Little is known about solving SSSP on *directed* graphs I/O-efficiently. For *undirected* graphs, the algorithm of Kumar and Schwabe (KS_SSSP) [90] performs $O(n+(m/B)\log(n/B))$ I/Os. For dense graphs, the second term dominates; but for sparse graphs, the I/O-bound becomes $O(n)$.

The SSSP-algorithm of Meyer and Zeh (MZ_SSSP) [111] extends the ideas of [104] for breadth-first search (BFS) to graphs with edge lengths between 1 and K, leading to an $O(\sqrt{nm\log K/B} + \mathrm{MST}(n,m))$ bound, where $\mathrm{MST}(n,m)$ is the cost of computing a minimum spanning tree.[1] Recently [112], the result was further improved to $O(\sqrt{nm/B}\log n + \mathrm{MST}(n,m))$ I/Os, thus removing MZ_SSSP's dependence on the edge lengths in the graph. However, the latter approach is extremely involved and would probably suffer from very high constant factors in any realistic implementation setting.

When it comes to recent *internal-memory* SSSP implementations, the 9th DIMACS implementation challenge [124] provides a good overview. As for external-memory SSSP algorithms, to the best of our knowledge, none of the $o(n)$-I/O SSSP algorithms has ever been tried out. However, there are two recent papers [34, 139] reporting on external-memory experiments for KS_SSSP like approaches. Unfortunately, all results are for graphs of at most 6 million vertices, forcing the authors to artificially restrict the usably main memory size to rather unrealistic 4 to 16 MB in order not to leave the (semi-)external setting. Even then, computing SSSP for a random graph with $n \simeq 10^6$ vertices in the best case takes over 6 hours [139], which is more time than needed to do n I/Os.

Furthermore, using the I/O-library STXXL [41], Ajwani et al. [5, 6], see also Chapter 6, studied implementations of external-memory BFS, i.e., the unweighted version of SSSP. They managed to compute BFS on different kinds of undirected graphs featuring over 250 million nodes and more than a billion of edges in less than 24 hours, see also Chapter 6 for details.

[1] The current bounds for $\mathrm{MST}(n,m)$ are $O(\mathrm{sort}(m)\log\log(nB/m))$ [10] deterministically and $O(\mathrm{sort}(m))$ randomized [36].

Another line of related research is algorithms for point-to-point shortest-path queries in (semi-)external memory using compression and extensive pre-computation in internal memory. Typical representatives are, e.g., [18, 59, 142]. The success of these approaches crucially depends on the special characteristics of the input graphs (in particular road networks). In contrast we are interested in I/O-efficient general purpose SSSP computation without any structural assumptions on the input graph.

Our Contribution. We provide initial experimental results for a practical I/O-efficient SSSP algorithm on undirected graphs where the ratio between the largest and the smallest edge weight is reasonably bounded (for example integer weights in $\{1, \ldots, 2^{32}\}$). Compared to the improved external-memory BFS implementation from Chapter 6 our new approach was never slower than a factor of five, while for the most difficult graph classes the difference was even less than a factor of two. The result is obtained by simplifying MZ_SSSP in two ways: (1) using the realistic assumption that the main memory is big enough to keep one bit per vertex (i.e., the weakest form of the semi-external memory setting), thus facilitating to apply a standard external-memory priority queue without support for decrease_key; (2) omitting a complicated weight-based clustering and using an already existing routine from the Chapter 6 instead. While this simplification maintains the $O(\sqrt{nm \log K/B} + \mathrm{MST}(n,m))$ I/O-bound of MZ_SSSP for uniformly distributed random edge-weights in $\{1, \ldots, K\}$ it could result in much more I/O for non-random edge weights: $O(\sqrt{nmK/B} + \mathrm{MST}(n,m))$.

However, the performance of our STXXL [41] based implementation revealed just the opposite behavior: executed on real-world graphs with original non-random weights it was actually faster than on the same graphs with artificially assigned random weights.

While previous implementation studies [34, 139] for (semi-)external Kumar/Schwabe [90] kind SSSP approaches dealt with graphs having at most six million vertices, our study covers graphs of up to 250 million vertices and a billion edges. For random graphs of $n = 2^{20}$ vertices and $m = 8 \cdot n$ edges, the best previous approach needed over six hours. In contrast, for a similar ratio of $(n+m)/M$, but on larger and sparser random graphs of $n = 2^{28}$ vertices and $m = 4 \cdot n$ edges, our approach was less than seven times slower, a relative gain of nearly 20. On a real-world 24 million node street graph, our implementation was over 40 times faster. Even larger gains of over 500 can be estimated for random line graphs based on previous experimental results, see Chapter 6, for Munagala/Ranade-BFS [116].

7.2. Algorithm Design

Overview. Our SSSP approach is an I/O-efficient version of Dijkstra's algorithm [44]. Dijkstra uses a priority queue Q to store all vertices of G that have not been settled yet (a vertex is said to be *settled* when its final distance from s has been determined); the priority of a vertex v in Q is the length of the currently shortest known path from s to v. Vertices are settled one-by-one by increasing distance from s. The next vertex v to be settled is retrieved from Q using a delete_min operation. Then the algorithm relaxes the edges between v and all its non-settled neighbors, that is, performs a decrease_key$(w, dist(s,v) + c(v,w))$ operation for each such neighbor w whose priority is greater than $dist(s,v) + c(v,w)$.

An I/O-efficient version of Dijkstra's algorithm has to (a) avoid accessing adjacency lists at random, (b) deal with the lack of optimal decrease_key operations in current external-memory priority queues, and (c) efficiently remember settled vertices. Since we allow ourselves one bit per node in internal memory problems (b) and (c) are easily solved. As for (c) the bit vector is used to keep track which vertices have been visited. Concerning (b) we allow up to degree(v) many entries for a vertex v in the priority-queue at the same time and when extracting them discard all but the first one with the help of the bit vector. As for (a) our approach forms clusters of vertices just like the EM-BFS algorithm of Mehlhorn and Meyer[104] (i.e., without considering the edge weights at all, see Chapter 6 for details) and loads the adjacency lists of all vertices in a cluster into a number of "hot pools" of edges as soon as the first vertex in the cluster is settled. For integer edge weights from $\{1, \ldots, K\}$ we have $k = \lceil \log_2 K \rceil$ pools, where the i-th pool is reserved for category i edges, that is, edges of weight between 2^{i-1} and $2^i - 1$.

In order to relax the edges incident to settled vertices, the hot pools are scanned and all relevant edges are relaxed. However, we use that the relaxation of edges of large weight can be delayed because if such an edge is on a shortest path, it takes some time before its other endpoint is settled. Hence, it is sufficient to touch hot pools for higher categories much less frequently than the pools containing short edges. Unfortunately, due to the simplified clustering, in a worst-case setting the majority of edges might have small weights and still belong to clusters of large diameter, thus resulting in huge scanning costs for the lower category pools of our approach: $O(\sqrt{nmK/B})$ I/Os. Still, for random edges weights uniformly distributed in $\{1, \ldots, K\}$ the total number of expected I/Os remains $O(\sqrt{nm \log K/B} + MST(n, m))$, just like for the much more complicated MZ_SSSP algorithm.

In the following we will provide some more details on the implementation.

Graph Data Structure. Boost libraries [158] are considered as the next level of standardization over Standard Template Library (STL for short). Unfortunately, even though the Boost Graph Library (BGL for short) includes several graph classes, such as adjacency list or adjacency matrix, missing guaranties on the layout of edges on the hard drive make them inapplicable for I/O efficient algorithms. Therefore, we have implemented our own I/O-efficient graph representation that conforms to the BGL interface, thus providing the same level of generality.

On a low level our graph class can be parameterized by a vector container compatible with the STL vector interface, that stores graph edges along with the additional information defined by the user. In our particular case such a container is an STXXL vector, since it guaranties that the scanning of edges is performed in $O(m/B)$ I/Os.

Priority Queue. We store nodes with their tentative distances in the I/O efficient priority queue being part of the STXXL library. Each of its operations takes $O(1/B \log_{M/B} I/B)$ I/O amortized, where I denotes the total number of insertions [141]. Note that we may keep several entries with different priorities for some vertices at the same time.

Pipelining. Our implementation intensively uses pipelining. Conceptually, *pipelining* is a partitioning of the algorithm into practically independent parts that conform to a common interface, so that the data can be streamed from one part to the other without any intermediate external-memory storage. This way the I/O complexity may be reduced by up

to a constant factor. Moreover, it also leads to a better structured implementation, while different parts of the pipeline only share a narrow common interface. On the other hand, the price one sometimes has to pay is higher computational costs and potentially somewhat larger debugging efforts. For more details on pipelining in the framework of I/O efficient algorithms, see [41].

Deterministic graph clustering. In the deterministic preprocessing we compute a spanning tree for the connected component containing the source node, obtain an Euler tour around that spanning tree, and eventually form the clusters based on subsequences of the Euler tour (generated by list-ranking and sorting). We apply the external-memory deterministic preprocessing implementation from Chapter 6, which in turn uses a spanning forest and connected components implementation by Dementiev et al. [43] with expected $sort(m)\lceil \log n/M \rceil$ I/O runtime [43]. Furthermore, they use an adaptation of Sibeyn's list ranking algorithm [157]. Both implementations are based on STXXL data structures and its sorting primitive. For more details on the deterministic preprocessing, refer to Chapter 6.

SSSP phase. Figure 7.1 shows the flow-chart of the pipelined loop of the SSSP phase. In the beginning of each iteration (point 1) we settle a vertex v that has the smallest tentative distance *dist* in the priority queue, and mark it visited in the internal memory bit array *done*. Along with the node index v each priority queue element stores a bit array, such that its i-th bit is set to true if v has an incident category i edge. The bit array is constructed for each node in the preprocessing phase and requires $2 \cdot k$ bits additional space per edge in the graph data structure. Having extracted v's bit array, if the i-th bit is 1 then we put a pair $(v, dist(v))$ in the corresponding queue relax_i of nodes waiting for relaxation of their incident category-i edges.

Then we check (point 2) if there are any previously settled nodes in some relax_i, whose incident edges have to be relaxed before settling the next node at the top of the priority queue. Thus, we check the *delayed relaxation condition* in Figure 7.1 for the oldest node within each relax_i queue. Observe, that this is sufficient, since the distances associated with the elements of any of relax_i starting from its oldest element do not decrease.

If the condition is satisfied for some category i, then the nodes of the corresponding relax_i queue are sorted by their node index (point 3) and their adjacent category i edges are either loaded from the corresponding HotPool_i (point 4) and relaxed or have to be loaded from the external graph and therefore are passed further through the pipeline (point 5).

In order not to access the clusters of the external graph more than once, all nodes v are accompanied with and sorted by their cluster indices c. After that we identify and load the required external clusters containing currently missing adjacency lists (point 6) and "relax" them by inserting a potentially non-improving value into the priority queue (recall that we emulate a decrease_key operation via a bit vector plus discarding). All other edges of the just loaded clusters are sorted and distributed over HotPools corresponding to their categories (point 7). The loop terminates when the priority queue becomes empty.

A heuristic for maintaining the pool. The asymptotic improvement and performance gain in MZ_SSSP as compared to KS_SSSP is due to the partitioning of the input graph into the clusters and maintaining an efficiently accessible graph cache (hot pools) of adjacency

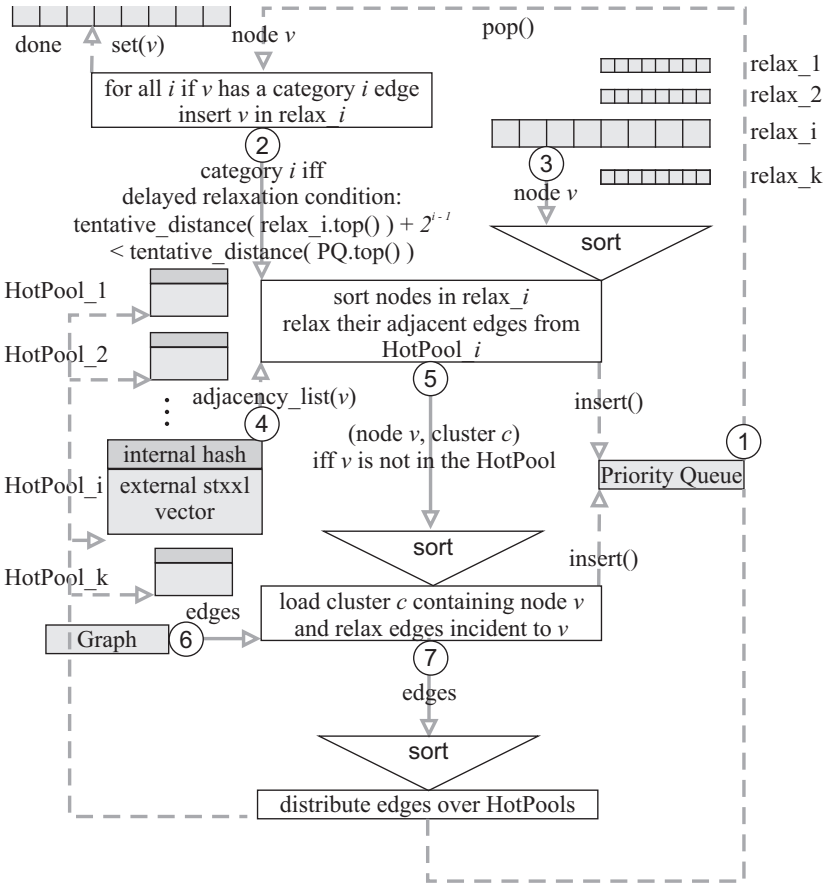


Figure 7.1.: Flow-chart of the pipelined SSSP phase implementation. The empty elements of the pipeline conform to the common pipelining interface, while the solid lines denote the data stream through the pipeline. Shaded elements represent non-pipelined data structures with the dashed lines denoting the data exchange through their auxiliary methods. The numbered circles reflect the order in which the elements flow through the pipeline.

lists, which are guaranteed to be requested soon after. Thus, efficient access patterns to the hot pools are crucial for the performance of MZ_SSSP.

In Chapter 6 we observed, that in the case of BFS for many large diameter graphs, the pool fits into the internal memory most of the time. We proposed maintaining it partially in an internal memory hash table, thus using efficient dictionary look up instead of computationally quite expensive scanning of all hot pool edges. Besides that, we observed that when the clusters are small enough ($O(\sqrt{B})$ for line graphs), it is worth caching all neighboring clusters that are anyway loaded into the main memory while reading B elements from the disk. The last fact is due to the special layout of clusters the deterministic preprocessing produces. As for implementation, we store adjacency lists in the STXXL vector, thus, loading the neighboring clusters in its internal memory cache using an LRU replacement strategy, see Figure 6.1. This heuristic approach appeared to be particularly efficient for medium and large diameter grid and line graphs.

Since the concept of the SSSP graph cache in many aspects resembles the BFS hot pool, we extended this approach and included it in our SSSP phase implementation.

While in Chapter 6 we had only *one* hot pool, now we have k hot pools for k different categories of edges. As well as in the BFS case, we use a multi-map hash table to maintain $O(M)$ edges internally. Observe, that due to the relaxation condition, Figure 7.1, hot pools with the low category edges are likely to be requested more often than those of higher categories. Thus, it is worthwhile reserving more internal memory for the smaller category hot pools. For the comparative study of different memory allocation strategies refer to Subsection 7.2.1. As for the caching of neighboring clusters, we use the same technique as in Chapter 6 to benefit from the special cluster disk layout produced by the deterministic preprocessing, see Figure 6.1.

7.2.1. Experimental Study

Configuration. We implemented our algorithm using the C++ programming language and the GNU compiler 4.2.1 (optimization level -O3) on an Open Suse Linux 10.3 distribution and the external-memory STXXL library version 1.1.0.

Our experimental **platform G**, see Section 3.2 has 250 GB Seagate Baracuda hard disks. The hard drive buffer cache is 8 MB big. The average seek time for read and write is 8.0 and 9.0 msec respectively. The data transfer rate for outer zone (maximum) is 65 MByte/s. Therefore, for a graph with 2^{28} nodes n random read and write I/Os would take around 600 and 675 hours, respectively.

In order to use equivalent hardware to the one for the BFS implementation in Chapter 6, we restrict the available memory to at most 1 GB and only use one processor and one disk.

Real world road network graphs. We did the experiments for the largest road network graphs that we could access, that is, the European[2] and the US graphs. The former one features around 33 million nodes and 40 million edges, while the later has 24 million nodes and 29 million edges.

While being one of the most popular applications, SSSP on road networks is not necessarily the best illustration for our algorithm due to the following reasons: (1) even the European road network is rather small for realistic external-memory settings; (2) the special structure of road networks allows recent specialized approaches to outperform the general purpose Dijkstra algorithm by several orders of magnitude, e.g., see [18, 59]. Although no theoretical I/O bounds are given, the algorithm in [59] has been designed with the explicit goal of being efficient on devices with small internal memory and slow storage memories (e.g., flash memories) such as pocket PCs. Similarly, in recent work Sanders et al. [142] propose a highly efficient algorithm for point-to-point shortest path queries on mobile devices.

In order to bring the problem closer to our settings we (1) reduced the memory size available for our algorithm to 128 MB and (2) randomly permuted the node indices.

Web graph. As an instance of real world graphs we also consider a crawl of the world wide web [165]. The nodes of the web graph represent internet pages, while the edges correspond

[2]provided for scientific use by Ortec company.

Graph class	n	m	BFS	SSSP
Random (16 bit)	2^{28}	2^{30}	8.6	36
Random (32 bit)	2^{28}	2^{30}	8.6	39.2
Grid ($2^{14} \times 2^{14}$, 16 bit)	2^{28}	2^{29}	21	33.6
Grid ($2^{14} \times 2^{14}$, 32 bit)	2^{28}	2^{29}	21	37.6
Random line (16 bit)	2^{28}	2^{28}	3.7	7.6
Webgraph (32 bit)	$\approx 135 \times 10^6$	$\approx 1.18 \times 10^9$	5.7	28.7

Table 7.1.: Timing in hours for the currently best BFS implementation vs. our SSSP approach (both including preprocessing).

to the links between them. Our instance of the web graph has around 135 million nodes and 1.2 billion edges. Structurally the web graph is close to a random graph, with a small fraction of larger diameter branches. Therefore, the I/O runtime is similar to the one for random graphs.

Synthetic graph classes. In order to isolate the performance penalty for computing SSSP as opposed to BFS, we consider the same graph classes as in Chapter 6:
Random graphs: A random graph with n nodes and about m edges is obtained by selecting m times a random source and a random target with source $\neq$ target and subsequently remove the duplicate edges.
Grid graph ($x \times y$): They consist of a xy grid, with edges joining the neighboring nodes in the grid.
Line graphs: They have n nodes and $n-1$ edges, such that there exist two nodes with the path between them containing all other nodes. A simple line graph is laid out on the disk, such that each disk block B contains consecutively lined nodes whereas for a random line graph the arrangement of nodes is given by a random permutation.

Comparing BFS and SSSP. We compared our SSSP implementation against BFS implementation from Chapter 6. The result in Table 7.1 indicates, that while Ajwani et al. perform BFS traversal for any of the graph classes within *one* day, we compute SSSP for the same graph classes with 16 and 32 bit random weights within just *two* days. Our SSSP approach was never slower than a factor of five, while for the most difficult graph class (grids) the difference was even less than a factor of two.

Comparing KS_SSSP and MZ_SSSP. If we try to relate different SSSP algorithms with their BFS counterparts, then KS_SSSP and the external-memory BFS algorithm by Munagala and Ranade (MR_BFS for short) [116] share similar ideas (and access patterns), whereas MZ_SSSP corresponds to Mehlhorn and Meyer's BFS algorithm (MM_BFS for short).

In Chapter 6 we showed that MR_BFS outperforms MM_BFS for low diameter graphs, such as random or web graphs, while medium and large diameter graph instances become practically infeasible for it (hours as opposed to months for line graphs). As for KS_SSSP and MZ_SSSP, we expect the later one to significantly outperform the former for the *whole* range of graphs that we consider. The reason for it is due to the incremental nature of Dijkstra's algorithm. Indeed, while MR_BFS extracts adjacency lists in a batched fashion level by

Node indices	$n/10^6$	$m/10^6$	RAM	SSSP by [139]		SSSP phase		Preprocessing	
				I/O w.	Total	I/O w.	Total	I/O w.	Total
original	24	29	1024	4550	4964	155	1414	420	547
original	24	29	512	4848	5222	191	1449	484	614
original	24	29	128	5059	5444	2815	4059	956	1123
permuted	24	29	2048	209350	209873	136	1417	428	589
permuted	24	29	1024	*	*	175	1458	455	611
permuted	24	29	512	*	*	187	1474	529	685
permuted	24	29	128	*	*	2892	4158	951	1139

Table 7.2.: Timing in *seconds* for US road network with *original* or *permuted* node indices and original edge weights using RAM in *megabytes*. Fields marked with * are omited due to high computation cost.

Road Network	$n/10^6$	$m/10^6$	SSSP phase	
			I/O wait	Total
original × original	34	39	4269	6011
permuted × original	34	39	4635	6392
permuted × 32-bit	34	39	7802	10819

Table 7.3.: Timing in *seconds* for European road network with *original* or *permuted* node indices and *original* or *32-bit* random edge weights using 128 MB of RAM.

level, KS_SSSP loads edges incident to the settled vertices consecutively vertex by vertex. Therefore, for the expected $O(\log n)$ levels of a random graph MR_BFS spends on average $O(n/B \log n)$ I/Os, while KS_SSSP requires one I/O per vertex, thus exhibiting worst case $\Omega(n)$ I/O performance in practice.

This observation is in line with the recent implementation of a Kumar/Schwabe-like approach by Sach and Clifford [139], who used a cache oblivious priority queue and an internal-memory bit array like us in our approach. They observed in practice that for random graphs the I/O complexity for extracting adjacency lists was a dominating factor over maintaining the priority queue.

Moreover, as it is shown in Table 7.2 the performance of their algorithm on the real world road networks also significantly depends on the layout of edges. As we already mentioned above, available real world road network instances initially incorporate spatial locality, thus facilitating more efficient adjacency lists extraction. Therefore, for original vertex numbering the runtime of their implementation only slightly depends on the internal memory available for the system. However, a random permutation of vertices has a significant impact on the performance, thus showing overwhelming dependence of the runtime on the layout of adjacency lists on a disk. On contrary, the runtime of our SSSP algorithm in Table 7.2 barely depends on the original vertex indices, which is a desirable feature for a general purpose SSSP solver.

Graph class	n	m	MR_BFS [116]	SSSP
Random line	2^{28}	2^{30}	4760	7.6

Table 7.4.: Timing in hours for MR_BFS and SSSP (including preprocessing, for *16-bit* random edge weights).

As for large diameter graphs, in Chapter 6 we showed that MM_BFS drastically outperforms MR_BFS for random line graphs. Since the I/O performance for MR_BFS constitutes a lower bound for KS_SSSP, we directly compare the MR_BFS results from Chapter 6 with our SSSP approach in order to estimate an advantage of more than a factor of 500, see Table 7.4.

To summarize, the MZ_SSSP preprocessing step allows the subsequent SSSP phase to significantly outperform any Kumar/Schwabe like approach that ignores I/O complexity for extracting adjacency lists.

Next, we show that the delayed relaxation condition further improves MZ_SSSP's performance by allowing a batched relaxation of edges of higher categories.

Delayed relaxation of edges. As we already discussed in Section 7.2, the delayed relaxation condition in Figure 7.1 allows postponing relaxation of longer edges. For each edge category we measured the number of batched relaxations of nodes having incident edges in this category, and compared it to the number of relaxations that would have been performed without the relaxation condition in use. In the series of diagrams Figure 7.2 and Figure 7.3 we compare the number of batched relaxations (in black) with the their overall number (in gray) in logarithmic scale.

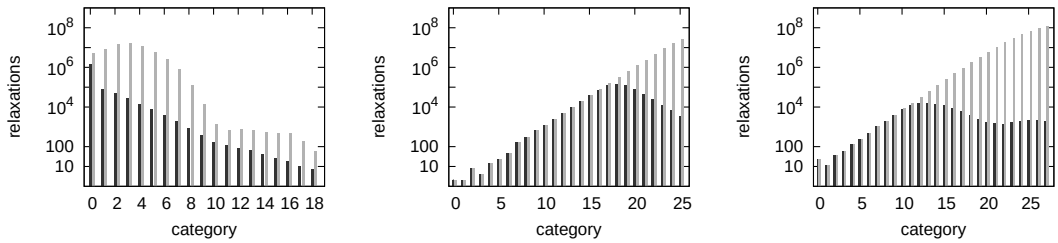


Figure 7.2.: The comparison between the number of batched relaxations (black) with the total number of relaxations (gray) in logarithmic scale. From left to right: European road network graph with original edges, with random 32 bit weights and web graph with 32 bit random weights

Note, that in case of the European road network with real distances, delayed relaxation is even more beneficial than for 32 bit random weights on the same graph (compare first and second histograms in Figure 7.2). Even in the first, most notable category, the number of batched relaxations is around 20% of the overall number. Thus, on average the algorithm relaxes a batch of first category edges incident to five different nodes at once. In the next categories the ratio drops to at most 1%, that is, on average at least 100 nodes at once. The

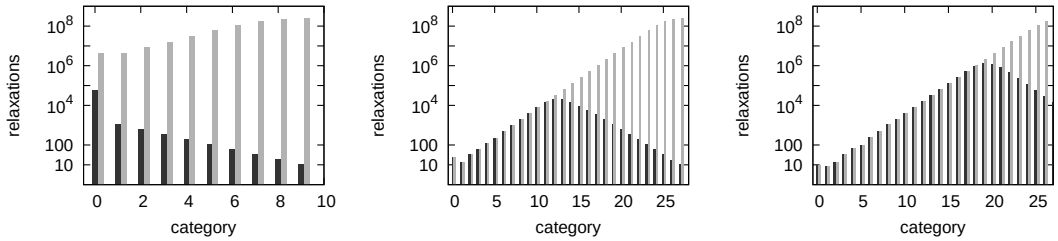


Figure 7.3.: The comparison between the number of batched relaxations (black) with the total number of relaxations (gray) in logarithmic scale. From left to right: random graph with 16 bit random weights, random graph with 32 bit random weights and random grid graph with 32 bit random weights

higher ratio for the last few categories is only due to the low number of long distances in the road network.

On the contrary, the same road network graph with 32 bit random weights (second histogram in Figure 7.2) demonstrates a small ratio only for the upper half of the categories, while in the first categories the relaxations are performed in a one-by-one manner. This in turn leads to a significant performance loss, refer to Table 7.3 for the exact timing.

Besides the European road network, we computed ratios for the web graph and synthetic instances, that have 16 and 32 bit random weights.

The first category of the random graph with 16 bit random weights has the largest number of batched relaxations with the ratio of about 1.3%, that further decreases to up to $4 \cdot 10^{-6}$% in the last category (first histogram in Figure 7.3).

For the web graph (last histogram in Figure 7.2) and the random graph with 32 bit random weights (second column in Figure 7.3) we see similar behavior: one-by-one relaxations in the lower categories, and rapidly decreasing ratio in the higher categories.

As a rule of thumb for graphs with random edge weights, the bigger the diameter, the larger the number of categories, where relaxations have to be performed in a one-by-one fashion. For instance, for the random graph with 32 bit edge weights the ratio drops significantly already for the categories $14 - 15$, while for the grid this value can be found around the $18 - 19$th category (compare the second and the third histograms in Figure 7.3). The most extreme case is the line graph, where essentially all relaxations have to be performed consecutively one after the other.

Quality of the spanning tree. In Chapter 6 we observed that the shape of the spanning tree in the preprocessing step plays an important role for the quality of the clustering. In line with the Chapter 6 for the grid graph we observed a considerable improvement in the SSSP phase I/O runtime when the spanning tree is "randomized". The reason for it is, that a spanning tree with elements in a snake-like row major order produces long and narrow clusters, while a "random" one is more likely to result in low diameter clusters. The former clusters tend to stay in the hot pools longer, hence, increasing their sizes, that eventually results in a larger I/O volume for storing hot pools and for re-scanning them while retrieving adjacency lists. On the other hand, the latter ones are evicted from the hot pools sooner,

thus reducing I/O runtime. Most notably, for a simple grid graph with random 16 bit weights, clustering with the randomized input of the spanning tree algorithm gives about 30% runtime improvement over the unrandomized one, Table 7.5.

Graph class	n	m	Preprocessing		SSSP phase	
			I/O wait	Total	I/O wait	Total
Simple Grid ($2^{14} \times 2^{14}$, 16 bit)	2^{28}	2^{29}	2.5	3	30	44.4
Simple Grid ($2^{14} \times 2^{14}$, 32 bit)	2^{28}	2^{29}	2.5	3	27.8	35.3
Random Grid ($2^{14} \times 2^{14}$, 16 bit)	2^{28}	2^{29}	2.4	3.2	23.7	30.4
Random Grid ($2^{14} \times 2^{14}$, 32 bit)	2^{28}	2^{29}	2.4	3.2	26.4	34.4

Table 7.5.: Quality of the spanning tree.

Different memory allocation strategies for the heuristic. The delayed relaxation

Graph class	n	m	no cache		decreasing		uniform	
			I/O wait	Total	I/O wait	Total	I/O wait	Total
Random (16 bit)	2^{28}	2^{30}	22.34	34	19.9	30.83	-	-
Random Grid (32 bit $2^{14} \times 2^{14}$)	2^{28}	2^{29}	26.6	34.56	26.4	34.4	26	31.3
Simple Line (16 bit)	2^{28}	2^{28}	-	-	0.1	6.6	0.1	4.1

Table 7.6.: Different memory allocation strategies for the heuristic.

condition for random edge weights, Figure 7.1, implies that edges in the lower categories are relaxed more often than those in the higher categories. This suggests, that in general the hot pools storing the lower category edges should get more internal memory than those storing the higher category edges.

In most of our experiments, Table 7.1 in particular, we used common 128 MB of RAM for all hot pool caches and 64 MB for the adjacency list vector cache, see Figure 6.1. By default the overall 128 MB of RAM are split among the hot pools such that the category i hot pool receives only half of the memory that is available for the category $i-1$ hot pool. We call this strategy "decreasing".

As it is indicated in Figure 7.2 and Figure 7.3, the number of categories, where relaxations have to be performed consecutively in a one by one fashion, increase with growing diameter. Therefore, for the middle range diameter grid graph and large diameter line graph it is worth distributing available memory equally throughout all hot pools. This strategy is denoted as "uniform".

For the random graph with 16 bit weights "decreasing" shows better results, since the bulk of batched relaxations is performed in the low level category hot pools, see Figure 7.3. As for the larger diameter grid and line graphs "uniform" appears to be the best choice. The reason for it is that due to the regular structure and small degree of these graph classes there

are not that many (only one in case of line graph) paths between any two nodes, meaning that the algorithm needs to load edges quite often even from high category category hot pools. This is in line with observation in Section 7.2 (Figure 7.3), that grid (and especially line) graphs only nodes having high category edges are relaxed in a batched manner, while in the low categories nodes need to be relaxed basically one by one.

7.2.2. Early Results on Flash Memory.

Graph class	n	m	SSSP phase on HDD		SSSP phase on SSD	
			I/O wait	Total	I/O wait	Total
Random (32 bit)	2^{28}	2^{29}	14.9	18	11.4	14.5
Random Grid ($2^{14} \times 2^{14}$, 32 bit)	2^{28}	2^{29}	18.6	22.1	11.4	15
Random Line (32 bit)	2^{28}	2^{28}	1.5	2.2	3.9	4.6

Table 7.7.: Preliminary results on flash memory.

We also performed preliminary tests of our SSSP implementation on modern flash memory also known as solid state disks (SSDs). These are non-volatile, reprogrammable memories, which have recently become a commonplace in storage device technology. Flash memory devices are lighter, more shock resistant and consume less power. Moreover, since random read accesses are faster on solid state disks compared to traditional mechanical hard-disks, flash memory is fast becoming the dominant form of end-user storage in mobile computing.

Flash memory devices typically consist of an array of memory cells that are grouped into *pages* of consecutive cells, where a fixed amount of consecutive pages form a *block*. Reading is performed pagewise whereas writing typically requires erasing a whole block. Thus, the latency for reading a byte is usually much smaller than for writing it. Finally, each block can sustain only a limited number of erasures. To prevent blocks from wearing prematurely, flash devices usually have an built-in micro-controller that dynamically maps the logical block addresses to physical addresses so as to even out the erase operations sustained by the blocks.

Previous and related work on flash. Most previous algorithmic work on flash memories deals with wear leveling, block-mapping and flash-targeted file systems (see [56] for a comprehensive survey). There are not many algorithms designed to exploit the characteristics of flash memories. Ajwani et. al [8] proposed a model reflecting similarities and differences to the external memory model. Wu et al. [173, 174] proposed flash-aware implementations of B-trees and R-trees without file system support by explicitly handling block-mapping within the application data structures.

Other works include the use of flash memories for model checking [16] or for route planning (point-to-point shortest paths) on mobile devices [59, 142].

An adaptation of our previously mentioned EM-BFS implementation (see Chapter 6) for flash memory was discussed in [7]. In there, a 32 GB Hama SSD (2.5" IDE) was used. Due to limited bandwidth of this device (less than 30 MB/s) only a combination of flash plus

a traditional hard disk (in that case a 500 GB SEAGATE Barracuda 7200.11) was more profitable than using the hard disk alone: the small read-only graph clusters reside on flash from where they can be retrieved using fast random reads, whereas the hot pool with its frequent sequential rewriting of large data sequences stays on the hard disk in order to profit from higher throughput.

Results. We performed experiments on a newer machine featuring an Intel Quad Core Q6600 CPU, 8GB of RAM, a fast hard drive and a solid state disk. We used the gcc compiler 4.3.2 with optimization level O3 on a Debian Linux distribution and STXXL version 1.2.2. The available internal memory was restricted to at most 1 GB and only one processor was used. As for 2009, we observed that the performance of solid state disks has significantly improved over the last year: priced similarly as the 32 GB device purchased for [7] a year ago, our current 64 GB Hama SSD (3.5" SATA) was not only offering double the capacity but also featured significantly increased throughput of measured 84 MB/s (reading), and 75 MB/s (writing). Although our 500 GB SEAGATE Barracuda 7200.11 hard disk applied in these experiments still offered higher throughput (about 100 MB/s for sequential access of large blocks), now for medium diameter grid and large diameter random graphs even using a SSD alone resulted in faster SSSP execution, see Table 7.7. On the other hand, for random line graphs, the SSSP phase using the hard drive benefits from a larger throughput and sequential reading speed. Indeed, as observed in Chapter 6, for line graphs, the Euler-tour based preprocessing lays out the clusters on external memory storage in a way, that the clusters that are visited soon after each other during a BFS traversal are located sequentially, thus facilitating sequential reading. While the node visiting order for BFS and SSSP traversals may differ significantly in general, it is very similar for line graphs.

In order to be able to accommodate larger data sets on flash, we actually used two 64 GB SSD devices. For fair comparison with a single hard disk the two SSDs were concatenated into one raid (thus to preventing parallel I/Os). Of course, even better results can be obtained by striping data blocks over the SSDs, thus significantly increasing the throughput. Note that we did not yet tune our SSSP code towards the special metrics of flash memory: The cluster size in the SSSP algorithm was chosen in a way so as to balance the random reads and sequential I/Os on the hard disks, but now in this new setting, we can reduce the cluster size as the random I/Os are being done much faster by the flash memory. Our experiments suggest that this leads to even further improvements in the runtime of the SSSP algorithm.

7.3. Conclusions

We have provided a practical implementation for undirected SSSP in external-memory under the assumptions that at least one bit can be kept for each vertex and that the edge weights are reasonably bounded. It remains a challenging open problem to come up with a practically feasible solution for sparse directed graphs, even without theoretical guarantees.

Acknowledgements. We would like to thank Deepak Ajwani and Andreas Beckmann for helpful discussions and assistance with the flash disks.

CHAPTER 8

Minimum Spanning Tree

A minimum spanning tree (MST) of a graph $G = (V, E)$ is a minimum total weight subset of E that forms a spanning tree of G. The MST problem has been intensively studied in the past since it is a fundamental network design problem with many applications and because it allows for elegant and multifaceted polynomial-time algorithms. We present *Filter-Kruskal* – a simple modification of Kruskal's algorithm that avoids sorting edges that are "obviously" not in the MST. For arbitrary graphs with *random edge weights Filter-Kruskal* runs in time $\mathcal{O}\big(m + n \log n \log \frac{m}{n}\big)$, i.e. in linear time for not too sparse graphs. Experiments indicate that the algorithm has very good practical performance over the entire range of edge densities. An equally simple parallelization seems to be the currently best practical algorithm on multicore machines.

References. The contents of this chapter is based on the joint work with Peter Sanders [127]. Most of the wording of the original publication is preserved.

8.1. Overview

In practice (on sequential machines and in internal memory), two simple algorithms dating back at least half a century still perform best in most cases [43, 84, 114].

The *Jarník–Prim* algorithm [71, 136] grows a tree starting from an arbitrary node. Implemented using efficient priority queues, its running time is $\mathcal{O}(m + n \log n)$. Even with simpler priority queues, it performs well for random edge weights[1] – time $\mathcal{O}\big(m + n \log n \log \frac{m}{n}\big)$ [123].

Kruskal's algorithm [88] grows a forest in time $\mathcal{O}((m + n) \log m)$ by scanning the edges in order of increasing weight and adding those that join two trees in the current forest. In practice, *Kruskal* outperfoms *Jarník–Prim* for sparse graphs. For denser graphs, *Kruskal*

[1] here and in the following we use the following model for random edge weights: the edge weights are all different, can otherwise have arbitrary values, and are randomly permuted.

suffers from the $\mathcal{O}(m \log m)$ time needed for sorting all the edges. Therefore it is a natural idea to avoid sorting heavy edges that cannot contribute to the MST. Kershenbaum and van Slyke [85, 114] do this by building a priority queue of edges in linear time. Then Kruskal's algorithm subsequently removes the lightest edge until $n-1$ tree edges have been found. For random graphs with random edge weights, the MST edges are expected to be among the $\mathcal{O}(n \log n)$ lightest edges. Hence, we get an average execution time of $\mathcal{O}\big(m + n \log^2 n\big)$. Unfortunately, the stopping idea fails already if the MST contains a single heavy edge. Note that this can even happen for random edge weights: Consider a "lollipop graph" consisting of a random graph and an additional path of length k attached to one of its nodes. The MST needs all the path edges, about half of which will belong to the heavier half of the edges for random edge weights. Brennan [72, 128] implements the stopping idea more cache efficiently by integrating Kruskal's algorithm with quicksort (*qKruskal*). Apply *Kruskal* to small inputs. Otherwise, as in quicksort, partition the edges into a light part and a heavy part. Recurse on the light part. If the MST is not complete yet, recurse on the heavy part.

A key idea for more robust improvements of *Kruskal* is *filtering* – early discarding of edges that connect nodes in the same component of the forest defined by the MST edges already found. In [85] filtering is applied to edges about to be sifted up in a heap based implementation of *Kruskal*. However, no analysis is given and heap based algorithms are unlikely to be efficient on modern machines due to the cache inefficiency. In this chapter we investigate the idea to apply filtering to *qKruskal* – before recursing on the heavy part, remove all heavy edges that are within a component of the current forest. In Section 8.3 we explain the algorithm *Filter-Kruskal* in more detail. The analysis shows that for arbitrary graphs with random edge weights, *Filter-Kruskal* runs in expected time $\mathcal{O}\big(m + n \log n \log \frac{m}{n}\big)$. Note that this is the same performance also achieved by *Jarník–Prim* using binary heaps [123]. The experiments reported in Subsection 8.3.5 confirm that *Filter-Kruskal* performs very well for both sparse and dense graphs. Moreover, *Filter-Kruskal* allows a more coarse-grained and hence more practical parallelization than *Jarník–Prim*.

Related Work

MSTs can even be found in linear (expected) time [73, 86]. This algorithm can filter out edges without any sorting using sophisticated data structures that can check whether an edge e is the heaviest edge on the cycle defined by the minimum spanning forest (MSF) of an edge sample and e. However, such algorithms are complicated and large constant factors are involved. To check whether general edge filters are useful at all, [84] invests $\Theta(n \log n)$ preprocessing to allow for a better constant factor in filtering. This algorithm only significantly outperforms *Jarník–Prim* for rather dense graphs with weights that force m decrease_key operations. Katajainen and Navalainen [83] refine the heap based algorithm from [85] by first performing a bucket sort of the edges. For uniformly distributed random edge weights, this yields (near) linear expected execution time. It should be noted that this result is quite different from ours: (1) bucket sorting does not work in the comparison based model, whereas our algorithms are comparison based. (2) the result in [83] only holds for "smooth" (i.e, close to uniform) distributions of independent random edge weights whereas our result holds for random permutations of arbitrary edge weights and in particular when

edge weights are independently drawn from an arbitrary probability distribution. (3) for uniformly distributed edge weights, even basic *Kruskal* runs in (near) linear time if we use bucket sort. Indeed, experiments in [83] demonstrate that filtering does not help if bucket sorting can be used.

8.2. Kruskal's Algorithm

Algorithm 7: Pseudocode for *Kruskal* and *Filter-Kruskal*. T is a set of MST edges already known and P is the partition of V induced by T. When used as a standalone method, the procedures are called with empty T and a trivial partition P. The result is output in T.

```
1  Kruskal(E, T : Sequence of Edge, P : UnionFind)
2  begin
3      Sort (E)                                  /* by increasing edge weight */
4      foreach {u, v} ∈ E do
5          if u, v are in different components of P then
6              add edge {u, v} to T
7              join the partitions of u and v in P
8      end
9  end
10
11 Filter-Kruskal(E, T : Sequence of Edge, P : UnionFind)
12 begin
13     if m ≤ kruskalThreshold(n, |E|, |T|) then
14         Kruskal(E,T,P)
15     else
16         pick a pivot p ∈ E
17         E≤:= ⟨e ∈ E : e ≤ p⟩
18         E>:= ⟨e ∈ E : e > p⟩
19         Filter-Kruskal(E≤, T, P)
20         E>:= Filter(E>, P)
21         Filter-Kruskal(E>, T, P)
22 end
23
24 Filter(E, T : Sequence of Edge, P : UnionFind)
25 begin
26     return ⟨{u, v} ∈ E : u, v are in different components of P⟩
27 end
```

Let $G = (V, E)$ denote an undirected, weighted graph with $|V| = \{1, \ldots, n\}$. Let $m = |E|$. Since we need Kruskal's algorithm as a subroutine, we outline it here for self-containedness.

Algorithm 7 gives pseudocode that should be self-explaining. When *Kruskal* skips an edge $\{u, v\}$ that falls within a single component of T, this is safe because $\{u, v\}$ closes a cycle in T and is at least as heavy as all edges in T. In this situation, the *cycle* property of MSTs tells us that $\{u, v\}$ is not needed for an MST. The most sophisticated aspect of the algorithm is the Union-Find data structure P maintaining a partition of the nodes into components defined by the MST edges T found so far. P supports an operation union joining two partitions and an operation find(v) returning the node number of the *representative* of the partition of the node v. Indeed, the implementation will exploit that partitions are represented using *parent* references defining trees rooted at the representatives and that the paths leading to the roots are very short in an amortized sense (union-by-rank and path compression). In particular, if $m \gg n$, most path lengths will be one.

8.3. Algorithm Design

Algorithm 7 gives pseudocode for *Filter-Kruskal*. Similar to [128], the basic approach is to use quicksort for sorting the edges and to move the edge scanning part of *Kruskal* into the quicksort code. Hence, the algorithm now calls *Kruskal* on small[2] inputs and it calls itself for the lighter part of the edges. The only new ingredient at this level of abstraction is that before recursing on the heavier edges $E_>$, they are filtered. Filtering removes those edges that fall within the same component of the current node partitioning. Note that these edges are heavier than all edges in T and close a cycle in T. Hence, the cycle property implies that the filtered edges are not needed for an MST. The advantage of filtering is that filtered edges need not be sorted.

8.3.1. Results for Random Edge Weights.

In this subsection we only state the lemmas and theorems, the corresponding proofs are available in the original paper [127].

We first show that we can essentially restrict the analysis to counting comparisons since this quantity is indicative of the total execution time:

Lemma 4. *Let C denote the number of (edge weight) comparisons performed by Filter-Kruskal. Then Filter-Kruskal performs $\leq n - 1$* union *operations, an expected number of $\leq 2m + C$* find *operations, and $\mathcal{O}(m + C)$ work outside* union *and* find *operations.*

Lemma 5 ([153]). *Consider n* union *operations and M* find *operations on a union-find data structure with n elements using path compression and union by rank. Then the total execution time[3] is $\mathcal{O}(M + n \log^* n)$ where $\log^* n$ denotes the iterated logarithm with $\log^* n = 0$ for $n \leq 1$ and $\log^* n = 1 + \log^* \log n$ otherwise.*

[2] As far as asymptotic performance is concerned, any choice of the function kruskalThreshold works as long as kruskalThreshold$(n, |E|, |T|) = \mathcal{O}(n)$.

[3] A more well known bound is $M\alpha(m, n)$ where α is the inverse Ackermann function. However, we need a bound that is linear in M.

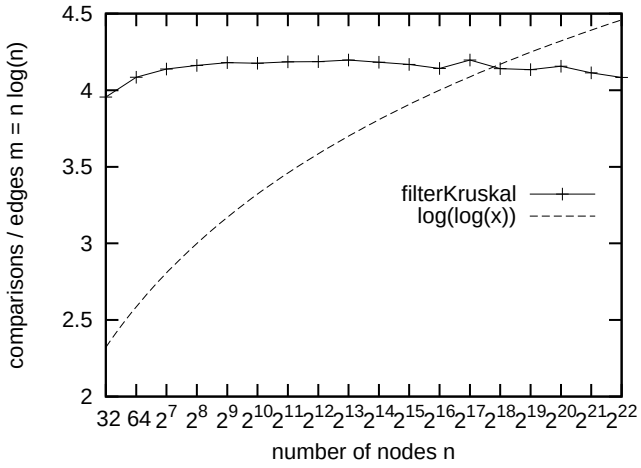


Figure 8.1.: Number of edge comparisons performed by algorithm *Filter-Kruskal* for random graphs with $n \log n$ edges.

Using these lemmas we analyze *Filter-Kruskal* for *random edge weights*. Without loss of generality, we can assume that the edge weights are the set $1..m$.

Theorem 6. *Given an arbitrary graph and random edge weights, the expected running time of* Filter-Kruskal *is* $\mathcal{O}\big(m + n \log(n) \log \frac{m}{n}\big)$.

We also give an informal argument why the complexity computed above is tight in the sense that using the sampling lemma from [33] we cannot expect a better result. Suppose, we had an algorithm that filters every edge with respect to all lighter edges "for free". Then, $\sum_{n<i\leq m} n/i = \Theta\big(n \log \frac{m}{n}\big)$ edges would survive this filtering (in expectation). Sorting those edges also yields the bound from Theorem 6.

Note that the term $n \log(n) \log \frac{m}{n}$ in the execution time of *Filter-Kruskal* can be simplified to $\mathcal{O}(n \log(n) \log \log n)$, i.e., for $m = \Omega\left(n \log(n) \log \log(n)\right)$ we get linear execution time. This is up to a factor $\log n / \log \log n$ better than *qKruskal* for random graphs with random edge weights and recall that our result applies to *arbitrary* graphs with random edge weights.

Indeed, it seems that for random graphs with random edge weights we get even a better bound. Figure 8.1 indicates that the number of edge comparisons executed by *Filter-Kruskal* for graphs with $n \log n$ edges[4] is proportional to $n \log n$ (at least the double-logarithmic upper bound from Theorem 6 is too pessimistic). This is quite strong evidence that the expected running time of *Filter-Kruskal* for random graphs with random edge weights is $\mathcal{O}(m + n \log(n))$: First observe that by Lemma 4 and Lemma 5, the comparisons are representative of the asymptotic execution time. Second, for instances with *less* than $n \log n$ edges, the running time cannot be larger. Finally, random graph theory tells us that the $n \log n$ lightest edges will define the MST with high probability[5]. Hence, all the heavier

[4]Throughout this chapter $\log x$ stands for $\log_2 x$.
[5]The threshold for connectivity is at $n \ln(n)/2$ edges.

edges will be filtered out anyway. We believe that a formal proof would not be too difficult by looking even closer at the structure of random graphs. In particular, since the number of nodes outside the giant component shrinks geometrically with the average degree, the probability that an edge survives filtering will also shrink geometrically with its rank divided by n. We believe that the same bound also applies to many other classes of graphs. Indeed we do not know a family of graphs for which *Filter-Kruskal* with random edge weights would take more than $\mathcal{O}(m + n \log(n))$ expected time.

8.3.2. Implementation.

For $m \gg n \log(n) \log \log n$, most of the work is done in $\mathcal{O}(m)$ element comparisons performed using quicksort partitioning and the associated find operations in function Filter. Therefore, it makes sense to think about the constant factors involved here and to compare them with the constant factors involved in the *Jarník–Prim* algorithm. The number of comparisons (and associated finds) can be reduced by a constant factor by choosing pivots more carefully. Therefore, for an input segment of size k, our pivot is the median of a random sample of size $\sqrt{k}$. For the find operations, observe that most of the find operations will follow two single parent references to a common representative. This common case can be made fast as follows: When filtering an edge (u, v), we first load the parent references pu and pv of u and v respectively. When pu = pv, we can immediately discard (u, v). Otherwise, we complete the find operations for u and v and compare the results as usual.[6] All in all, when most edges are filtered out immediately in this way, the resulting $2m$ random memory references may dominate the running time for large, not too sparse graphs – the quicksort partitioning operations work cache efficiently.

Now let us compare this to the best case of the *Jarník–Prim* algorithm where for most edges (u, v), we perform one random memory access to the distance value of v, compare it with the edge weight and discard (u, v) without accessing the priority queue. Since all edges are stored in both directions, we also get $2m$ random memory accesses. Hence, we can expect *Filter-Kruskal* and *Jarník–Prim* to perform similarly for large, sufficiently dense instances.

8.3.3. Parallelization.

Most parts of *Filter-Kruskal* are well suited for parallelization – we can parallelize partitioning, sorting, and filtering. It is interesting to note that the find-operations done for filtering are logically completely independent although, due to path compression, there may be simultaneous read and write accesses to the same parent references. However, no matter how such operations are executed by the hardware, we will get correct results since we maintain the invariant that parent references eventually lead to the representative. Only the union-find operations in the *Kruskal*-call for the base case have to be executed sequentially. On average, these will only be called for $\mathcal{O}(n)$ edges however.

[6] We use the same trick within Kruskal's algorithm.

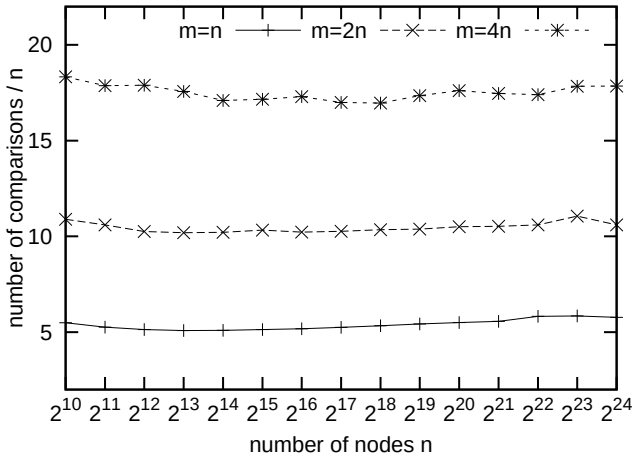


Figure 8.2.: Number of edge comparisons performed by algorithm *Filter-Kruskal+* for random graphs.

We have implemented a multicore parallel version of *Filter-Kruskal*. Sorting and partitioning uses a parallel implementation of the C++ standard template library [159]. Partitioning is done using the inplace parallel algorithm from [167].

8.3.4. More Sophisticated Variants.

Besides *removing* edges during filtering that are not needed for an MST, we can also *identify* some MST edges for good. Consider the multigraph G_s whose nodes are the components currently represented in the union-find data structure and whose edges are $E_s := \{(\mathsf{find}(u), \mathsf{find}(v)) : (u,v) \in E_>\}$ where $E_>$ contains the edges which survived filtering. For an edge $(u,v) \in E_>$, if $\mathsf{find}(u)$ or $\mathsf{find}(v)$ have degree one in G_s, then (u,v) is an MST edge. More generally, all edges outside the two-core[7] of G_s correspond to MST edges and can be found in time linear in the size of G_s.

We have implemented the first variant of the algorithm (henceforth called *Filter-Kruskal+*) since degree-one nodes of G_s can be identified easily by maintaining a counter for each representative. Indeed, counter values 0, 1, and "> 1" suffice and can be stored together with the rank information in the unused parent references of component representatives (see also [43]). Figure 8.2 indicates that we get a near linear number of comparisons for sparse random graphs. Unfortunately, we will see that the overhead even for this simple measure is such that we see no improvement with respect to running time. Therefore we refrain from implementing the two-core refinement because this would even require building a proper adjacency-list representation of G_s which would probably be even slower. For the conversion time, refer to the Figure 8.7.

If we would take the time to build G_s explicitly, it would probably be even better to solve

[7] The k-core of a graph G is the maximal vertex induced subgraph of G with minimal degree k. All k-cores can be found in linear time by subsequently removing small degree nodes.

the MST problem recursively for G_s. With this measure we would move into the direction of a variant of the linear time randomized algorithm [73]. The difference would be that by taking advantage of random edge weights, we do not need a complicated data structure for filtering out heaviest edges on a cycle. Instead, we recurse on the lighter half of the edges and use a simple union-find data structure (which for this application could be made to run in linear time). The only missing ingredient to a linear time algorithm would be a node reduction algorithm. We could use the traditional Boruvka algorithm [119], or the sequential node reduction from [43]. The latter algorithm has an interesting deterministic variant where in each step, we remove the node with minimum degree. We have not implemented any of these algorithms because we do not think they could be competitive to our simple *Filter-Kruskal* algorithm in practice.

8.3.5. Experiments

Algorithms. We have experimented with *Kruskal*, *qKruskal* – the *Kruskal* modification from [128], *Filter-Kruskal*, *Filter-Kruskal+* from Subsection 8.3.4, and several variants of *Jarník–Prim* (*JP*). For *JP* we only show results for the best implementations: *pJP* for Irit Katriel's implementation with pairing heaps [84] and *qJP*, our own implementation combined with Paredes's quickHeap priority queue [128]. *qJP* is considerably faster than Paredes's own code since we use a faster graph representation (adjacency arrays rather than adjacency lists). We also use a multicore implementation of *Filter-Kruskal* (*Filter-Kruskal* P for P cores) and a version of *Kruskal* with parallel sorting of edges (*KruskalP*). Our graph data structure implements the interface of the Boost Graph Library [158], but uses a graph representation that is specific to the algorithm. For the variants of Kruskal's algorithm this is simply an array of edges, for the *JP* algorithm, we use an adjacency array representation. We have also measured the time needed for converting between these representations.

Implementation. The implementation uses C++ with the GNU compiler version 4.3.1 and optimization level O3. The experiments were run on a **platform H**, see Section 3.2.

Instances. Unfortunately, there is no established suite of real world instances for MST problems. Mostly, synthetic graphs families from the study [114] were used in the past. From these we use random graphs with random edge weights, random graphs with weights that force a decrease_key for every edge, and random geometric graphs where n random points in the unit square are connected with their k closest neighbors with respect to Euclidean distance. Note that the resulting edge weights are not independent random numbers. We also use lollipop graphs with random edge weights where a path of length $n/2$ is appended to a random graph with $n/2$ nodes. Perhaps most interestingly, we have obtained a few instances generated by the image segmentation method by Jan Wassenberg (see also [48]), that was applied to satelite images, [162].

Random Edge Weights. Figure 8.3 (left) shows the running time of the algorithms discussed above for random graphs with random edge weights. Lets first consider the middle

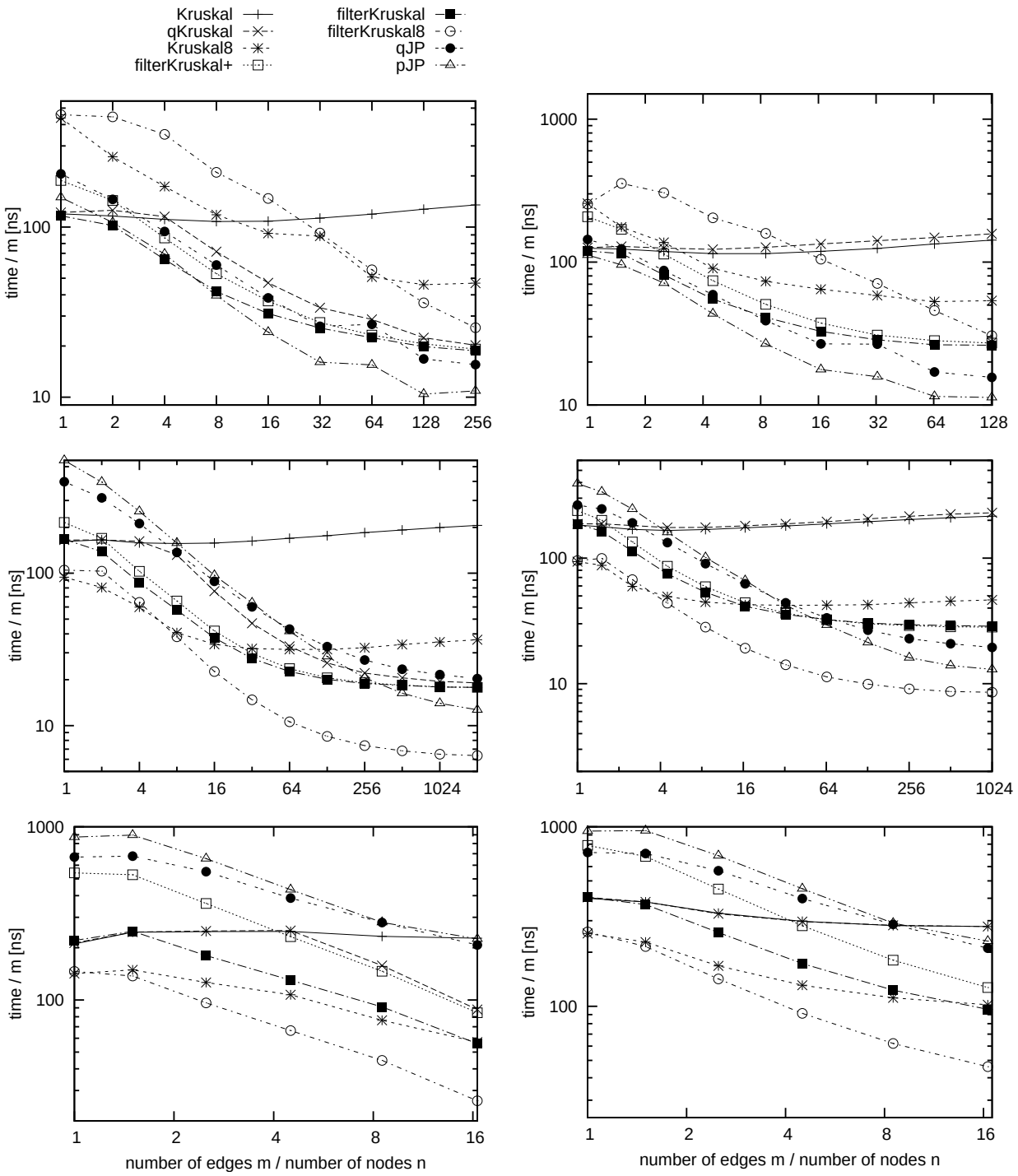


Figure 8.3.: On the left: time per edge for random graphs with random edge weights and 2^{10} (top), 2^{16} (middle), and 2^{22} (bottom) nodes. On the right: time per edge for lollipop graphs with random edge weights and 2^{11} (top), 2^{17} (middle), and 2^{23} (bottom) nodes

graph with measurements for $n = 2^{16}$ nodes. Kruskal's algorithm performs well for up to $8n$ edges where it is also well parallelizable. For more dense graphs, *JP* is better. None

of the two priority queue variants is a clear winner. Quickheaps are a bit better for very sparse graphs whereas pairing heaps win for more dense graph. *qKruskal* does improve on *Kruskal* and outperforms *qJP*. *Filter-Kruskal* shows uniformly good performance over the entire range of densities. It is clearly better than *qKruskal* and only for rather dense graph it is still outperformed by *JP*. On 8 cores, *Filter-Kruskal* becomes the clear winner. Note that a parallel implementation of *JP* does not look promising except for very large, very dense graphs where parallelizing the innermost loop becomes interesting.

A more direct comparison to the sophisticated parallel MST implementations by Bader and Cong [15] would be interesting. However, they only report speedups for at least one million nodes and our codes are considerably faster than their codes if one simply scales the clock frequency of the machines. Hence, it currently looks like our algorithms are better at least for a small number of cores and in particular for small inputs.

Somewhat disappointingly, the "improved" Algorithm *Filter-Kruskal+* is always slightly *slower* than *Filter-Kruskal* – even the moderate additional effort for including degree-one edges never really pays off. We view this as an indication that even more complicated algorithms like [73] are even more far from being practical than we thought.

We have performed analogous experiments for smaller and larger random graphs with $n = 1024$ and $n = 2^{22}$ (see Figure 8.3, left). The ranking of the algorithms is similar as before, except that for very large graphs, *Filter-Kruskal* consistently outperforms *JP*. For small graphs, it is not astonishing that parallelization is not worthwhile. Here, *pJP* is the best algorithm throughout whereas *qJP* performs worse than both *pJP* and *Filter-Kruskal*.

Lollipop Graphs. For lollipop graphs (see Figure 8.3, right) we see similar result as for random graphs. The biggest difference is that *qKruskal* is now no better than *Kruskal*. *JP* ourperforms sequential *Filter-Kruskal* for sufficiently dense graphs but not by a large margin.

Difficult Instances. For the difficult instances, we see in Figure 8.4 that *qJP* becomes extremely slow, *pJP* and *Filter-Kruskal* are now somewhat worse than *Kruskal*, *qKruskal* yields a slight improvement over *Kruskal* and parallel *Kruskal* is the best algorithm.

Random Geometric Graphs. For random geometric graphs (see Figure 8.5) with 2^{16} nodes, we again have similar ranking as for random graphs, except that this time *Filter-Kruskal* outperforms the *JP* variants.

Real World Instances. Surprisingly, for the image segmentation instances shown in Figure 8.6, *Filter-Kruskal* is again the best algorithm. As for lollipop graphs, *qKruskal* performs no better than *Kruskal* which is a confirmation of the intuition that this heuristics is not very robust.

Summary. The bottom line is that *Kruskal* remains a good algorithm for very sparse graphs and *Filter-Kruskal* and *pJP* contend for the best performance on more dense instances. We tend to give preference to *Filter-Kruskal* for three reasons. First, it shows good performance also for sparse graphs. Second, it is easily parallelizable, yielding a speedup of above two

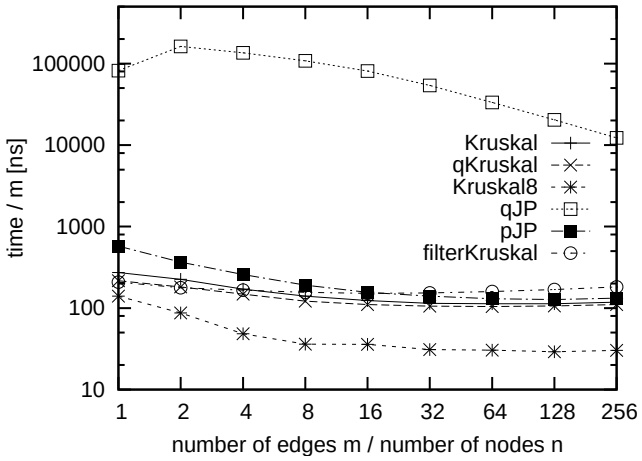


Figure 8.4.: Time per edge for graphs that are bad for *JP* with 2^{16} nodes.

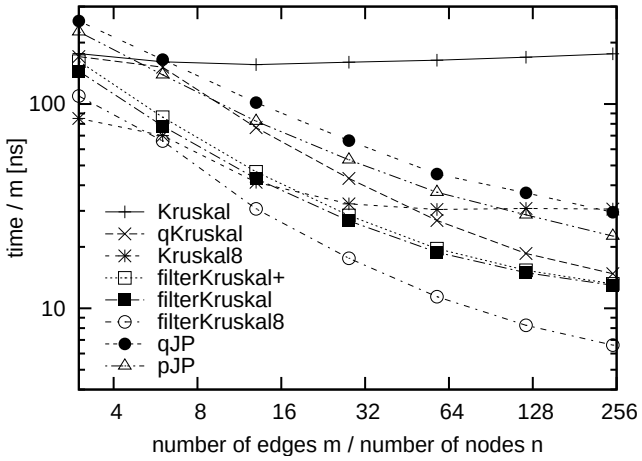


Figure 8.5.: Time per edge for random geometric graphs with 2^{16} nodes.

already on low cost servers. Third, it only requires a list of edges as its input whereas *JP* needs a full fledged adjacency array. Figure 8.7 shows that building an adjacency array from a list of edges can take an order of magnitude longer than computing the MST![8] This means that in cases where the adjacency array is not available, *Filter-Kruskal* will be much faster than *JP*. In contrast, building a list of edges from an adjacency array is very fast, indeed, the times given in Figure 8.7 are overestimates because we could fuse the loops for conversion and for the top level partitioning – scan the adjacency array and output to a partitioned array of edges.

[8] We use the standard conversion algorithm that essentially performs a bucket sort on the endpoints of the edges [105, page 169], causing $\approx 10m$ cache faults. For large instances, this could be somewhat accelerated by using multipass algorithms but it is unlikely that the general picture gets reversed.

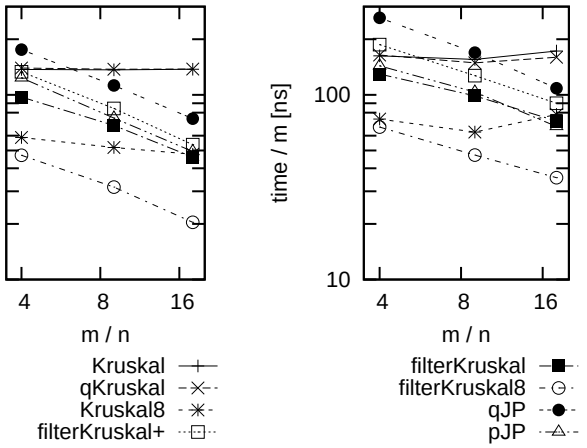


Figure 8.6.: Time per edge for two families of graphs stemming from two satelite image segmentation problems for two different resolutions (800 000 nodes on the left, 4 163 616 nodes on the right) and different number of pixels considered in the "neighborhood" of each pixel.

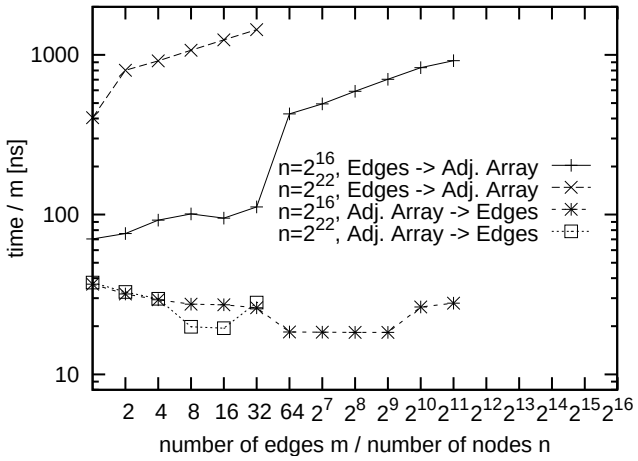


Figure 8.7.: Time per edge for converting between edge sequences and adjacency arrays.

8.4. Conclusions

Enhancing Kruskal's algorithm with a simple filtering step leads to considerably improved performance. In particular, for arbitrary graphs with random edge weights, we obtain linear expected execution time for all but rather sparse graphs. It seems that this also applies to edge weights occurring in some real world applications. The resulting *Filter-Kruskal* algorithm not only outperforms Kruskal's algorithm but is also competitive with the *Jarník–Prim* algorithm even for dense graph. *Filter-Kruskal* considerably outperforms *Jarník–Prim* if multiple cores are available or if the adjacency array is not given as part of the input.

The more sophisticated *Filter-Kruskal+* algorithm that also includes some edges into the MST without sorting, is interesting because it seems to yield a practical and relatively simple algorithm with average case linear execution time. Its somewhat disappointing practical performance might be offset in the future by more opportunities for parallelization. First experiments indicate that its nonparallelizable component, that is union-find operations within *Kruskal*-call for the base case, grows sublinearly with n (something like $n^{0.6}$).

Acknowledgements.

We would like to thank Irit Katriel, Rodrigo Paredes, Dominik Schultes and Jan Wassenberg for providing graph generators, instances, and source codes. We also thank Martin Dietzfelbinger, Gonzalo Navarro, Rodrigo Paredes for fruitful discussions and Jyrki Katajainen for useful comments on the original paper.

CHAPTER 9

Graph Matching

A matching M of a graph $G = (V, E)$ with $|V| = n, |E| = m$ is a subset of edges such that no two elements of M have a common end point. Many applications require the computation of matchings with certain properties, like being maximal (no edge can be added to M without violating the matching property), having maximum cardinality, or having maximum total weight $\sum_{e \in M} w(e)$.

The first polynomial time maximum weight matching algorithm for general graphs is by Edmonds and dates back to the 60s [47]. Currently the best known theoretical bound is by Gabow [54] for general graphs, though there exist better algorithms for restricted graph classes [93], or edge weights [55].

Although as maximum weight as maximal matching problems can be solved optimally in polynomial time, exact algorithms are not fast enough for many applications involving large graphs where we need near linear time algorithms. For example, the most efficient algorithms for graph partitioning rely on repeatedly contracting maximal matchings, often trying to maximize some edge rating function w. Refer to [68] and Chapter 10 for details and examples. A folklore algorithm achieving 1/2-approximation ratio sorts the edges by weight and greedily adds edges into the matching by scanning edges in decreasing order [14]. Note, that the algorithm is linear for integer edge weights. Preis [133] proposed the first linear time 1/2-approximation algorithm for arbitrary edge weights. Drake and Hougardy [45] designed a simple algorithm within the same runtime and approximation bounds that they called Path Growing Algorithm (PGA) and another $(2/3 - \epsilon)$-algorithm running in $\mathcal{O}(n/\epsilon)$. Sanders and Pettie developed en even simpler $\mathcal{O}(n \log 1/\epsilon)$ algorithm achieving the same approximation ratio [132]. By combining greedy [14] with a path growing heuristic [45] Maue and Sanders designed a Global Path Algorithm (GPA) [103], which is faster and often better than the $(2/3 - \epsilon)$ algorithm[132].

By using an idea that we expoited in our Filter-Kruskal algorithm in Chapter 8 we propose a modification to the original GPA algorithm that allows the algorithm to avoid sorting of edges that can not be added to the matching it constructs. Our modified algorithm

Filter-GPA is almost always faster than GPA and allows easy parallelization similarly to Filter-Kruskal.

Due to the inherently sequential part of our Filter-GPA its usage for highly parallel architectures is limited. Therefore, we turn to even simpler algorithm, a *local max* [66]. We propose an approach for implementing the *local max* algorithm for computing matchings that is easy to adapt to many models of computation. We show that for computing maximal matchings the algorithm needs only linear work on a sequential machine and in several parallel models.

In Section 9.4 we show runtime benefits of our sequential and multicore Filter-GPA. We also evaluate performance of our local max implementation on commodity Graphical Processing Units (GPUs). Our experiments indicate that local max yields surprisingly good quality in comparison to GPA for the weighted matching problem and runs very efficiently as on sequential machines as on GPUs.

References. The contents of this chapter is based on the joint work with Marcel Birn, Peter Sanders, Christian Schulz and Nodari Sitchinava [23]. Most of the wording of the original publication is preserved. Filter-GPA part is based on unpublished work with Michael Axtmann.

9.1. Global Path Algorithm

Algorithm 8: Pseudocode for *GPA*

```
1  GPA(E, M : Sequence of Edge, P : PathDS)
2  begin
3      P := ∅
4      Sort (E)                                   /* by decreasing weight */
5      foreach e ∈ E do
6          if IsApplicable(e, P) then add e to P
7      end
8      foreach Path or cycle P in P do
9          M' := MaxWeightMatching(P)
10         M := M ∪ M'
11     end
12 end
13
14 IsApplicable(e = {u, v} : Edge, P : PathDS)
15 begin
16     return u and v are endpoints of either an odd length path P ∈ P or two different
       paths P_1, P_2 ∈ P
17 end
```

The original Global Path Algorithm (GPA) [103] is given in Pseudocode in Algorithm 8.

Similarly to Greedy, [14] GPA sorts the edges by weight first and then adds *applicable* edges into the *path data structure* by traversing the edges in decreasing order. A *path data structure* $\mathcal{P}$ is a collection of paths and even length cycles. At the beginning of the algorithm we can assume that each node of G is a singleton path. An edge $e = \{u, v\}$ is called *applicable* if either u and v are end points of two different paths P_1 and P_2 in $\mathcal{P}$. Or they are endpoints of a single odd length path in $\mathcal{P}$. Having added all applicable edges into $\mathcal{P}$, the algorithm uses dynamic programming to compute an optimal maximum weight matching of $\mathcal{P}$. Maue and Sanders show that the algorithm yields approximation ratio of $1/2$ for maximum weight matching of the original graph. Since the path data structure can be implemented to support constant updates, the running time is dominated by sorting, and thus is $\mathcal{O}(m \log n)$.

9.1.1. Filter-GPA

Algorithm 9: Pseudocode for *Filter-GPA*

```
1  Filter-GPA(E : Sequence of Edge, P : PathDS)
2  begin
3      M := ∅
4      if m ≤ GPAThreshold(n, m) then
5          Sort (E)                                   /* by decreasing weight */
6          foreach e ∈ E do
7              if IsApplicable(e, P) then  add e to P
8          end
9      else
10         pick a pivot p ∈ E
11         E≥ := ⟨e ∈ E : e ≥ p⟩
12         E< := ⟨e ∈ E : e < p⟩
13         Filter-GPA(E≥, P)
14         E< := Filter(E<, P)
15         Filter-GPA(E<, P)
16     if top level of recursion then
17         foreach Path or cycle P in P do
18             M' := MaxWeightMatching(P)
19             M := M ∪ M'
20         end
21         return M
22 end
23
24 Filter(E : Sequence of Edge, P : PathDS)
25 begin
26     return ⟨e = {u, v} ∈ E : IsApplicable(e,P)⟩
27 end
```

Similarly to the Filter-Kruskal algorithm described in Chapter 8, we modify the original GPA algorithm to avoid sorting of edges that can not be part of the matching produced by GPA, see Algorithm 9.

If the number of edges exceeds some GPAThreashold, we either choose a random pivot edge p and partition the input list of edges into two parts $E_{\geq}$ and $E_{<}$. Or we sort the edges and compute the set of paths and even cycles similar to the original GPA. Further, we recurse on $E_{\geq}$. Having computed an intermediate set of paths and even length cycles $\mathcal{P}$ of $E_{\geq}$, we first apply Filter subroutine that filters out nonapplicable edges from $E_{<}$ and only than recurse on $E_{<}$. In this manner, we compute the set of applicable edges $\mathcal{P}$ of the whole input edge list E. Therefore we can apply dynamic programming to obtain an optimal maximum weight matching of $\mathcal{P}$, whose weight is at least $1/2$ the weight of the optimal matching of the input graph.

Note, if we use a common deterministic tie breaking for edge weights, the matchings computed by GPA and Filter-GPA will be the same.

In Section 9.4 we show that the denser the graph is the more considerably Filter-GPA outperforms GPA. A simple parallelization further improves its performance.

Parallelization. Some parts of GPA and Filter-GPA are easy to parallelize. This includes sorting, partitioning and filtering. In terms of Standard Template Library (STL) these procedures correspond to sort, partition, copy_if On the other hand adding applicable edges to $\mathcal{P}$ and dynamic programming are difficult to parallelize.

9.1.2. Analysis for Random Edge Weights

Let $G = (V, E)$ be an undirected graph with random edge weights, $|V| = n$, $|E| = m$ and $w = E \to \mathcal{R}$. Let R be any subset of E. For simplicity we assume that all edge weights are different, which is easy to achieve if we break ties by edge ids.

Definition 1. *A set of edges $S \subseteq R$ satisfies the* path-cycle *property for R if*

1. *S is a set of vertex disjoint paths and even-length cycles*
2. *S is maximal.*

In this case, we say that S is a PC-Set of R or just $S \in \mathsf{PC\text{-}Set}(R)$ for short, where $\mathsf{PC\text{-}Set}(R)$ is a set of sets that satisfy Definition 1.

The second property of Definition 1 means that S can not be extended by any additional edge $e \in R$ without violating the first property.

Let $X_{\geq y}(X_{>y}$ respectively) denote a set of edges $\{e \in X : w(e) \geq w(y)\}$ ($\{e \in X : w(e) > w(y)\}$ respectively) for some $y \in E$.

Definition 2. *An edge $\bar{e} \in E$ is called* heavy *for $S \in \mathsf{PC\text{-}Set}(R)$ if*

- *either $\bar{e} \in S$*
- *or $S_{\geq \bar{e}} \cup \bar{e} \in \mathsf{PC\text{-}Set}(R_{\geq \bar{e}} \cup \bar{e})$*

Definition 3. $S \in \mathsf{PC\text{-}Set}(R)$ *is called a* heavy weight set satisfying the path-cycle property *if* $\forall e \in R \setminus S : e$ *is not heavy for* S.

We say that S is a $\mathsf{HPC\text{-}Set}$ of R or $S \in \mathsf{HPC\text{-}Set}(R)$ for short.

Lemma 7. *Let* $\{R = \{e_1, e_2, \ldots, e_r\} \subseteq E : w(e_1) > w(e_2) > \ldots > w(e_r)\}$.
Then for $0 \leq i < r$

$$S_{i+1} := \begin{cases} e_1, & \text{if } i = 0; \\ S_i \cup e_{i+1}, & \text{if } e_{i+1} \text{ is heavy for } S_i \\ S_i, & \text{otherwise.} \end{cases}$$

is a $\mathsf{HPC\text{-}Set}$ *of* $R_{\geq e_{i+1}}$.

Proof. For simplicity let X_i denote $X_{\geq e_i}$ for $X \subseteq R$.
By induction on i.
Base of induction: $i = 0$.
$R_1 = \{e_1\}$ and it easy to see that $S_1 = \{e_1\} \in \mathsf{HPC\text{-}Set}(R_1)$.
Induction step $i > 0$.
If e_{i+1} is heavy for $S_i \in \mathsf{HPC\text{-}Set}(R_i)$, then by Definition 2

$$S_i \cup e_{i+1} \in \mathsf{PC\text{-}Set}(R_{i+1})$$

Now, we prove that $S_i \cup e_{i+1} \in \mathsf{HPC\text{-}Set}(R_{i+1})$ by contradiction

$$\exists 1 < \bar{\imath} < i+1 : e_{\bar{\imath}} \in R_{i+1} \setminus (S_i \cup e_{i+1}) \wedge e_{\bar{\imath}} \text{ is heavy for } S_i \cup e_{i+1}$$

then by Definition 2,

$$(S_i \cup e_{i+1})_{\bar{\imath}} \cup e_{\bar{\imath}} \in \mathsf{PC\text{-}Set}((R_{i+1})_{\bar{\imath}} \cup e_{\bar{\imath}})$$

By inductive assumption

$$(S_i)_{\bar{\imath}} = S_{\bar{\imath}} \in \mathsf{HPC\text{-}Set}(R_{\bar{\imath}}).$$

Therefore,

$$(S_i \cup e_{i+1})_{\bar{\imath}} \cup e_{\bar{\imath}} = S_{\bar{\imath}} \cup e_{\bar{\imath}} \in \mathsf{PC\text{-}Set}(R_{\bar{\imath}} \cup e_{\bar{\imath}})$$

Since

$$e_{\bar{\imath}} \notin S_i \supseteq S_{\bar{\imath}} \Rightarrow S_{\bar{\imath}-1} = S_{\bar{\imath}},$$

$e_{\bar{\imath}}$ is heavy for $S_{\bar{\imath}-1}$ and the algorithm should have added $e_{\bar{\imath}}$ to the $S_{\bar{\imath}-1} \subseteq S_i$.
A contradiction.

The last case to consider is when e_{i+1} is not heavy for S_i.
By Definition 2

$$S_i \cup e_{i+1} \notin \mathsf{PC\text{-}Set}(R_i \cup e_{i+1}) = \mathsf{PC\text{-}Set}(R_{i+1})$$

Therefore $S_i \in \mathsf{HPC\text{-}Set}(R_i)$ can not be extended by e_{i+1} on R_{i+1} and, therefore, $S_i \in \mathsf{PC\text{-}Set}(R_{i+1})$. The fact, that $S_i \in \mathsf{HPC\text{-}Set}(R_{i+1})$ can be proven analogously to the previous case. □

Corollary 1. *For distinct edge weights* $|\mathsf{HPC\text{-}Set}(R)| = 1$

Proof. It is clear that for distinct edge weights the sets S_i are unique. In particular $S_r \in \mathsf{HPC\text{-}Set}(R)$ is unique. Assume now that

$$\exists \bar{S} \subseteq R : \bar{S} \in \mathsf{HPC\text{-}Set}(R) \wedge \bar{S} \neq S_r.$$

We prove by induction on i that $\bar{S}_i = S_i$.
Base of induction $i = 1$. It is easy to show that $e_1 \in \bar{S}$. Therefore $\bar{S}_1 = e_1 = S_1$.
Assume that $\bar{S}_i = S_i$ and consider $e_{i+1} \in R_{i+1}$.

$$e_{i+1} \in S_{i+1} \Leftrightarrow e_{i+1} \text{ is heavy for } S_i \Leftrightarrow e_{i+1} \text{ is heavy for } \bar{S}_i$$
$$\Leftrightarrow \bar{S}_i \cup e_{i+1} \in \mathsf{PC\text{-}Set}(R_{i+1}) \Leftrightarrow e_{i+1} \in \bar{S} \Leftrightarrow e_{i+1} \in \bar{S}_{i+1} \quad \square$$

Corollary 2. *GPA computes a set* $\mathsf{GPA} \in \mathsf{HPC\text{-}Set}(E)$.

Proof. GPA traverses the edges $e \in E$ in decreasing order. Let $e_i \in E$ be the i-th edge in this order. And E_i denote the set $E_{\geq e_i}$, which is the set of traversed edges at the point of time GPA looks at the edge e_i. Since GPA never removes edges from the set GPA it constructs, let GPA_i denote the set of edges added to the GPA after considering e_i.

In order to prove the claim it is sufficient to show that $\mathsf{GPA}_i = S_i \in \mathsf{HPC\text{-}Set}(E_i)$ or, equivalently, by using Lemma 7 for $R = E$, that GPA adds an edge e_{i+1} into $\mathsf{GPA} \Leftrightarrow e_{i+1}$ is heavy for S_i.

This is again easy to see by induction.
The base of induction $E_1 = e_1$ is trivial.
Induction step. GPA adds an edge e_{i+1} to GPA_i if and only if it is applicable, that is $\mathsf{GPA}_i \cup e_{i+1}$ is a set of paths and even-length cycles. Since by the inductive hypothesis

$$\mathsf{GPA}_i = S_i \in \mathsf{HPC\text{-}Set}(E_i)$$

this condition is equivalent to $S_i \cup e_{i+1} \in \mathsf{PC\text{-}Set}(E_i \cup e_{i+1})$, which in turn equivalent to e_{i+1} being heavy for S_i. $\square$

Lemma 8. $\forall R \subseteq E : e \in E$ *is heavy for* $S^{(1)} \in \mathsf{HPC\text{-}Set}(R) \Leftrightarrow e \in S^{(2)} \in \mathsf{HPC\text{-}Set}(R \cup e)$

Proof. $e \in S^{(2)} \in \mathsf{HPC\text{-}Set}(R \cup e) \Leftrightarrow e$ was added to $S^{(2)}_{>e} \in \mathsf{HPC\text{-}Set}\left((R \cup e)_{>e}\right) \Leftrightarrow e$ is heavy for $S^{(2)}_{>e} = S^{(1)}_{>e} \in \mathsf{HPC\text{-}Set}(R_{>e}) \Leftrightarrow e$ is heavy for $S^{(1)} \in \mathsf{HPC\text{-}Set}(R)$. $\square$

Lemma 9 (Sampling Lemma). *For a random* $R \subseteq E$ *of size* r*, the expected number of edges that are heavy for* $S \in \mathsf{HPC\text{-}Set}(R)$ *is* $< mn/r$

Proof. Similar to Chan [33] we pick a random edge $e \in E$ (independent of R). It suffices to show that e is heavy for S with probability $< n/r$. By Lemma 8, it suffices to bound the probability that $e \in S' \in \mathsf{HPC\text{-}Set}(R \cup e)$.

We follow "backward analysis" approach: instead of adding an edge to R we imagine deleting a random edge from $R' = R \cup e$. Since e is equally likely to be any of edges of S' and S' has at most n edges (if we have exactly one even length cycle), the probability that $e \in S'$ conditioned to fixed choice of R' is at most $n/|R'| = n/r$. As this upper bound does not depend on R', it holds unconditionally and the result is proven. $\square$

Since checking if an edge is applicable and adding it to the current set of paths and even-length cycles can be done in $\mathcal{O}(1)$ (corresponding to find and union operations in Lemma 4 respectively) we can formulate the following lemma, similar to Lemma 4.

Lemma 10. *Let C denote the number of (edge weight) comparisons performed by Filter-GPA. Then Filter-GPA performs expected $\mathcal{O}(n + 3m + C)$ operations.*

Having established sampling lemma Lemma 9 and Lemma 10 we can formulate the following theorem, which is again similar to Theorem 6.

Theorem 11. *Given an arbitrary graph $G = (V, E)$ with random edge weights, the expected running time of Filter-GPA is $\mathcal{O}\big(m + n \log(n) \log \frac{m}{n}\big)$.*

The proofs of Lemma 10 and Theorem 11 are completely analogous to their MST counterparts, therefore we omit them here.

9.2. Massively Parallel Matchings

Though our Filter-GPA algorithm allows partial parallelization, which might be already sufficient for multicores with small number of cores, the limited parallelizability makes a problem for massively parallel architectures like GPUs. Therefore we are forced to turn to alternatives that would allow us to fully utilize massively parallel hardware.

Parallel matching algorithms have been widely studied. There is even a book on the subject [79] but most theoretical results concentrate on work-inefficient algorithms. The only linear work parallel algorithms that we are aware of are randomized CRCW PRAM algorithms by Israeli and Itai [69] and Blelloch et al. [27]. We will call them IIM and BFSM, respectively. IIM runs in expected $\mathcal{O}(\log n)$ time and BFSM runs in $\mathcal{O}\big(\log^3 n\big)$ time with high probability.

Fagginger Auer and Bisseling [13] study an algorithm similar to [69] which we call red-blue matching (RBM) here. They implement RBM on shared memory machines and GPUs. They prove good shrinking behavior for random graphs, however, provide no analysis for arbitrary graphs.

Our CRCW PRAM variant of local max algorithm [66] matches the optimal asymptotic bounds of IIM. However, our algorithm is simpler (resulting in better constant factors), removes higher fraction of edges in each iteration (IIM's proof shows less than 5% per iteration, while we show at least 50%) and our analysis is a lot simpler. We also provide the first CREW PRAM algorithm which performs linear work and runs in expected $\mathcal{O}\big(\log^2 n\big)$ time.[1] (see Section 9.3). Compared to RBM, the local max implementations remove more edges in each iteration and provide better quality results for the weighted case.

[1] While a generic simulation of IIM on the CREW PRAM model will result in a $\mathcal{O}\big(\log^2 n\big)$ time algorithm, the simulation incurs $\mathcal{O}(n \log n)$ work due to sorting.

9.3. Local Max

Here we consider the following simple *local max* algorithm [66]: Call an edge locally maximal, if its weight is larger than the weight of any of its incident edges; for unweighted problems, assign unit weights to the edges. When comparing edges of equal weight, use tie breaking based on random perturbations of the edge weights. The algorithm starts with an empty matching M. It repeatedly adds locally maximal edges to M and removes their incident edges until no edges are left in the graph. The result is obviously a maximal matching (every edge is either in M or it has been removed because it is incident to a matched edge). The algorithm falls into a family of weighted matching algorithms for which Preis [133] shows that they compute a 1/2-approximation of the maximum weight matching problem. Hoepman [66] derives the local max algorithm as a distributed adaptation of Preis' idea. Based on this, Manne and Bisseling [100] devise sequential and parallel implementations. They prove that the algorithm needs only a logarithmic number of iterations to compute maximal matchings by noticing that a maximal matching problem can be translated into a maximal independent set problem on the *line graph* which can be solved by Luby's algorithm [94]. However, this does not yield an algorithm with linear work since it is not proven that the edge set indeed shrinks geometrically.[2] Manne and Bisseling also give a sequential algorithm running in time $\mathcal{O}(m \log \Delta)$ where Δ is the maximum degree. On a NUMA shared memory machine with 32 processors (SGI Origin 3800) they get relative speedup < 6 for a complete graph and relative speedup ≈ 10 for a more sparse graph partitioned with Metis. Since this graph still has average degree ≈ 200 and since the speedups are not impressive, this is a somewhat inconclusive result when one is interested in partitioning large sparse graphs on a larger number of processors.

Our central observation is:

Lemma 12. *[23] Each iteration of the local max algorithm for the unit weight case removes at least half of the edges in expectation.*

Yves et al. [175] uses a similar proof technique to define "preemptive removal" of nodes for distributed maximal independent set problem.

Assume now that each iteration can be implemented to run with work linear in the number of surviving edges (independent of the number of nodes). Working naively with the expectations, this gives us a logarithmic number of rounds and a geometric sum leading to linear total work for computing a maximal matching. This can be made rigorous by distinguishing *good* rounds with at least $m/4$ matched edges and bad rounds with less matched edges. By Markov's inequality, we have a good round with constant probability. This is already sufficient to show expected linear work and a logarithmic number of expected rounds. We skip the details since this is a standard proof technique and since the resulting constant factors are unrealistically conservative. An analogous calculation for median selection can be found in [105, Theorem 5.8]. One could attempt to show a shrinking factor close to 1/2 rigorously by showing that large deviations (in the wrong direction) from the expectation are unlikely

[2] Manne and Bisseling show such a shrinking property under an assumption that unfortunately does not hold for all graphs.

(e.g., using Martingale tail bounds). However this would still be a factor two away from the more heuristic argument in Footnote 4 and thus we stick to the simple argument.

There are many ways to implement an iteration which of course depend on the considered model of computation.

9.3.1. Sequential Model

For each node v maintain a candidate edge $C[v]$, originally initialized to a dummy edge with zero weight. In an iteration go through all remaining edges $e = \{u, v\}$ three times. In the first pass, if $w(e) > w(C[u])$ set $C[u]{:=}\, e$ (add random perturbation to $w(e)$ in case of a tie). If $w(e) > w(C[v])$ set $C[v]{:=}\, e$. In the second pass, if $C[u] = C[v] = e$ put e into the matching M. In the third pass, if u or v is matched, remove e from the graph. Otherwise, reset the candidate edge of u and v to the dummy edge. Note that except for the initialization of C which happens only once before the first iteration, this algorithm has no component depending on the number of nodes and thus leads to linear running time in total if Lemma 12 is applied.

9.3.2. CRCW PRAM Model.

In the most powerful variant of the *Combining CRCW PRAM* that allows concurrent writes with a maximum reduction for resolving write conflicts, the sequential algorithm can be parallelized directly running in constant time per iteration using m processors.

9.3.3. MapReduce Model.

The CRCW PRAM result together with the simulation result of Goodrich et al. [61] immediately implies that each iteration of local max can be implemented in $\mathcal{O}(\log_M n)$ rounds and $\mathcal{O}(m \log_M n)$ communication complexity in the MapReduce model, where M is the size of memory of each compute node. Since typical compute nodes in MapReduce have at least $\Omega(m^\epsilon)$ memory [77], for some constant $\epsilon > 0$, each iteration of local max can be performed in MapReduce in constant rounds and linear communication complexity.

9.3.4. External Memory Models.

Using the PRAM emulation techniques for algorithms with geometrically decreasing input size from [36, Theorem 3.2] the above algorithm can be implemented in the external memory [3] and cache-oblivious [52] models in $\mathcal{O}(sort(m))$ I/O complexity, which seems to be optimal.

9.3.5. $\mathcal{O}\big(\log^2 n\big)$ work-optimal CREW solution

In this section, we present a $\mathcal{O}\big(\log^2 n\big)$ CREW PRAM algorithm, which incurs only $\mathcal{O}(m)$ work.

Consider the following representation of the graph $G = (V, E)$. Let V be a totally ordered set, i.e., given two vertices $u, v \in V$ we can uniquely determine whether $u < v$ or not. Let $\mathtt{e}$ be an array of undirected edges with each entry $\mathtt{e}[\mathtt{k}]$ storing all the information of a single edge $\{u, v\} \in E$, i.e., vertex endpoints u and v, its weight or any other auxiliary data. Let $\mathtt{A}$ be an array of tuples (v, e_k), where $v \in V$ and e_k is the *pointer* to $\mathtt{e}[\mathtt{k}]$ representing the edge $\{u, v\}$. Let $\mathtt{A}$ be sorted by the first entry, i.e. all tuples (v, e_k) pointing to the edges incident on the same vertex v are in contiguous space in $\mathtt{A}$.

Note that any edge $\mathtt{e}[\mathtt{k}] = \{\mathtt{u}, \mathtt{v}\}$ contains two corresponding entries in $\mathtt{A}$ pointing to it: (u, e_k) and (v, e_k). During our algorithm, a processor responsible for (u, e_k) might need to find and update entry (v, e_k) (and vice versa). The following lemma describes how to compute for each entry (u, e_k) the index of the corresponding entry (v, e_k) in $\mathtt{A}$.

Lemma 13. *For every edge $\mathtt{e}[\mathtt{k}] = \{\mathtt{u}, \mathtt{v}\}$ entries $\mathtt{A}[\mathtt{i}] = (\mathtt{u}, \mathtt{e_k})$ and $\mathtt{A}[\mathtt{j}] = (\mathtt{v}, \mathtt{e_k})$ of $\mathtt{A}$ can compute each other's index in $\mathtt{A}$ in $\mathcal{O}(1)$ time and $\mathcal{O}(|\mathtt{A}|)$ work in the CREW PRAM model.*

Lemma 14. *Using our graph representation, each node v in the graph can apply an associative operator $\oplus$ to all edges incident on v in $\mathcal{O}(\log |\mathtt{A}|)$ time and $\mathcal{O}(|\mathtt{A}|)$ work on the CREW PRAM model.*

Now we are ready to describe the solution to the matching problem. We perform the following in each phase of the local max algorithm.

1. Each edge $\mathtt{e}[\mathtt{k}]$ picks a random weight w_k.

2. Using Lemma 14, each vertex v identifies k' such that $\mathtt{e}[\mathtt{k}']$ is the heaviest edge incident on v by applying the associative operator MAX to the edge weights picked in the previous step.

3. Using Lemma 13, each entry $(v, e_{k'})$ checks if $\mathtt{e}[\mathtt{k}'] = \{\mathtt{u}, \mathtt{v}\}$ is also the heaviest incident edge on u. If so, the smaller of u and v adds $e_{k'}$ to the matching and sets the deletion flag $f = 1$ on $\mathtt{e}[\mathtt{k}']$.

4. Using Lemma 14, each entry $(v, e_{k'})$ spreads the deletion flag over all edges incident on v by applying MAX associative operator on the deletion flags of incident edges on v. Thus, if at least one edge incident on v was added to the matching, all edges incident on v will be marked for deletion.

5. Now we must prepare the graph representation for the next phase by removing all entries of $\mathtt{e}$ and $\mathtt{A}$ marked for deletion, compacting $\mathtt{e}$ and $\mathtt{A}$ and updating the pointers of $\mathtt{A}$ to point to the compacted entries of $\mathtt{e}$. To perform the compaction, we compute for each entry $\mathtt{e}[\mathtt{k}]$, how many entries $\mathtt{e}[\mathtt{i}]$ and $\mathtt{A}[\mathtt{i}]$, $\mathtt{i} \leq \mathtt{k}$ must be deleted. This can be accomplished using parallel prefix sums on the deletion flags of each entry in $\mathtt{e}$ and $\mathtt{A}$. Let the result of prefix sums for edge $\mathtt{e}[\mathtt{k}]$ be d_k and for entry $\mathtt{A}[\mathtt{i}]$ be r_i. Then $k - d_k$ is the new address of the entry $\mathtt{e}[\mathtt{k}]$ and $i - r_i$ is the new address of $\mathtt{A}[\mathtt{i}]$ once all edges marked for deletion are removed.

6. Each entry $\mathtt{e}[\mathtt{k}]$ that is not marked for deletion copies itself to $\mathtt{e}[\mathtt{k} - \mathtt{d}_\mathtt{k}]$. The corresponding entry $(v, e_k) \in \mathtt{A}$ updates itself to point to the new entry $\mathtt{e}[\mathtt{k} - \mathtt{d}_\mathtt{k}]$, i.e., (v, e_k) becomes (v, e_{k-d_k}), and copies itself to $\mathtt{A}[\mathtt{i} - \mathtt{r}_\mathtt{i}]$.

The algorithm defines a single phase of the local max algorithm. Each step of the phase takes at most $\mathcal{O}(|\mathtt{A}|) = \mathcal{O}(m)$ work and $\mathcal{O}(\log |\mathtt{A}|) = \mathcal{O}(\log m) = \mathcal{O}(\log n)$ time in the CREW PRAM model. Over $\mathcal{O}(\log m)$ phases, each with geometrically decreasing number of edges, the local max algorithm takes overall $\mathcal{O}\big(\log^2 n\big)$ time and $\mathcal{O}(m)$ work in the CREW PRAM model.

9.4. Implementation and Experimental Study

We now report experiments focusing on computing approximate maximum weight matchings. We consider the following families of inputs, where the first two classes allow comparison with the experiments from [103].

Delaunay Instances are created by randomly choosing $n = 2^x$ points in the unit square and computing their Delaunay triangulation. Edge weights are Euclidean distances.

Random graphs with $n := 2^x$ nodes, αn edges for $\alpha = \{4, 16, 64\}$, and random edge weight chosen uniformly from $[0, 1]$.

Random geometric graphs with 2^x nodes (rgg$x$). Each vertex is a random point in the unit square and edges connect vertices whose Euclidean distance is below $0.55 \ln n/n$. This threshold was chosen in order to ensure that the graph is almost connected.

Florida Sparse Matrix. Following [13] we use 126 symmetric non-0/1 matrices from [39] using absolute values of their entries as edge weights, see Table A.1 in Appendix A for the full list. The number of edges of the resulting graphs $m \in (0.5 \ldots 16) \times 10^6$.

We compare implementations of local max, the red-blue algorithm from [13] (RBM) (their implementation), heavy edge matching (HEM) [80], greedy, the global path algorithm (GPA) [103], and Filter-GPA described in Algorithm 9. HEM iterates through the nodes (optionally in random order) and matches the heaviest incident edge that is nonadjacent to a previously matched edge. The greedy algorithm sorts the edges by decreasing weights, scans them and inserts edges connecting unmatched nodes into the matching.

All GPA algorithms use gcc sort routine for sorting edges. All other Algorithms involving sorting use standard STL Visual Studio 2010 sort routine.

Sequential and shared-memory parallel experiments were performed on a **platform C** with a commodity GPU with the **configuration D**, see Section 3.2. Since our GPA implementation include parallel sorting we compiled all GPA implementation in 64-bit Ubuntu Linux 10.04 using gcc 4.7 in parallel mode with the maximum optimization level. We compiled all other implementations using CUDA 4.2 and Microsoft Visual Studio 2010 on 64-bit Windows 7 Enterprise with maximum optimization level.

9.4.1. Sequential Speed and Quality

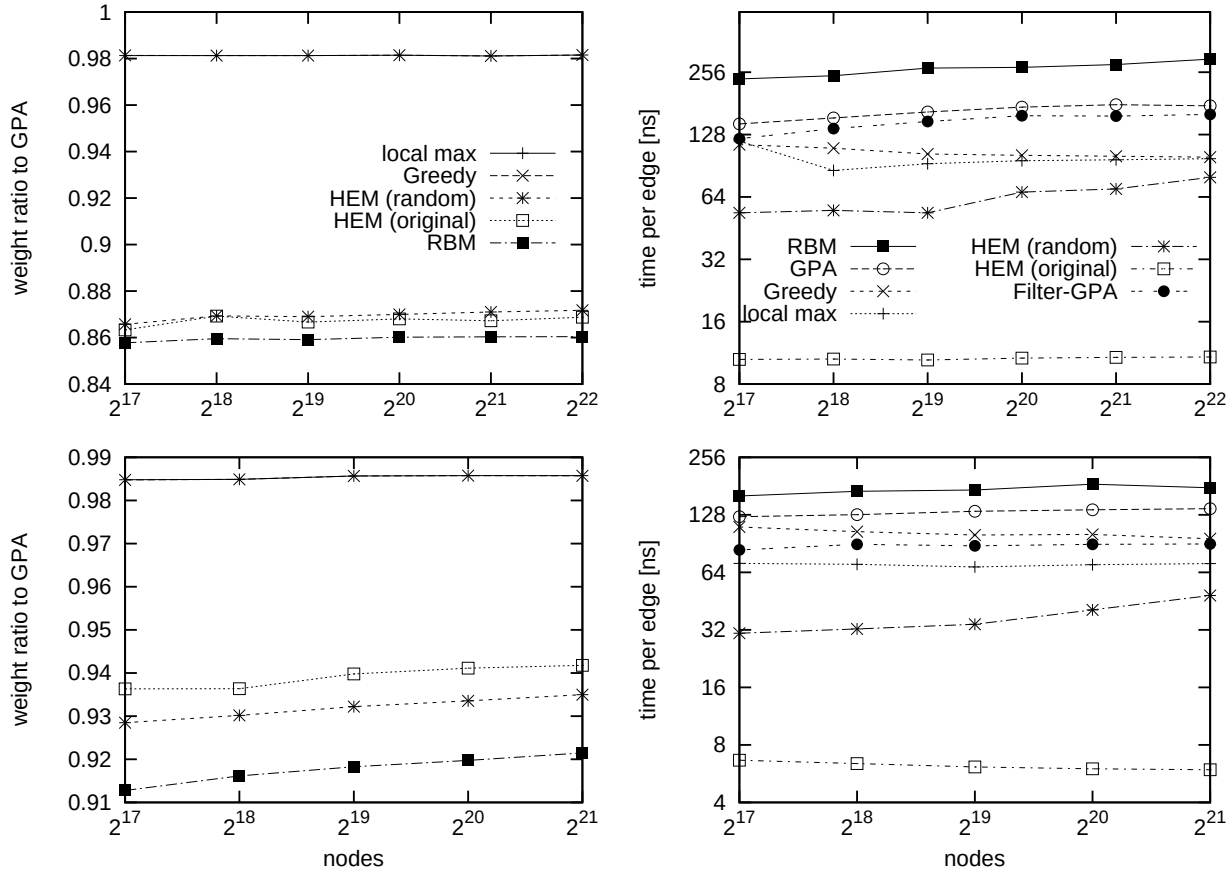


Figure 9.1.: Ratio of the weights computed by GPA and other algorithms (left) for Delaunay (top), random geometric graph (bottom) instances and the running times (right).

We compare solution quality of the algorithms relative to GPA. Via the experiments in [103] this also allows some comparison with optimal solutions, which are only a few percent better there. Figure 9.1 (top) shows the quality for Delaunay graphs (where GPA is about 5 % from optimal [103]). We see that local max achieves almost the same quality as greedy which is only about 2 % worse than GPA. HEM, possibly the fastest nontrivial sequential algorithm is about 13 % away while RBM is 14 % worse than GPA, i.e., HEM and RBM almost double the gap to optimality of local max. Looking at the running times, we see that HEM is the fastest (with a surprisingly large cost for actually randomizing node orders) followed by local max, greedy, Filter-GPA, GPA, and RBM. From this it looks like HEM, local max, and Filter-GPA are the winners in the sense that none of them is dominated by another algorithm with respect to both quality and running time. Greedy has similar quality as local max but takes somewhat longer and is not so easy to parallelize. RBM as a sequential algorithm is dominated by all other algorithms. Perhaps the most surprising thing is that RBM is fairly slow. This has to be taken into account when evaluating reported

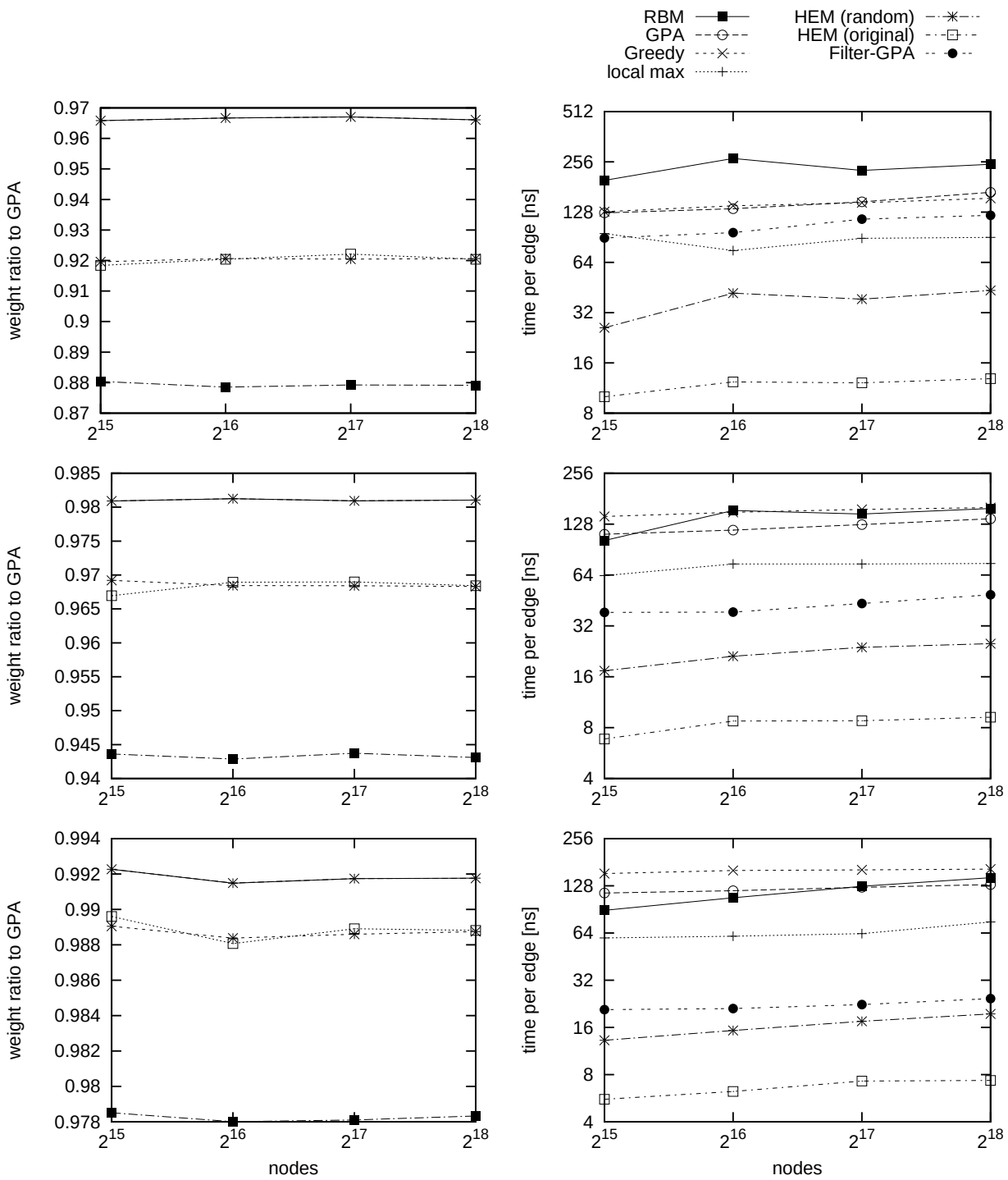


Figure 9.2.: Ratio of the weights computed by GPA and other algorithms (left) for Random Graph instances instances with $m = 4n$ (top), $m = 16n$ (middle), $m = 64n$ (bottom) and the running times (right).

speedups. We suspect that a more efficient implementation is possible but do not expect that this changes the overall conclusion.

Almost the same behaviour we observe for random geometric graphs Figure 9.1 (bottom), though our Filter-GPA already outperforms greedy. The denser the graph is, the more efficient is the Filter-GPA heuristic. Indeed, for random graphs with $m = 16n$ Filter-GPA outperforms local max already. While for $m = 64n$ it is more than factor two faster than the local max and factor 4 than the original GPA algorithm, see Figure 9.2

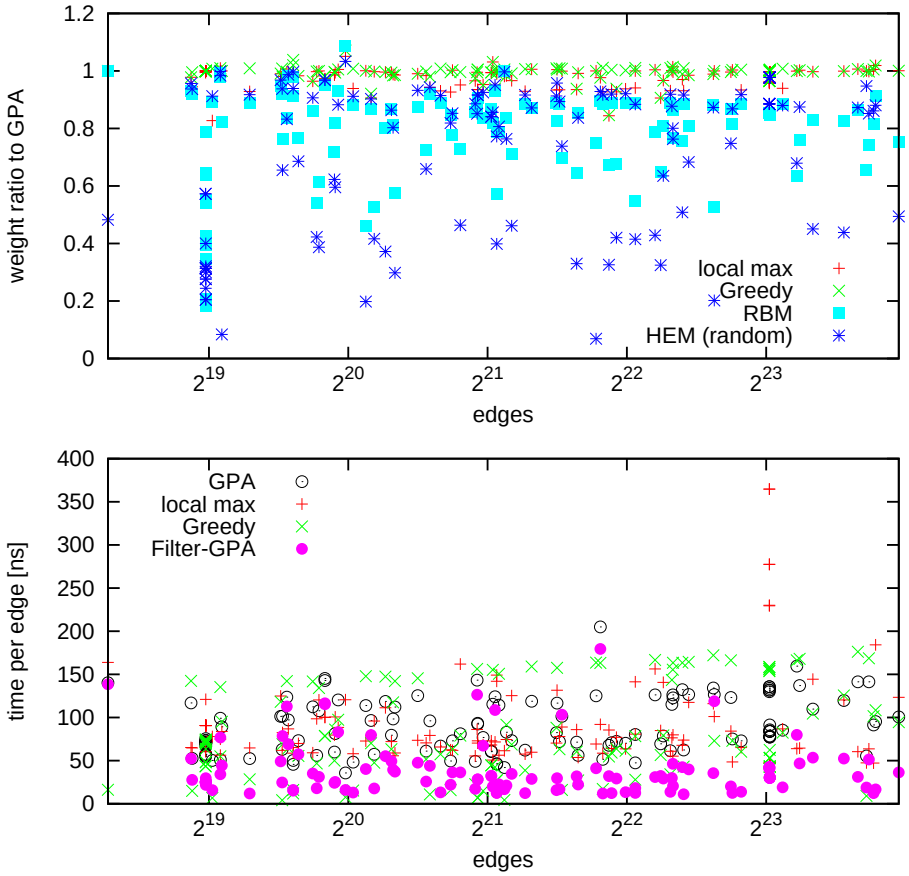


Figure 9.3.: Ratio of the weights computed by GPA and other sequential algorithms for sparse matrix instances and running time.

Looking at the wide range of instances in the Florida Sparse Matrix collection leads to similar but more complicated conclusions. Figure 9.3 shows the solution qualities for greedy, local max, RBM and HEM relative to GPA. RBM and even more so HEM shows erratic behavior with respect to solution quality. Greedy and local max are again very close to GPA and even closer to each other although there is a sizable minority of instances where greedy is somewhat better than local max. Looking at the corresponding running times one gets a surprisingly diverse picture. HEM which is again fastest and RBM which is again dominated by local max are not shown. There are instances where local max is considerably faster then greedy and vice versa. A possible explanation is that greedy becomes quite fast

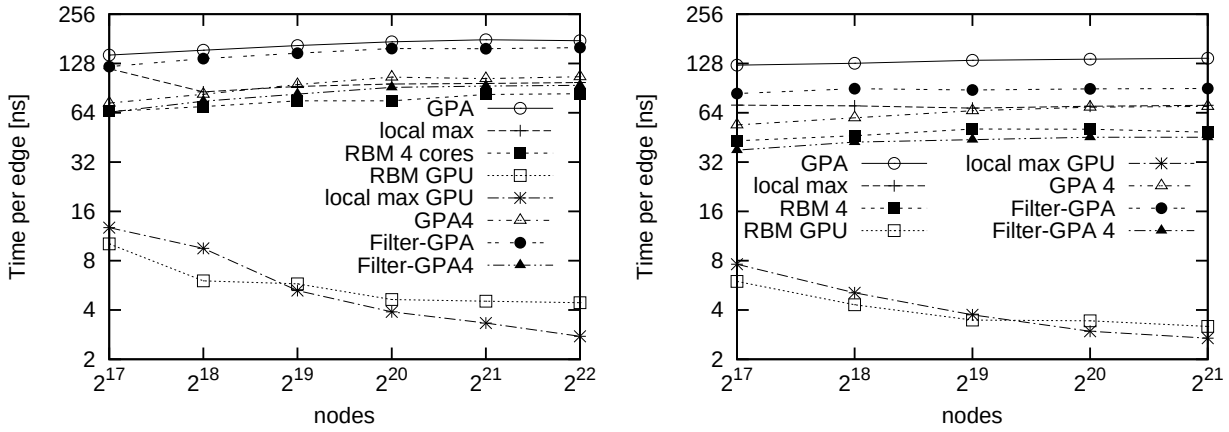


Figure 9.4.: Time per edge of sequential, multicore and GPU algorithms for Delaunay(left) and random geometric graph (right) instances.

when there is only a small number of different edge weights since then sorting is quite an easy problem. Filter-GPA performance on the other hand seems to be the most stable. It often outperforms all algorithms. The reason is that if there are small number of different edges, sorting involved in Filter-GPA is fast similar to greedy. If there are many different edges, filtering helps to avoid extensive resorting. In some sense it combines the best of both worlds. The only case, when Filter-GPA performs worse is when the graph is sparse making filtering not effective.

9.4.2. GPU Implementation

Our GPU algorithm is a fairly direct implementation of the CRCW algorithm. We reduce the algorithm to the basic primitives such as segmented prefix sum, prefix sum and random gather/scatter from/to GPU memory. As a basis for our implementation we use back40computing library by Merrill [106].

Figure 9.4 compares the running time of sequential implementations of GPA, Filter-GPA, local max, RBM, as well as multicore versions of GPA, Filter-GPA, and RBP algorithm parallelized for 4 cores. We also include GPU parallelization from [13] and our GPU implementation of local max. While the RBM CPU multicore implementation has troubles recovering from its sequential inefficiency and is only slightly faster than even sequential local max, the GPU implementation is impressively fast in particular for small graphs. For large graphs, the GPU implementation of local max is faster. Since local max has better solution quality, we consider this a good result. For Delaunay graphs our GPU code is up to 35 times faster than sequential local max. The results for rgg are slightly worse for GPU local max – speedup is up to 24 over sequential local max and a speed advantage over GPU RBM only for the very largest inputs. The Filter-GPA algorithm seems to be the fastest multicore implementation and has the best quality for all graphs besides sparse Delaunay instances. We may also be able to learn from the implementation techniques of RBM GPU

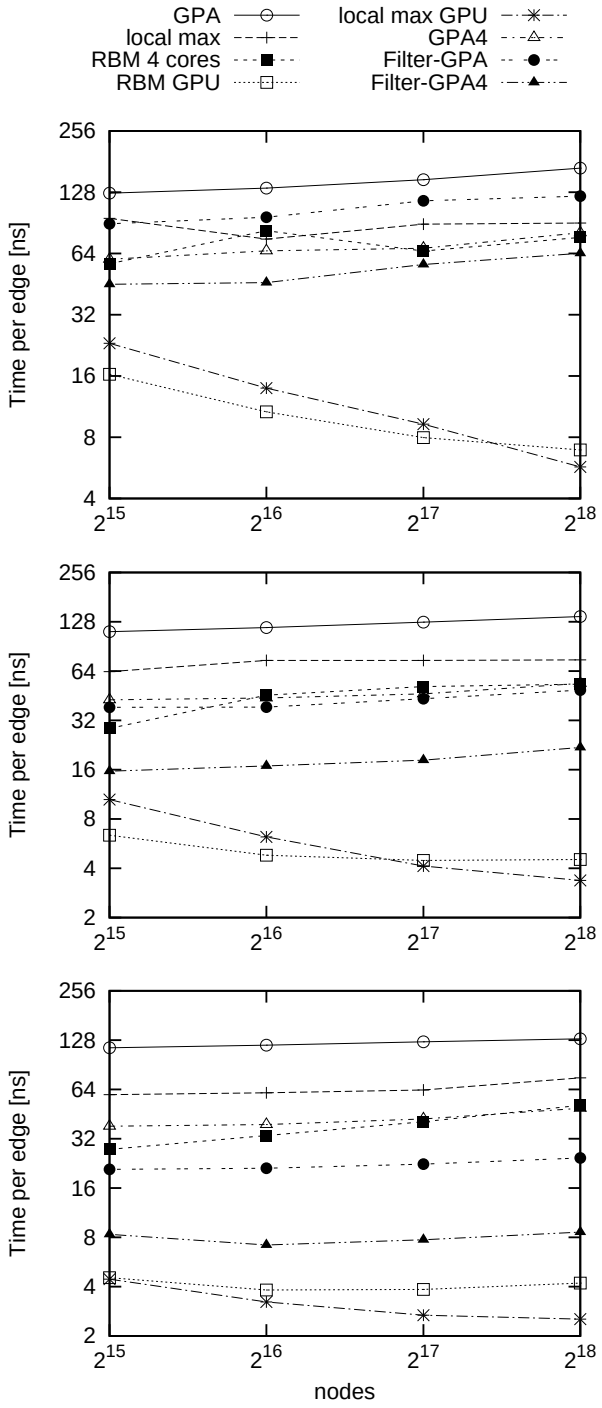


Figure 9.5.: Time per edge of sequential, multicore and GPU algorithms for random graph instances with $m = 4n$ (top), $m = 16n$ (middle) and $m = 64n$ (bottom)

for small inputs in future work.

For random graphs, we get similar behavior. The denser the graph the larger is our speedup

over the sequential and GPU RBM implementations. for $\alpha = 64$ our implementation is faster than GPU RBM already for $n = 2^{15}$. For $n = 2^{18}$ it is 65% faster than GPU RBM and 30 times faster than the sequential local max.

Our Filter-GPA implementation is also the most efficient for $\alpha = 64$. Thus, the sequential Filter-GPA already outperforms multicore RBM. While the parallel one, is less than a factor 4 slower than our GPU implementation.

9.5. Conclusions And Future Work

The local max algorithm is a good choice for massively parallel or external computation of maximal and approximate maximum weight matchings. On the theoretical side it is provably efficient for computing maximal matchings and guarantees a 1/2-approximation. On the practical side it yields better quality at faster speed than several competitors including the greedy algorithm and RBM. The Filter-GPA algorithm currently seems to be the best algorithm for sequential and multicore machines. It delivers the best quality and performs very well for not too sparse graphs.

CHAPTER 10

Graph Partitioning

Many important applications of computer science involve processing large graphs, e.g., stemming from finite element methods, digital circuit design, route planning, social networks, etc. Very often these graphs need to be partitioned or clustered such that there are few edges between the blocks (pieces).

Consider an undirected graph $G = (V, E, c, \omega)$ with edge weights $\omega : E \to \mathbf{R}_{>0}$, node weights $c : V \to \mathbf{R}_{\geq 0}$, $n = |V|$, and $m = |E|$. We extend c and ω to sets, i.e., $c(V') := \sum_{v \in V'} c(v)$ and $\omega(E') := \sum_{e \in E'} \omega(e)$. $\Gamma(v) := \{u : \{v, u\} \in E\}$ denotes the neighbors of v.

We are looking for *blocks* of nodes $V_1, \ldots, V_k$ that partition V, i.e., $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. The *balancing constraint* demands that $\forall i \in 1..k : c(V_i) \leq L_{\max} := (1 + \epsilon)c(V)/k + \max_{v \in V} c(v)$ for some parameter ϵ. The last term in this equation arises because each node is atomic and therefore a deviation of the heaviest node has to be allowed. The objective is to minimize the total *cut* $\sum_{i<j} w(E_{ij})$ where $E_{ij} := \{\{u, v\} \in E : u \in V_i, v \in V_j\}$.

References. The contents of this chapter is based on the joint work with Peter Sanders [126]. Most of the wording of the original publication is preserved.

10.1. Multilevel Graph Partitioning

A successful heuristic for partitioning large graphs is the *multilevel graph partitioning* approach (MGP) depicted in Figure 10.1 where the graph is recursively *contracted* to a smaller graph with the same basic structure.

After applying an *initial partitioning* algorithm to this small graph, the contraction is undone and, at each level, a *local refinement* method improves the partition induced by the coarser level.

By default, our initial inputs will have unit edge and node weights. However, even those will be translated into weighted problems in the course of the algorithm.

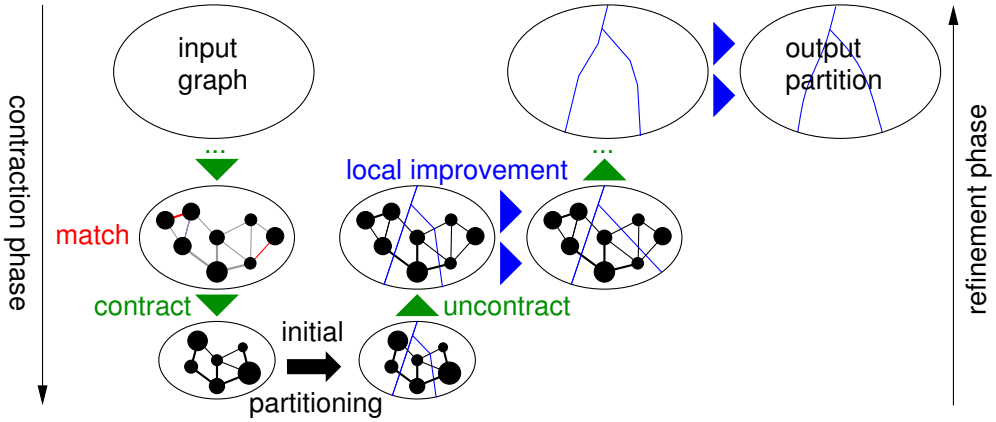


Figure 10.1.: Multilevel graph partitioning.

Contracting an edge $\{u, v\}$ simply means replacing the nodes u and v by a new node x connected to the former neighbors of u and v. We set $c(x) = c(u) + c(v)$. If replacing edges of the form $\{u, w\}, \{v, w\}$ would generate two parallel edges $\{x, w\}$, we insert a single edge with $\omega(\{x, w\}) = \omega(\{u, w\}) + \omega(\{v, w\})$.

Uncontracting an edge e undoes its contraction. Partitions computed for the contracted graph are extrapolated to the uncontracted graph in the obvious way, i.e., u and v are put into the same block as x.

Local Refinement is done by moving single nodes between blocks. The gain $g_B(v)$ of moving node v to block B is decrease in total cut size caused by this move. For example, if v has 5 incident edges of unit weight, 2 of which are inside v's block and 3 of which lead to block b then $g_B(v) = 3 - 2 = 1$.

Most systems instantiate MGP in a very similar way: Maximal matchings are contracted between two levels that try to include as many heavy edges as possible. Local refinement uses a linear time variant of local search. MGP has two crucial advantages over most other approaches to graph partitioning: We get near linear execution time since the graph shrinks geometrically and we get good partitioning quality since a good solution on some level yields a good initial solution on the next finer level, i.e., local search needs little work to further improve the solution.

Our central idea is to get even better partitions by making subsequent levels as similar as possible – we (un)contract only a *single* edge between two levels. We call this n-GP since we have (almost) n levels of hierarchy. More details are described in Section 10.2. n-GP has the additional advantage that there is no longer a need for an algorithm finding heavy matchings. This is remarkable insofar as a considerable amount of work on approximate maximum weight matching was motivated by the MGP application [45, 103, 132, 133]. Still, at first glance, n-GP seems to have substantial disadvantages also. Firstly, storing each level explicitly would lead to quadratic space consumption. We avoid this by using a dynamic graph data structure with little space overhead. Secondly, choosing maximal matchings instead of just a single edge for contraction has the side effect that the graph is contracted everywhere, leading to a more uniform distribution of node weights. We solve this problem by explicitly factoring node weights into the *edge rating* function prioritizing the edges to

be contracted. Already in [67, 68] edge ratings have proven to lead to better results for graph partitioning. Perhaps the most serious problem is that the most common approach to local search is to let it run for a number of steps proportional to the current number of nodes. In the context of n-GP this could lead to a quadratic overall number of local search steps. Therefore, we develop a new, more adaptive stopping criteria for the local search that drastically accelerates n-GP without significantly reducing partitioning quality.

We have implemented n-GP in the graph partitioner KaSPar (Karlsruhe Sequential Partitioner). Experiments reported in Subsection 10.2.3 indicate that KaSPar scales well to large networks, computes the best known partitions for many instances of a "standard benchmark" and needs time comparable to system that previously computed the best results for large networks. Section 10.3 summarizes the results and discusses future directions.

Related Work

There has been a huge amount of research on graph partitioning so that we refer to introductory and overview papers such as [50, 80, 150, 171] for more material. Well-known software packages based on MGP are Chaco [65], DiBaP [113], Jostle [170, 171], Metis [81, 82], Party [134, 135], and Scotch [129, 130].

KaSPar was developed partly in parallel with KaPPa (Karlsruhe Parallel Partitioner) [68]. KaPPa is a "classical" matching based MGP algorithm designed for scalable parallel execution and its local search only considers independent pairs of blocks at a time. Still, for $k = 2$, its interesting to compare KaSPar and KaPPa since KaPPa achieves the previously best partitioning results for many large graphs, since both systems use a similar edge ratings, and since running times for a two processor parallel code and a sequential code could be expected to be roughly comparable. Since our implementation of KaSPar Schulz and Sanders further improved their partitioner by using max-flow min-cut computations in KaFFPa (Karlsruhe Fast Flow Partitioner) [144]. They also developed its distributed evolutionary version [143] and developed a special algorithm for perfectly balanced graph partitioning problem [145].

There is a long tradition of n-level algorithms in geometric data structures based on randomized incremental construction (e.g, [24, 63]). Our motivation for studying n-level are *contraction hierarchies* [58], a preprocessing technique for route planning that is at the same time simpler and an order of magnitude more efficient than previous techniques using a small number of levels.

10.2. n-Level Graph Partitioning

Algorithm 10 gives a high-level recursive summary of n-GP. The base case is some other partitioner used when the graph is sufficiently small. In KaSPar, contraction is stopped when either only $20k$ nodes remain, no further nodes are eligible for contraction, or there are less edges than nodes left. The latter happens when the graph consists of many independent components. As observed in [68] Scotch [130] produces better initial partitions than metis, and therefore we also use it in KaSPar .

Algorithm 10: n-GP

```
1  n-GP(G, k, ε)
2  begin
3      if G is small then
4          return initialPartition(G, k, ε)
5      pick the edge e = {u, v} with the highest rating
6      contract e
7      P := n-GP(G, k, ε)
8      uncontract e
9      activate(u), activate(v), localSearch()
10     return P
11 end
```

The edges to be contracted are chosen according to an edge rating function. KaSPar adopts the rating function

$$\text{expansion}^*(\{u,v\}) := \frac{\omega(\{u,v\})}{c(u)c(v)}$$

which fared best in [68]. As a further measure to avoid unbalanced inputs to the initial partitioner, KaSPar never allows a node v to participate in a contraction if the weight of v exceeds $1.5n/(20k)$. Selecting contracted edges can be implemented efficiently by keeping the contractable *nodes* in a priority queue sorted by the rating of their most highly rated incident edge.

In order to make contraction and uncontraction efficient, we use a "semidynamic" graph data structure: When contracting an edge $\{u, v\}$, we mark both u and v as deleted, introduce a new node w, and redirect the edges incident to u and v to w. The advantage of this implementation is that edges adjacent to a node are still stored in adjacency arrays which are more efficient than linked lists needed for a full fledged dynamic graph data structure. A disadvantages of our approach is a certain space overhead. However, it is relatively easy to show that this space overhead is bounded by a logarithmic factor even if we contract edges in some random fashion (see [43]). In Subsection 10.2.3 we will demonstrate experimentally that the overhead is actually often a small constant factor. Indeed, this is not very surprising since the edge rating function is not random, but designed to keep the contracted graph sparse. Overall, with respect to asymptotic memory overhead, n-GP is no worse than methods with a logarithmic number of levels.

10.2.1. Local Search Strategy

Our local search strategy is similar to the FM-algorithm [49] that is also used in many other MGP systems. We now outline our variant and then discuss differences.

Originally, all nodes are unmarked. Only unmarked nodes are allowed to be activated or moved from one block to another. Activating a node $v \in B'$ means that for blocks

$\{B \neq B' : \exists \{v, u\} \in E \wedge u \in B\}$ we compute the gain

$$g_B(v) = \sum \{\omega(\{v, u\}) : \{v, u\} \in E, v \in B\} - \sum \{\omega(\{v, u\}) : \{v, u\} \in E, v \in B'\}$$

of moving v to block B. Node v is then inserted into the priority queue P_B using $g_B(v)$ as the priority. We call a queue P_B eligible if the highest gain node in P_b can be moved to block B without violating the balance constraint for block B. Local search repeatedly looks for the highest gain node v in any eligible priority queue P_B and moves v to block B. When this happens, node v becomes nonactive and marked, the unmarked neighbors of v get activated and the gains of the active neighbors are updated. The local search is stopped if either no eligible nonempty queues remain, or one of the stopping criteria described below applies. After the local search stops, it is rolled back to the lowest cut state reached during the search (which is the starting state if no improvement was achieved). Subsequently all previously marked nodes are unmarked. The local search is repeated until no improvement is achieved.

The main difference to the usual FM-algorithm is that our routine performs a highly localized search starting just at the uncontracted edge. Indeed, our local search does nothing if none of the uncontracted nodes is a *border node*, i.e., has a neighbor in another block. Other FM-algorithms initialize the search with all border nodes. In n-GP the local search may find an improvement quickly after moving a small number of nodes. However, in order to exploit this case, we need a way to stop the search much earlier than previous algorithms which limit the number of steps to a constant fraction of the current number of nodes $|V|$.

Stopping Using a Random Walk Model. It makes sense to make a stopping rule more adaptive by making it dependent on the past history of the search, e.g., on the difference between the current cut and the best cut achieved before.

We model the gain values in each step as identically distributed, independent random variables whose expectation μ and Variance σ^2 is obtained from the previously observed p steps. Then in the next s steps, we can expect a deviation from the expectation $(p+s)\mu$ by something of the order $\sqrt{s\sigma^2}$. The expression $(p+s)\mu + \sqrt{s\sigma^2}$ is maximized for $s^* := \frac{\sigma^2}{4\mu^2}$. Now the idea is to stop when for some tuning parameter x, $(p+s^*)\mu + x\sqrt{s^*\sigma^2} > 0$, i.e., it is reasonably likely that a random walk modelling our local search can still give an improvement. This translates to the condition $p > \frac{\sigma^2}{\mu^2}(\frac{x}{2} - \frac{1}{4})$ or simply $p\mu^2 \gg \sigma^2$.

Thus, we can derive that it is unlikely that the local search will produce a better cut if

$$p\mu^2 > \alpha\sigma^2 + \beta \tag{10.1}$$

where α and β are tuning parameters. Parameter β is a base value that avoids stopping just after a small constant number of steps that happen to have small variance. Currently we set it to $\ln n$.

10.2.2. Trial Trees

It is a standard technique in optimization heuristics to improve results by repeating various parts of the algorithm. We generalize several approaches used in MGP by adapting an idea

initially used in a fast randomized min-cut algorithm [74]: After reducing the number of nodes by a factor c, we perform two independent trials using different random seeds for tie breaking during contraction, initial partitioning, and local search. Among these trials the one with the smaller cut is used for continuing upwards. This way, we perform independent trials at many levels of contraction controlled by a single tuning parameter c. As long as $c > 2$, the total number of contraction steps performed stays $\mathcal{O}(n)$.

10.2.3. Experimental Study

Implementation. We implemented KaSPar in C++ using gcc-4.3.2. We use priority queues based on paring heaps [164] available in the policy-based elementary data structures library (pb_ds) for implementing contraction and refinement procedures. In the following experimental study we compared KaSPar to Scotch 5.1, kMetis 4.0 and the same version of KaPPa as in [68].

System. We performed our experiments on a single core the **platform E**, see Section 3.2 running Suse Linux Enterprise 10.

Instances. We report results on two suites of instances summarized in Table 10.1. rggX is a *random geometric graph* with 2^X nodes that represent random points in the unit square and edges connect nodes whose Euclidean distance is below $0.55\sqrt{\ln n/n}$. This threshold was chosen in order to ensure that the graph is almost connected. DelaunayX is the Delaunay triangulation of 2^X random points in the unit square. Graphs bcsstk29..fetooth and ferotor..auto come from Chris Walshaw's benchmark archive [161]. Graphs bel, nld, deu and eur are undirected versions of the road networks of Belgium, the Netherlands, Germany, and Western Europe respectively, used in [40]. Instances af_shell9 and af_shell10 come from the Florida Sparse Matrix Collection [39]. coAuthorsDBLP, coPapersDBLP, citationCiteseer, coAuthorsCiteseer and cnr2000 are examples of social networks taken from [57].

For the number of partitions k we choose the values used in [161]: 2, 4, 8, 16, 32, 64. Our default value for the allowed imbalance is 3 % since this is one of the values used in [161] and the default value in Metis.

When not otherwise mentioned, we perform 10 repetitions for the small networks and 5 repetitions for the other. We report the arithmetic average of computed cut size, running time and the best cut found. When further averaging over multiple instances, we use the geometric mean in order to give every instance the same influence on the final figure.

Configuring the Algorithm. We use two sets of parameter settings *fast* and *strong*. These methods only differ in the constant factor α in the local search stopping rule, see Equation (10.1) in the contraction factor c for the trial tree (Subsection 10.2.2), and in the number of initial partitioning attempts a performed at the coarsest level of contraction:

strategy	α	c	a
fast	1	8	$25/\log_2 k$
strong	4	2.5	$100/\log_2 k$

Table 10.1.: Basic properties of the graphs from our benchmark set. The large instances are split into five groups: geometric graphs, FEM graphs, street networks, sparse matrices, and social networks. Within their groups, the graphs are sorted by size.

Medium sized instances		
graph	n	m
rgg17	2^{17}	1 457 506
rgg18	2^{18}	3 094 566
Delaunay17	2^{17}	786 352
Delaunay18	2^{18}	1 572 792
bcsstk29	13 992	605 496
4elt	15 606	91 756
fesphere	16 386	98 304
cti	16 840	96 464
memplus	17 758	108 384
cs4	33 499	87 716
pwt	36 519	289 588
bcsstk32	44 609	1 970 092
body	45 087	327 468
t60k	60 005	178 880
wing	62 032	243 088
finan512	74 752	522 240
ferotor	99 617	662 431
bel	463 514	1 183 764
nld	893 041	2 279 080
af_shell9	504 855	17 084 020

Large instances		
graph	n	m
rgg20	2^{20}	13 783 240
Delaunay20	2^{20}	12 582 744
fetooth	78 136	905 182
598a	110 971	1 483 868
ocean	143 437	819 186
144	144 649	2 148 786
wave	156 317	2 118 662
m14b	214 765	3 358 036
auto	448 695	6 629 222
deu	4 378 446	10 967 174
eur	18 029 721	44 435 372
af_shell10	1 508 065	51 164 260
Social networks		
coAuthorCiteseer	227 320	1 628268
coAutorhDBLP	299 067	1 955 352
cnr2000	325 557	3 216 152
citationCiteseer	434 102	32 073 440
coPaperDBLP	540 486	30 491 458

Note that this are considerably less parameters compared to KaPPa. In particular, there is no need for selecting a matching algorithm, an edge coloring algorithm, or global and local iterations for refinement.

Scalability. Figure 10.2 shows the number of edges touched during contraction (KaSPar strong, small and large instances). We see that this scales linearly with the number of input edges and with a fairly small constant factor between 2 and 3. Interestingly, the number of local search steps during local improvement (Figure 10.3) *decreases* with increasing input size. This can be explained by the sublinear number of border vertices that we have in graphs that have small cuts and by small average search space sizes for the local search. Indeed, Figure 10.4 indicates that the average length of local searches grows only logarithmically with n. All this translates into fairly complicated running time behavior. Still, Figure 10.5 warrants the conclusion that running time scales "near linearly" with the input size.[1] The

[1]This may not apply to the social networks which have considerably worse behavior.

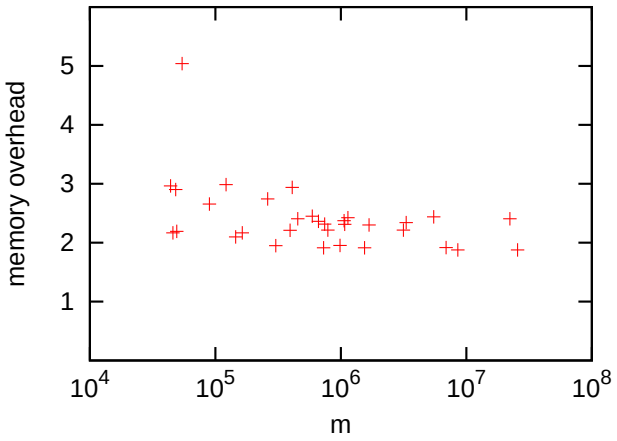


Figure 10.2.: Number of edges created during contraction.

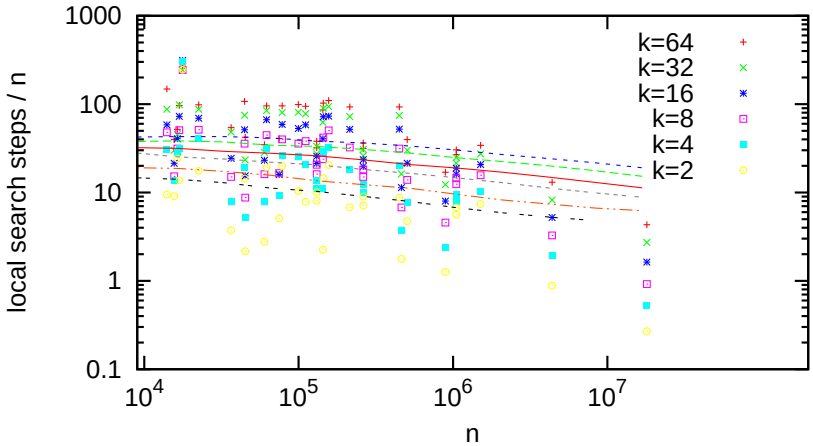


Figure 10.3.: Total number of local search steps. The nearly straight lines represent series for the graphs rgg15..rgg24 and Delaunay15..Delaunay24 for different k.

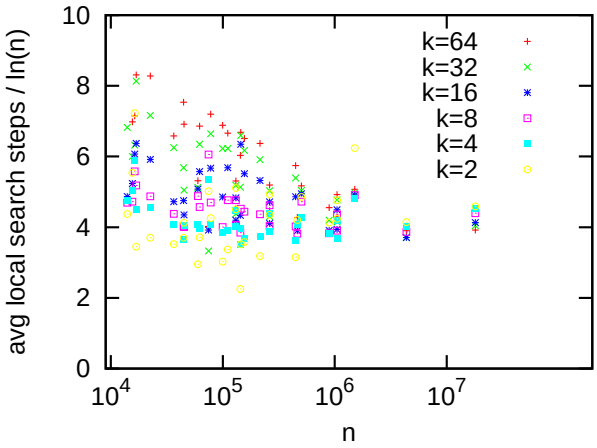


Figure 10.4.: Average length of local searches.

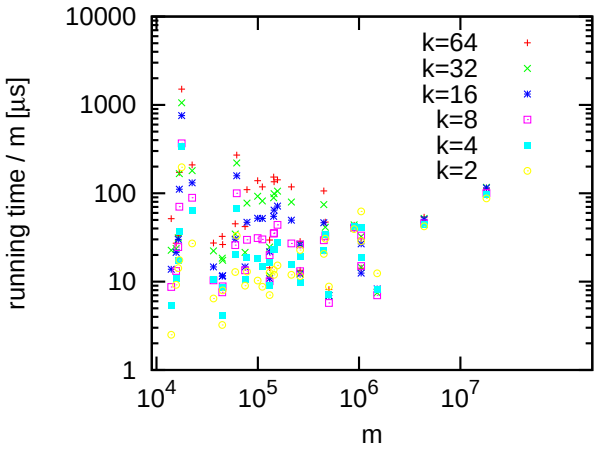


Figure 10.5.: Running Time.

term in the running time depending on k grows sublinearly with the input size so that for very large graphs the number of blocks does not matter much.

Does the Random Walk Model Work? We have compared KaSPar fast with a variant where the stopping rule is disabled (i.e., $\alpha = \infty$). For the small instances this yields about 1 % better cut sizes at the cost of an order of magnitude larger running time. This is a small improvement both compared to the improvement KaSPar achieves over other systems and compared to just repeating KaSPar fast 10 times (see Table 10.2).

Do Trial trees help? We use the following evaluation: We run KaSPar strong and measure its elapsed time. Then for different values of initial partitionings a we repeat KaSPar strong without trial trees($c = 0$), until the sum of the run times of all repetitions exceeds the run time of KaSPar strong. Than for different values a we compare the best edge cut achieved during repeated runs to the one produced by KaSPar strong. Finally, we average the obtained results over 5 repetitions of this procedure. If we then quality the computed partitions, we

usually get almost identical results (a fraction of a percent difference). However, most of the time trial trees are a bit better and for *road networks* we get considerable improvements. For example, for the European network we get an improvement of 10 % on average over all k.

Comparison with other Systems. Table 10.2 summarizes the results by computing geometric means over 10 runs for the small instances and over 5 runs for the large instances and social networks. We exclude the European road network for $k = 2$ because KaPPa runs out of memory in that case. Detailed per instance results can be found in the Appendix A, Tables A.2 and A.3. KaPPa strong produces 5.9 % larger cuts than KasPar strong for small instances (average value) and 8.1 % larger cuts for the large instances. This comparison might seem a bit unfair because KaPPa is about five times faster. However, KaPPa is using k processors in parallel. Indeed, for $k = 2$ KaSPar strong needs only about twice as much time. Also note that KaPPa strong needs about twice as much time as KaSPar fast while still producing 6 % larger cuts despite running in parallel. The case $k = 2$ is also interesting because here KaPPa and KaSPar are most similar – parallelism does not play a big role (2 processors) and both local search strategies work only on two blocks at all time. Therefore 6 % improvement of KaSPar over KaPPa we can attribute mostly to the larger number of levels.

Scotch and kMetis are much faster than KaSPar but also produce considerably larger cuts – e.g., 32 % larger for large instances (kMetis, average). For the European road network, the difference in cut size even exceeds a factor of two. Such gaps usually cannot be breached by just running the faster solver a larger number of times. For example, for large instances, Scotch is only a factor around 4 faster than KaSPar fast, yet its best cut values obtained from 5 runs are still 12.7 % larger than the average values of KaSPar fast.

For social networks all systems have problems. KaSPar lags further behind in terms of speed but extends its lead with respect to the cut size. We mostly attribute the larger run time to the larger cut sizes relative to the number of nodes which greatly increase the number of local searches necessary. A further effect may be that the time for a local search step is proportional to the number of partitions adjacent to the nodes participating in the local search. For "well behaved" graphs this is mostly two, but for social networks which get denser on the coarser levels this value can get larger.

The Walshaw Benchmark [161] considers 34 graphs using $k \in \{2, 4, 8, 16, 32, 64\}$ and balance parameter $\epsilon \in \{0, 0.01, 0.03, 0.05\}$ giving a total of 816 table entries. Only cut sizes count – running time is not reported. We tried all combinations except the case $\epsilon = 0$ which KaSPar cannot handle yet. We ran KaSPar strong with a time limit of one hour and report the best result obtained in the Appendix A, Tables A.4 to A.6. KaSPar improved 155 values in the benchmark table: 42 for 1%, 49 for 3% and 64 for 5% allowed imbalance. Moreover, it reproduced equally sized cuts in 83 additional cases. If we count only results for graphs having over $44k$ nodes and $\epsilon > 0$, KaSPar improved 131 and reproduced 27 cuts, thus summing up to 63% of large graph table slots. We should note, that 51 of the new improvements are over partitioners different from KaPPa. Most of the improvements lie in

Table 10.2.: Geometric means over all instances.

code	small graphs			large graphs			social networks		
	best	avg.	t[s]	best	avg.	t[s]	best	avg.	t[s]
KaSPar strong	2 675	2 729	7.37	12 450	12 584	87.12	-	-	-
KaSPar fast	2 717	2 809	1.43	12 655	12 842	14.43	93657	99062	297.34
KaSPar fast, $\alpha = \infty$	2 697	2 780	23.21	-	-	-	-	-	-
KaPPa strong	2 807	2 890	2.10	13 323	13 600	28.16	117701	123613	78.00
KaPPa fast	2 819	2 910	1.29	13 442	13 727	16.67	117927	126914	46.40
kMetis	3 097	3 348	0.07	15 540	16 656	0.71	117959	134803	1.42
Scotch	2 926	3 065	0.48	14 475	15 074	3.83	168764	168764	17.69

Large Instances						
k	KaSPar strong			KaPPa strong		
	best	avg.	t[s]	best	avg.	t[s]
2	2 842	2 873	36.89	2 977	3 054	15.03
4	5 642	5 707	60.66	6 190	6 384	30.31
8	10 464	10 580	75.92	11 375	11 652	37.86
16	17 345	17 567	102.52	18678	19 061	39.13
32	27 416	27 707	137.08	29 156	29 562	31.35
64	41 284	41 570	170.54	43 237	43 644	22.36

the lower triangular part of the table, meaning that KaSPar is particularly good for either large graphs, or smaller graphs with small k. On the other hand, for small graphs, large k, and $\epsilon = 1\%$ KaSPar was often not able to obtain a feasible solution. A primary reason for this seems to be that initial partitioning yields highly infeasible solutions that KaSPar is not able to to improve considerably during refinement. This is not astonishing, since Scotch targets $\epsilon = 3\%$ and does not even guarantee that.

10.3. Conclusion

n-GP is a graph partitioning approach that scales to large inputs and currently computes the best known partitions for many large graphs, at least when a certain imbalance is allowed. It is in some sense simpler than previous methods since no matching algorithm is needed. Although our current implementation of KaSPar is a considerable constant factor slower than the fastest available MGP partitioners, we see potential for further tuning. In particular, thanks to our adaptive stopping rule, KaSPar needs to do very little local search, in particular for large graphs and small k. Thus it suffices to tune the relatively simple contraction routine to obtain significant improvements. On the other hand, the adaptive stopping rule might also turn out to be useful for matching based MGP algorithms.

A lot of opportunities remain to further improve KaSPar. In particular, we did not yet attempt to handle the case $\epsilon = 0$ since this may require different local search strategies. We

also want to try other initial partitioning algorithms and ways to integrate n-GP into other metaheuristics like evolutionary search.

We expect that n-GP could be generalized for other objective functions, for hypergraphs, and for graph clustering. More generally, the success of n-GP also suggests to look for more applications of the n-level paradigm.

An apparent drawback of n-GP is that it looks inherently sequential. However, we could probably obtain a good parallel algorithm by contracting *small* sets of highly rated, independent edges in parallel. Indeed, in the light of our results for KaSPar the complications coming from the need to find maximal matchings of heavy edges seem unnecessary, i.e., a parallelization of n-GP might be fast and simple.

Acknowledgements. We would like to thank Christian Schulz for supplying data for KaPPa, Scotch and Metis.

CHAPTER 11

Discussion

In this thesis we showed that many classical algorithmic problems need to be reconsidered under new challenging circumstances, that is, for instance, massive data or growing parallelism. The advances even in classical well studied problems are possible and highly required with the development of new hardware and ever growing data. We showed that following "algorithm engineering" methodology we were able to design algorithms, that are good not only in theory but demonstrate excellent behaviour in practice also.

Trying to solve the problem, we first chose the appropriate machine model having right objective function for the problem and data size at hand. Having fixed the model, we designed and analyzed the algorithm taking as asymptotic behaviour as constant factors into account. We devoted considerable time to implementations and optimizations to achieve the best performance. We used as synthetical and worst-case as real-world instances for evaluation of our implementations. We believe that in order to achieve the best performance one should look at the problem at very detail rather than pursue generality. Typical inputs may also often give algorithmic insights and lead to even larger performance boost.

Having used this general approach we developed algorithms with the world leading performance at the time of the writing (in time or/and quality). We worked as on basic algorithms as sorting (sample sort and suffix array construction) as more advanced graph algorithms (breadth first search, minimum spanning tree, single-source shortest paths, matchings, graph partitioning). Our main focus was parallelism (sample sort, suffix array, matching, minimum spanning tree) and massive data (suffix array, breadth first search, single-source shortest paths). We also proposed heuristics that allowed us to show better expected runtime bounds (minimum spanning tree, matchings) as well as to tackle NP hard problem (graph partitioning).

Bibliography

[1] James Abello, Adam L. Buchsbaum, and Jeffery R. Westbrook. "A Functional Approach to External Graph Algorithms". In: *Algorithmica* 32.3 (2002), pp. 437–458.

[2] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. "Replacing Suffix Trees with Enhanced Suffix Arrays". In: *Journal of Discrete Algorithms* 2.1 (2004), pp. 53–86.

[3] Alok Aggarwal and Jeffrey S. Vitter. "The Input/Output Complexity of Sorting and Related Problems". In: *Communications of the ACM* 31.9 (1988), pp. 1116–1127.

[4] Deepak Ajwani. "Traversing Large Graphs in Realistic Settings". PhD thesis. Universität des Saarlandes, 2008.

[5] Deepak Ajwani, Roman Dementiev, and Ulrich Meyer. "A Computational Study of External-Memory BFS Algorithms". In: *Proceedings of the 17th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'06)*. SIAM, 2006, pp. 601–610.

[6] Deepak Ajwani, Ulrich Meyer, and Vitaly Osipov. "Improved External Memory BFS Implementation". In: *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*. SIAM, 2007.

[7] Deepak Ajwani, Itay Malinger, Ulrich Meyer, and Sivan Toledo. "Characterizing the Performance of Flash Memory Storage Devices and Its Impact on Algorithm Design". In: *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*. Vol. 5038. Lecture Notes in Computer Science. Springer, 2008, pp. 208–219.

[8] Deepak Ajwani, Andreas Beckmann, Riko Jacob, Ulrich Meyer, and Gabriel Moruz. "On Computational Models for Flash Memory Devices". In: *Proceedings of the 8th International Symposium on Experimental Algorithms (SEA'09)*. Vol. 5526. Lecture Notes in Computer Science. Springer, 2009, pp. 16–27.

[9] Lars Arge. "The Buffer Tree: A Technique for Designing Batched External Data Structures". In: *Algorithmica* 37.1 (2003), pp. 1–24.

[10] Lars Arge, Gerth Stølting Brodal, and Laura Toma. "On External-Memory MST, SSSP and Multi-Way Planar Graph Separation". In: *Algorithms* 53.2 (2004), pp. 186–206.

[11] Lars Arge, Laura Toma, and Jeffrey S. Vitter. "I/O-efficient Algorithms for Problems on Grid-based Terrains". In: *ACM Journal of Experimental Algorithmics* 6 (2001), pp. 1–20.

[12] Lars Arge, Paolo Ferragina, Roberto Grossi, and Jeffrey S. Vitter. "On Sorting Strings in External Memory". In: *Proceedings of the 29th Annual ACM Symposium on the Theory of Computing (STOC'97)*. ACM Press, 1997, pp. 540–548.

[13] Bas O. Fagginger Auer and Rob H. Bisseling. "A GPU Algorithm for Greedy Graph Matching". In: *Facing the Multicore-Challenge II*. Vol. 7174. Lecture Notes in Computer Science. Springer, 2012, pp. 108–119.

[14] David Avis. "A Survey of Heuristics for the Weighted Matching Problem". In: *Networks* 13.4 (1983), pp. 475–493.

[15] David A. Bader and Guojing Cong. "A Fast, Parallel Spanning Tree Algorithm for Symmetric Multiprocessors (SMPs)". In: *Journal of Parallel and Distributed Computing* 65.9 (2005), pp. 994–1006.

[16] Jiří Barnat, Luboš Brim, Stefan Edelkamp, Damian Sulewski, and Pavel Šimeček. "Can Flash Memory Help in Model Checking?" In: *Proceedings of the 13th Int'l Workshop on Formal Methods for Industrial Critical Systems*. Vol. 5596. Lecture Notes in Computer Science. Springer, 2008, pp. 159–174.

[17] Marina Barsky, Ulrike Stege, and Alex Thomo. "A Survey of Practical Algorithms for Suffix Tree Construction in External Memory". In: *Software – Practice and Experience* 40.11 (2010), pp. 965–988.

[18] Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. "In Transit to Constant Time Shortest-Path Queries in Road Networks". In: *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*. SIAM, 2007.

[19] Kenneth E. Batcher. "Sorting Networks and Their Applications". In: *Proceedings of AFIPS Spring Joint Computing Conference*. Vol. 32. ACM Press, 1968, pp. 307–314.

[20] Andreas Beckmann, Roman Dementiev, and Johannes Singler. "Building a Parallel Pipelined External Memory Algorithm Library". In: *Proceedings of the 23rd Int'l Parallel and Distributed Processing Symposium (IPDPS'09)*. IEEE Computer Society Press, 2009, pp. 1 –10.

[21] Jon L. Bentley and Robert Sedgewick. "Fast Algorithms for Sorting and Searching Strings". In: *Proceedings of the 8th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'97)*. SIAM, 1997, pp. 360–369.

[22] Timo Bingmann, Johannes Fischer, and Vitaly Osipov. "Inducing Suffix and LCP Arrays in External Memory". In: *Proceedings of the 15th Workshop on Algorithm Engineering and Experiments (ALENEX'13)*. SIAM, 2013.

[23] Marcel Birn, Vitaly Osipov, Peter Sanders, Christian Schulz, and Nodari Sitchinava. "Efficient Parallel and External Matching". In: *Proceedings of the Int'l Conference on Parallel Processing (Euro-Par'13)*. Lecture Notes in Computer Science. to appear. Springer, 2013.

[24] Marcel Birn, Manuel Holtgrewe, Peter Sanders, and Johannes Singler. "Simple and Fast Nearest Neighbor Search". In: *Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX'10)*. SIAM, 2010.

[25] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. "An Experimental Analysis of a Compact Graph Representation". In: *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04)*. SIAM, 2004, pp. 49–61.

[26] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. "Compact Representation of Separable Graphs". In: *Proceedings of the 14th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'03)*. SIAM, 2003, pp. 679–688.

[27] Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. "Greedy Sequential Maximal Independent Set and Matching are Parallel on Average". In: *Proceedings of the 24th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'12)*. ACM Press, 2012, pp. 308–317.

[28] Gerth Stølting Brodal and Rolf Fagerberg. "Cache Oblivious Distribution Sweeping". In: *Proceedings of the 28th International Colloquium on Automata, Languages, and Programming (ICALP'02)*. Vol. 1470. Lecture Notes in Computer Science. Springer, 2002, pp. 426–438.

[29] Gerth Stølting Brodal, Rolf Fagerberg, and Kristoffer Vinther. "Engineering a Cache-oblivious Sorting Algorithm". In: *ACM Journal of Experimental Algorithmics* 12 (2008).

[30] Gerth Stølting Brodal, Rolf Fagerberg, Ulrich Meyer, and Norbert Zeh. "Cache-oblivious Data Structures and Algorithms for Undirected Breadth-First Search and Shortest Paths". In: *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory (SWAT'04)*. Vol. 3111. Lecture Notes in Computer Science. Springer, 2004, pp. 480–492.

[31] Adam L. Buchsbaum, Michael Goldwasser, Suresh Venkatasubramanian, and Jeffery R. Westbrook. "On External Memory Graph Traversal". In: *Proceedings of the 14th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'03)*. SIAM, 2003, pp. 859–860.

[32] Daniel Cederman and Philippas Tsigas. "A Practical Quicksort Algorithm for Graphics Processors". In: *Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08)*. Vol. 5193. Lecture Notes in Computer Science. Springer, 2008, pp. 246–258.

[33] Timothy M. Chan. "Backwards Analysis of the Karger-Klein-Tarjan Algorithm for Minimum Spanning Trees". In: *Information Processing Letters* 67.6 (1998), pp. 303–304.

[34] Mo Chen, Rezaul Alam Chowdhury, Vijaya Ramachandran, David Lan Roche, and Lingling Tong. *Priority Queues and Dijkstra's Algorithm*. Tech. rep. TR-07-54. The University of Texas at Austin, Department of Computer Sciences, 2007.

[35] Shifu Chen, Jing Qin, Yongming Xie, Junping Zhao, and Pheng-Ann Heng. "A Fast and Flexible Sorting Algorithm with CUDA". In: *Proceedings of the 9th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'09)*. Vol. 5574. Lecture Notes in Computer Science. 2009, pp. 281–290.

[36] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey S. Vitter. "External-Memory Graph Algorithms". In: *Proceedings of the 6th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'95)*. SIAM, 1995, pp. 139–149.

[37] Frederik J. Christiani. "Cache-Oblivious Graph Algorithms". MA thesis. University of Southern Denmark, 2005.

[38] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2001.

[39] Tim Davis. *The University of Florida Sparse Matrix Collection*. http://www.cise.ufl.edu/research/sparse/matrices/.

[40] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. "Engineering Route Planning Algorithms". In: *Algorithmics of Large and Complex Networks*. Vol. 5515. Lecture Notes in Computer Science. Springer, 2009, pp. 117–139.

[41] Roman Dementiev, Lutz Kettner, and Peter Sanders. "STXXL: Standard Template Library for XXL Data Sets". In: *Software – Practice and Experience* 38.6 (2008), pp. 589–637.

[42] Roman Dementiev, Juha Kärkkäinen, Jens Mehnert, and Peter Sanders. "Better External Memory Suffix Array Construction". In: *ACM Journal of Experimental Algorithmics* 12 (2008), 3.4:1–3.4:24.

[43] Roman Dementiev, Peter Sanders, Dominik Schultes, and Jop Sibeyn. "Engineering an External Memory Minimum Spanning Tree Algorithm". In: *Exploring New Frontiers of Theoretical Informatics* 155 (2004), pp. 195–208.

[44] Edsger W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numerische Mathematik* 1 (1959), pp. 269–271.

[45] Doratha E. Drake and Stefan Hougardy. "Improved Linear Time Approximation Algorithms for Weighted Matchings". In: *Proceedings of the 6th Int'l Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX'03)*. Vol. 2764. Lecture Notes in Computer Science. Springer, 2003, pp. 14–23.

[46] Stefan Edelkamp, Shahid Jabbar, and Stefan Schrödl. "External A*". In: *Proceedings of the 271th Annual German Conference on Advances in Artificial Intelligence (KI'04)*. Vol. 3238. Lecture Notes in Computer Science. Springer, 2004, pp. 226–240.

[47] Jack Edmonds. "Maximum Matching and a Polyhedron with 0,1-vertices". In: *Journal of Research of the National Bureau of Standards B* 69 (1965), pp. 125–130.

[48] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. “Efficient Graph-Based Image Segmentation”. In: *International Journal of Computer Vision* 59.2 (2004), pp. 167–181.

[49] C. M. Fiduccia and R. M. Mattheyses. “A Linear-Time Heuristic for Improving Network Partitions”. In: *Proceedings of the 19th ACM/IEEE Conference on Design Automation*. IEEE, 1982, pp. 175–181.

[50] Per-Olof Fjallstrom. “Algorithms for Graph Partitioning: A Survey”. In: *Linkoping Electronic Articles in Computer and Information Science* 3.10 (1998).

[51] Steven Fortune and James Wyllie. “Parallelism in Random Access Machines”. In: *Proceedings of the 10th Annual ACM Symposium on the Theory of Computing (STOC’78)*. ACM Press, 1978, pp. 114–118.

[52] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. “Cache-Oblivious Algorithms”. In: *Proceedings of the 40st Annual IEEE Symposium on Foundations of Computer Science (FOCS’99)*. IEEE Computer Society Press, 1999, pp. 285–297.

[53] Natsuhiko Futamura, Srinivas Aluru, and Stefan Kurtz. “Parallel Suffix Sorting”. In: *Electrical Engineering and Computer Science* (2001). paper 64.

[54] Harold N. Gabow. “Data Structures for Weighted Matching and Nearest Common Ancestors with Linking”. In: *Proceedings of the 1th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA’90)*. SIAM, 1990, pp. 434–443.

[55] Harold N. Gabow and Robert E. Tarjan. “Faster Scaling Algorithms for General Graph Matching Problems”. In: *Journal of the ACM* 38.4 (1991), pp. 815–853.

[56] Eran Gal and Sivan Toledo. “Algorithms and Data Structures for Flash Memories”. In: *ACM Computing Surveys* 37.2 (2005), pp. 138–163.

[57] Robert Geisberger, Peter Sanders, and Dominik Schultes. “Better Approximation of Betweenness Centrality”. In: *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX’08)*. SIAM, 2008, pp. 90–100.

[58] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. “Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks”. In: *Proceedings of the 7th Workshop on Experimental Algorithms (WEA’08)*. Vol. 5038. Lecture Notes in Computer Science. Springer, 2008, pp. 319–333.

[59] Andrew V. Goldberg and Renato F. Werneck. “Computing Point-to-Point Shortest Paths from External Memory”. In: *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX’05)*. SIAM, 2005, pp. 26–40.

[60] Leslie M. Goldschlager. “A Unified Approach to Models of Synchronous Parallel Machines”. In: *Proceedings of the 10th Annual ACM Symposium on the Theory of Computing (STOC’78)*. ACM Press, 1978, pp. 89–94.

[61] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. "Sorting, Searching, and Simulation in the MapReduce Framework". In: *Proceedings of the 22th International Symposium on Algorithms and Computation (ISAAC'11)*. Vol. 7074. Lecture Notes in Computer Science. Springer, 2011, pp. 374–383.

[62] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. "GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management". In: *Proceedings of the the Int'l Conference on Management of Data (SIGMOD'06)*. ACM Press, 2006, pp. 325–336.

[63] Leonidas J. Guibas, Donald E. Knuth, and Micha Sharir. "Randomized Incremental Construction of Delaunay and Voronoi Diagrams". In: *Algorithmica* 7.1-6 (1992), pp. 381–413.

[64] David R. Helman, David A. Bader, and Joseph JáJá. "A Randomized Parallel Sorting Algorithm with an Experimental Study". In: *Journal of Parallel and Distributed Computing* 52.1 (1998), pp. 1–23.

[65] Bruce Hendrickson. "Chaco: Software for Partitioning Graphs". http://www.sandia.gov/~bahendr/chaco.html.

[66] Jaap-Henk Hoepman. "Simple Distributed Weighted Matchings". In: *arXiv preprint cs/0410047* (2004).

[67] Manuel Holtgrewe. "A Scalable Coarsening Phase for a Multi-Level Partitioning Algorithm". Diploma thesis. Universität Karlsruhe, 2009.

[68] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. "Engineering a Scalable High Quality Graph Partitioner". In: *Proceedings of the 24th Int'l Parallel and Distributed Processing Symposium (IPDPS'10)*. IEEE Computer Society Press, 2010, pp. 1–12.

[69] Amos Israeli and Alon Itai. "A Fast and Simple Randomized Parallel Algorithm for Maximal Matching". In: *Information Processing Letters* 22.2 (1986), pp. 77–80.

[70] Hideo Itoh and Hozumi Tanaka. "An Efficient Method for in Memory Construction of Suffix Arrays". In: *Proceedings of the 6th Symposium on String Processing and Information Retrieval (SPIRE'99)*. IEEE Computer Society Press, 1999, pp. 81–88.

[71] Vojtěch Jarník. "O jistém problému minimálním". In: *Práca Moravské Přírodovědecké Společnosti* 6 (1930). In Czech., pp. 57–63.

[72] J. J.Brennan. "Minimal Spanning Trees and Partial Sorting". In: *Operations Research Letter* 1.3 (1982), pp. 113–116.

[73] David R. Karger, Philip N. Klein, and Robert E. Tarjan. "A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees". In: *Journal of the ACM* 42.2 (1995), pp. 321–328.

[74] David R. Karger and Clifford Stein. "A New Approach to the Minimum Cut Problem". In: *Journal of the ACM* 43.4 (1996), pp. 601–640.

[75] Juha Kärkkäinen and Tommi Rantala. "Engineering Radix Sort for Strings". In: *Proceedings of the 16th Symposium on String Processing and Information Retrieval (SPIRE'09)*. Vol. 5280. Lecture Notes in Computer Science. Springer, 2009, pp. 3–14.

[76] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. "Linear Work Suffix Array Construction". In: *Journal of the ACM* 53.6 (2006), pp. 918–936.

[77] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. "A Model of Computation for MapReduce". In: *Proceedings of the 21st Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'10)*. SIAM, 2010, pp. 938–948.

[78] Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg. "Rapid Identification of Repeated Patterns in Strings, Trees and Arrays". In: *Proceedings of the 4th Annual ACM Symposium on the Theory of Computing (STOC'78)*. ACM Press, 72, pp. 125–136.

[79] Marek Karpiński and Wojciech Rytter. *Fast Parallel Algorithms for Graph Matching Problem*. Oxford University Press, 1998.

[80] George Karypis and Vipin Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". In: *SIAM Journal on Scientific Computing* 20.1 (1998), pp. 359–392.

[81] George Karypis and Vipin Kumar. *MeTiS, A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Version 4.0*. 1998. URL: http://glaros.dtc.umn.edu/gkhome/views/metis.

[82] George Karypis and Vipin Kumar. *MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0*. http://www.cs.umn.edu/~metis. 2009. URL: http://www.cs.umn.edu/~metis.

[83] Jyrki Katajainen and Olli Nevalainen. "An Alternative for the Implementation of Kruskal's Minimal Spanning Tree Algorithm". In: *Science of Computer Programming* 3.2 (1983), pp. 205–216.

[84] Irit Katriel, Peter Sanders, and Jesper Larsson Träff. "A Practical Minimum Spanning Tree Algorithm Using the Cycle Property". In: *11th European Symposium on Algorithms (ESA)*. Vol. 2832. Lecture Notes in Computer Science. Springer, 2003, pp. 679–690.

[85] A. Kershenbaum and Richard M. Van Slyke. "Computing Minimum Spanning Trees Efficiently". In: *Proceedings of the ACM Annual Conference (ACM'72)*. ACM Press, 1972, pp. 518–527.

[86] Valerie King. "A Simpler Minimum Spanning Tree Verification Algorithm". In: *Algorithmica* 18.2 (1997), pp. 263–270.

[87] Pang Ko and Srinivas Aluru. "Space Efficient Linear Time Construction of Suffix Arrays". In: *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM'03)*. Vol. 2676. Lecture Notes in Computer Science. 2003, pp. 200–210.

[88] Joseph B. Kruskal. "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem". In: *Proceedings of the American Mathematical Society* 7.1 (1956), pp. 48–50.

[89] Fabian Kulla and Peter Sanders. “Scalable Parallel Suffix Array Construction”. In: *Parallel Computing* 33.9 (2007), pp. 605 –612.

[90] Vijay Kumar and Eric J. Schwabe. “Improved Algorithms and Data Structures for Solving Graph Problems in External Memory”. In: *Proceedings of the 8th Int’l Symposium on Parallel and Distributed Processing (PDP’96)*. IEEE Computer Society Press, 1996, pp. 169–176.

[91] Jesper N. Larsson and Kunihiko Sadakane. “Faster Suffix Sorting”. In: *Theoretical Computer Science* 387.3 (2007), pp. 258–272.

[92] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders. “GPU Sample Sort”. In: *Proceedings of the 24th Int’l Parallel and Distributed Processing Symposium (IPDPS’10)*. IEEE Computer Society Press, 2010.

[93] Richard J. Lipton and Robert E. Tarjan. “Applications of a Planar Separator Theorem”. In: *SIAM Journal on Computing* 9.3 (1980), pp. 615–627.

[94] Michael Luby. “A Simple Parallel Algorithm for the Maximal Independent Set Problem”. In: *SIAM Journal on Computing* 15.4 (1986), pp. 1036–1053.

[95] Anil Maheshwari and Norbert Zeh. “External Memory Algorithms for Outerplanar Graphs”. In: *Proceedings of the 10th Int’l Symposium on Algorithms and Computation (ISAAC’99)*. Vol. 1741. Lecture Notes in Computer Science. Springer, 1999, pp. 307–316.

[96] Anil Maheshwari and Norbert Zeh. “I/O-efficient Algorithms for Graphs of Bounded Treewidth”. In: *Proceedings of the 12th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA’01)*. SIAM, 2001, 89—90.

[97] Anil Maheshwari and Norbert Zeh. “I/O-optimal Algorithms for Planar Graphs Using Separators”. In: *Proceedings of the 13th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA’02)*. SIAM, 2002, 372—381.

[98] Udi Manber and Gene Myers. “Suffix Arrays: A New Method for On-line String Searches”. In: *Proceedings of the 1th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA’90)*. SIAM, 1990, pp. 319–327.

[99] Michael A. Maniscalco and Simon J. Puglisi. “An Efficient, Versatile Approach to Suffix Sorting”. In: *ACM Journal of Experimental Algorithmics* 12 (2008), 1.2:1–1.2:23.

[100] Fredrik Manne and Rob H. Bisseling. “A Parallel Approximation Algorithm for the Weighted Maximum Matching Problem”. In: *Proceedings of the 7th Int’l Conference on Parallel Processing and Applied Mathematics (PPAM’07)*. Vol. 4967. Lecture Notes in Computer Science. Springer, 2008, pp. 708–717.

[101] Giovanni Manzini and Paolo Ferragina. “Engineering a Lightweight Suffix Array Construction Algorithm”. In: *Algorithmica* 40.1 (2004), pp. 33–50.

[102] Makoto Matsumoto and Takuji Nishimura. “Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator”. In: *ACM Transactions on Modeling and Computer Simulation* 8.1 (1998), pp. 3–30.

[103] Jens Maue and Peter Sanders. "Engineering Algorithms for Approximate Weighted Matching". In: *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*. Vol. 4525. Lecture Notes in Computer Science. Springer, 2007, pp. 242–255.

[104] Kurt Mehlhorn and Ulrich Meyer. "External-Memory Breadth-First Search with Sublinear I/O". In: *Proceedings of the 10th Annual European Symposium on Algorithms (ESA'02)*. Vol. 2461. Lecture Notes in Computer Science. Springer, 2002, pp. 723–735.

[105] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures. The Basic Toolbox*. Springer, 2008.

[106] Duane Merrill. *Back40computing: Fast and Efficient Software Primitives for GPU Computing*. http://code.google.com/p/back40computing/.

[107] Duane Merrill and Andrew S. Grimshaw. "High Performance and Scalable Radix Sorting: a Case Study of Implementing Dynamic Parallelism for GPU Computing". In: *Parallel Processing Letters* 21.2 (2011), pp. 245–272.

[108] Duane Merrill and Andrew S. Grimshaw. *Parallel Scan for Stream Architectures*. Tech. rep. Department of Computer Science, University of Virginia, 2009.

[109] Ulrich Meyer and Vitaly Osipov. "Design and Implementation of a Practical I/O-efficient Shortest Paths Algorithm". In: *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09)*. SIAM, 2009.

[110] Ulrich Meyer, Peter Sanders, and Jop Sibeyn. *Algorithms for Memory Hierarchies*. Vol. 2625. Lecture Notes in Computer Science. Springer, 2003.

[111] Ulrich Meyer and Norbert Zeh. "I/O-Efficient Undirected Shortest Paths". In: *Proceedings of the 11th Annual European Symposium on Algorithms (ESA'03)*. Vol. 2832. Lecture Notes in Computer Science. Springer, 2003, pp. 434–445.

[112] Ulrich Meyer and Norbert Zeh. "I/O-Efficient Undirected Shortest Paths with Unbounded Edge Lengths". In: *Proceedings of the 14th Annual European Symposium on Algorithms (ESA'06)*. Vol. 4168. Lecture Notes in Computer Science. Springer, 2006, pp. 540–551.

[113] Henning Meyerhenke, Burkhard Monien, and Thomas Sauerwald. "A New Diffusion-Based Multilevel Algorithm for Computing Graph Partitions". In: *Journal of Parallel and Distributed Computing* 69.9 (2009), pp. 750–761.

[114] Bernard M.E. Moret and Henry D. Shapiro. "An Empirical Assessment of Algorithms for Constructing a Minimum Spanning Tree". In: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 15 (1994), pp. 99–117.

[115] Yuta Mori. *Suffix Array Construction Algorithms Benchmark Set*. http://code.google.com/p/libdivsufsort/wiki/SACA_Benchmarks.

[116] Kamesh Munagala and Abhiram G. Ranade. "I/O-Complexity of Graph Algorithms". In: *Proceedings of the 10th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'99)*. SIAM, 1999, pp. 687–694.

[117] David R. Musser. "Introspective Sorting and Selection Algorithms". In: *Software: Practice and Experience* 27.8 (1997), pp. 983–993.

[118] Marc Najork and Janet L. Wiener. "Breadth-first Crawling Yields High-quality Pages". In: *Proceedings of the 10th Int'l Conference on World Wide Web (WWW'01)*. ACM Press, 2001, pp. 114–118.

[119] Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. "Otakar Boruvka on Minimum Spanning Tree Problem: Translation of Both the 1926 Papers, Comments, History". In: *Discrete Mathematics* 233.1 (2001), pp. 3–36.

[120] John von Neumann. *First Draft of a Report on the EDVAC*. Tech. rep. University of Pennsylvania, 1945.

[121] Ge Nong, and Wai Hong Chan. "Linear Suffix Array Construction by Almost Pure Induced-Sorting". In: *Proceedings of Data Compression Conference (DCC'09)*. IEEE Computer Society Press, 2009, pp. 193–202.

[122] Ge Nong, and Wai Hong Chan. "Two Efficient Algorithms for Linear Time Suffix Array Construction". In: *IEEE Transactions on Computers* 60.10 (2011), pp. 1471–1484.

[123] Kohei Noshita. "A Theorem on the Expected Complexity of Dijkstra's Shortest Path Algorithm". In: *Journal of Algorithms* 6.3 (1985), pp. 400–408.

[124] *Online Resources of the 9th DIMACS Implementation Challenge: Shortest Paths, 2006*. http://www.dis.uniroma1.it/~challenge9/.

[125] Vitaly Osipov. "Parallel Suffix Array Construction for Shared Memory Architectures". In: *Proceedings of the 19th Symposium on String Processing and Information Retrieval (SPIRE'12)*. Vol. 7608. Lecture Notes in Computer Science. Springer, 2012.

[126] Vitaly Osipov and Peter Sanders. "n-Level Graph Partitioning". In: *Proceedings of the 18th Annual European Symposium on Algorithms (ESA'10)*. Vol. 6346. Lecture Notes in Computer Science. Springer, 2010.

[127] Vitaly Osipov and Peter Sanders. "The Filter-Kruskal Minimum Spanning Tree Algorithm". In: *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09)*. SIAM, 2009.

[128] Rodrigo Paredes and Gonzalo Navarro. "Optimal Incremental Sorting". In: *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX'06)*. SIAM, 2006, pp. 171–182.

[129] Francois Pellegrini. *SCOTCH 5.1 User's Guide*. Tech. rep. Laboratoire Bordelais de Recherche en Informatique, Bordeaux, France, 2008. URL: http://www.labri.fr/perso/pelegrin/scotch/.

[130] Francois Pellegrini. *SCOTCH: Static Mapping, Graph, Mesh and Hypergraph Partitioning, and Parallel and Sequential Sparse Matrix Ordering Package*. http://www.labri.fr/perso/pelegrin/scotch/. 2007. URL: http://www.labri.fr/perso/pelegrin/scotch/.

[131] *Performance Portability and Programmability for Heterogeneous Many-core Architectures - PEPPHER*. http://www.peppher.eu/.

[132] Seth Pettie and Peter Sanders. "A Simpler Linear Time $2/3 - \epsilon$ Approximation for Maximum Weight Matching". In: *Information Processing Letters* 91.6 (2004), pp. 271–276.

[133] Robert Preis. "Linear Time 1/2-Approximation Algorithm for Maximum Weighted Matching in General Graphs". In: *Proceedings of the 16th International Symposium on Theoretical Aspects of Computer Science (STACS'99)*. Vol. 1563. Lecture Notes in Computer Science. Springer, 1999, pp. 259–269.

[134] Robert Preis. *PARTY Partitioning Library*. http://www2.cs.uni-paderborn.de/cs/robsy/party.html. 1996.

[135] Robert Preis and Ralf Diekmann. *The PARTY Partitioning Library, User Guide*. Tech. rep. Tr-rsfb-96-02. University of Paderborn, Germany, 1996. URL: http://www2.cs.uni-paderborn.de/cs/robsy/party.html.

[136] Robert C. Prim. "Shortest Connection Networks and Some Generalizations". In: *Bell System Technical Journal* 36.6 (1957), pp. 1389–1401.

[137] Simon J. Puglisi and William F. Smyth and. "A Taxonomy of Suffix Array Construction Algorithms". In: *ACM Computing Surveys* 39.2 (2007).

[138] Simon J. Puglisi and William F. Smyth and. "The Performance of Linear Time Suffix Sorting Algorithms". In: *Proceedings of Data Compression Conference (DCC'05)*. IEEE Computer Society Press, 2005, pp. 358–367.

[139] Benjamin Sach and Raphael Clifford. "An Empirical Study of Cache-Oblivious Priority Queues and their Application to the Shortest Path Problem". Available online under http://www.cs.bris.ac.uk/~sach/COSP/. 2008.

[140] Peter Sanders. "Algorithm Engineering–an Attempt at a Definition". In: *Efficient Algorithms*. Vol. 5760. Lecture Notes in Computer Science. Springer, 2009, pp. 321–340.

[141] Peter Sanders. "Fast Priority Queues for Cached Memorys". In: *ACM Journal of Experimental Algorithmics* (2000), pp. 316–321.

[142] Peter Sanders, Dominik Schultes, and Christian Vetter. "Mobile Route Planning". In: *Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08)*. Vol. 5193. Lecture Notes in Computer Science. Springer, 2008, pp. 732–743.

[143] Peter Sanders and Christian Schulz. "Distributed Evolutionary Graph Partitioning". In: *Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX'12)*. SIAM, 2012, pp. 16–29.

[144] Peter Sanders and Christian Schulz. "Engineering Multilevel Graph Partitioning Algorithms". In: *Proceedings of the 19th Annual European Symposium on Algorithms (ESA'11)*. Vol. 6942. Lecture Notes in Computer Science. Springer, 2011, pp. 469–480.

[145] Peter Sanders and Christian Schulz. "Think Locally, Act Globally: Highly Balanced Graph Partitioning". In: *Proceedings of the 12th Int'l Symposium on Experimental Algorithms (SEA'13)*. Vol. 7933. Lecture Notes in Computer Science. Springer, 2013, pp. 164–175.

[146] Peter Sanders and Sebastian Winkel. "Super Scalar Sample Sort". In: *Proceedings of the 12th Annual European Symposium on Algorithms (ESA'04)*. Vol. 3221. Lecture Notes in Computer Science. 2004, pp. 784–796.

[147] Nadathur Satish, Mark Harris, and Michael Garland. "Designing Efficient Sorting Algorithms for Manycore GPUs". In: *Proceedings of the 23rd Int'l Parallel and Distributed Processing Symposium (IPDPS'09)*. IEEE Computer Society Press, 2009.

[148] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. "Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort". In: *Proceedings of the the Int'l Conference on Management of Data (SIGMOD'10)*. ACM Press, 2010, pp. 351–362.

[149] Walter J. Savitch and Michael J. Stimson. "Time Bounded Random Access Machines with Parallel Processing". In: *Journal of the ACM* 26.1 (1979), pp. 103–118.

[150] Kirk Schloegel, George Karypis, and Vipin Kumar. *Graph Partitioning for High Performance Scientific Simulations*. Tech. rep. 00-018. University of Minnesota, 2000.

[151] Klaus-Bernd Schürmann and Jens Stoye. "An Incomplex Algorithm for Fast Suffix Array Construction". In: *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*. SIAM, 2005, pp. 77–85.

[152] *Seagate Technology*. http://www.seagate.com/cda/products/discsales/marketing/detail/0,1081,628,00.html.

[153] Raimund Seidel and Micha Sharir. "Top-Down Analysis of Path Compression". In: *SIAM Journal on Computing* 34.3 (2005), pp. 515–525.

[154] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. "Scan Primitives for GPU Computing". In: *Proceedings of the the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. ACM Press, 2007, pp. 97–106.

[155] John C. Shepherdson and Howard E. Sturgis. "Computability of Recursive Functions". In: *Journal of the ACM* 10.2 (1963), pp. 217–255.

[156] Vladislav Shkapenyuk and Torsten Suel. "Design and Implementation of a High-performance Distributed Web Crawler". In: *Proceedings of the 18th Int'l Conference on Data Engineering (ICDE'02)*. IEEE Computer Society Press, 2002, pp. 357–368.

[157] Jop Sibeyn. *From Parallel to External List Ranking*. Tech. rep. Max Planck Institut für Informatik, Saarbrücken, Germany, 1997.

[158] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[159] Johannes Singler, Peter Sanders, and Felix Putze. "MCSTL: The Multi-core Standard Template Library". In: *Proceedings of the Int'l Conference on Parallel Processing (Euro-Par'07)*. Vol. 4641. Lecture Notes in Computer Science. 2007, pp. 682–694.

[160] Erik Sintorn and Ulf Assarsson. “Fast Parallel GPU-sorting Using a Hybrid Algorithm”. In: *Journal of Parallel and Distributed Computing* 68.10 (2008), pp. 1381–1388.

[161] A. J. Soper, Chris Walshaw, and Mark Cross. “A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph Partitioning”. In: *Journal of Global Optimization* 29.2 (2004), pp. 225–241.

[162] *Space Imaging Gallery*. http://www.spaceimagingme.com/content/Gallery/.

[163] Weidong Sun and Zongmin Ma. “Parallel Lexicographic Names Construction with CUDA”. In: *Proceedings of the 15th Int’l Conference on Parallel and Distributed Systems (ICPADS’09)*. IEEE Computer Society Press, 2009, pp. 913–918.

[164] Ami Tavory, Vladimir Dreizin, and Benjamin Kosnik. “Policy-Based Data Structures”. http://gcc.gnu.org/onlinedocs/libstdc++/ext/pb_ds/ IBM Haifa and Redhat. 2004.

[165] *The Stanford WebBase Project*. http://www-diglib.stanford.edu/~testbed/doc2/WebBase/.

[166] *Transparent Parallel I/O Environment*. http://www.cs.duke.edu/TPIE/.

[167] Philippas Tsigas and Yi Zhang. “A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000”. In: *Proceedings of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing*. IEEE, 2003, pp. 372–381.

[168] Jeffrey S. Vitter. “External Memory Algorithms and Data Structures: Dealing with Massive Data”. In: *ACM Computing Surveys* 33.2 (2001), pp. 209–271.

[169] Jeffrey S. Vitter and E.A.M. Shriver. “Algorithms for Parallel Memory, I: Two-level Memories”. In: *Algorithmica* 12.2 (1994), pp. 110–147.

[170] Chris Walshaw. *JOSTLE –graph partitioning software*. http://staffweb.cms.gre.ac.uk/~wc06/jostle/. 2005. URL: http://staffweb.cms.gre.ac.uk/~wc06/jostle/.

[171] Chris Walshaw and Mark Cross. “JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview”. In: *Mesh Partitioning Techniques and Domain Decomposition Techniques*. Civil-Comp Ltd., 2007, pp. 27–58.

[172] David Weese. *Entwurf und Implementierung eines Generischen Substring-Index*. http://www.seqan.de/publications/weese06.pdf. 2006.

[173] Chin-Hsien Wu, Li-Pin Chang, and Tei-Wei Kuo. “An efficient R-Tree Implementation over Flash-memory Storage Systems”. In: *Proceedings of the 11th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM Press, 2003, pp. 17–24.

[174] Chin-Hsien Wu, Tei-Wei Kuo, and Li-Pin Chang. “An Efficient B-tree Layer Implementation for Flash-memory Storage Systems”. In: *ACM Transactions on Embedded Computing Systems* 6.3 (2007).

[175] Métivier Yves, John Michael Robson, Saheb-Djahromi Nasser, and Akka Zemmari. "An Optimal Bit Complexity Randomized Distributed MIS Algorithm". In: *Proceedings of the 16th Int'l Conference on Structural Information and Communication Complexity (SIROCCO'09)*. Vol. 5869. Lecture Notes in Computer Science. Springer, 2010, pp. 323–337.

APPENDIX A

Complete Data Sets

file	n	m
2cubes_sphere.mtx.graph	101492	772886
af_0_k101.mtx.graph	503625	8523525
af_1_k101.mtx.graph	503625	8523525
af_2_k101.mtx.graph	503625	8523525
af_3_k101.mtx.graph	503625	8523525
af_4_k101.mtx.graph	503625	8523525
af_5_k101.mtx.graph	503625	8523525
af_shell1.mtx.graph	504855	8542010
af_shell2.mtx.graph	504855	8542010
af_shell3.mtx.graph	504855	8542010
af_shell4.mtx.graph	504855	8542010
af_shell5.mtx.graph	504855	8542010
af_shell6.mtx.graph	504855	8542010
af_shell7.mtx.graph	504855	8542010
af_shell8.mtx.graph	504855	8542010
af_shell9.mtx.graph	504855	8542010
apache2.mtx.graph	715176	2051347
BenElechi1.mtx.graph	245874	6452311
bmw3_2.mtx.graph	227362	5530634
bmw7st_1.mtx.graph	141347	3599160
bmwcra_1.mtx.graph	148770	5247616
boneS01.mtx.graph	127224	3293964
boyd1.mtx.graph	93279	558985
c-73.mtx.graph	169422	554926
c-73b.mtx.graph	169422	554926
c-big.mtx.graph	345241	997885
cant.mtx.graph	62451	1972466
case39.mtx.graph	40216	516021
case39_A_01.mtx.graph	40216	516021
case39_A_02.mtx.graph	40216	516026
case39_A_03.mtx.graph	40216	516026
case39_A_04.mtx.graph	40216	516026
case39_A_05.mtx.graph	40216	516026
case39_A_06.mtx.graph	40216	516026
case39_A_07.mtx.graph	40216	516026
case39_A_08.mtx.graph	40216	516026
case39_A_09.mtx.graph	40216	516026
case39_A_10.mtx.graph	40216	516026
case39_A_11.mtx.graph	40216	516026
case39_A_12.mtx.graph	40216	516026
case39_A_13.mtx.graph	40216	516026
cfd1.mtx.graph	70656	878854
cfd2.mtx.graph	123440	1482229
CO.mtx.graph	221119	3722469
consph.mtx.graph	83334	2963573
cop20k_A.mtx.graph	99843	1262244
crankseg_1.mtx.graph	52804	5280703
crankseg_2.mtx.graph	63838	7042510
ct20stif.mtx.graph	52329	1323067
darcy003.mtx.graph	389874	933557
dawson5.mtx.graph	51537	479620
denormal.mtx.graph	89400	533412
dielFilterV2clx.mtx.graph	607232	12351020
dielFilterV3clx.mtx.graph	420408	16232900
Dubcova2.mtx.graph	65025	482600
Dubcova3.mtx.graph	146689	1744980
d_pretok.mtx.graph	182730	756256
ecology1.mtx.graph	1000000	1998000
ecology2.mtx.graph	999999	1997996
engine.mtx.graph	143571	2281251
F1.mtx.graph	343791	13246661
F2.mtx.graph	71505	2611390
Fault_639.mtx.graph	638802	13987881

file	n	m
filter3D.mtx.graph	106437	1300371
G3_circuit.mtx.graph	1585478	3037674
Ga10As10H30.mtx.graph	113081	3001276
Ga19As19H42.mtx.graph	133123	4375858
Ga3As3H12.mtx.graph	61349	2954799
Ga41As41H72.mtx.graph	268096	9110190
GaAsH6.mtx.graph	61349	1660230
gas_sensor.mtx.graph	66917	818224
Ge87H76.mtx.graph	112985	3889605
Ge99H100.mtx.graph	112985	4169205
gsm_106857.mtx.graph	589446	10584739
H2O.mtx.graph	67024	1074856
helm2d03.mtx.graph	392257	1174839
hood.mtx.graph	220542	5273947
IG5-17.mtx.graph	30162	1034600
invextr1_new.mtx.graph	30412	906915
kkt_power.mtx.graph	2063494	6482320
Lin.mtx.graph	256000	755200
mario002.mtx.graph	389874	933557
mixtank_new.mtx.graph	29957	982542
mouse_gene.mtx.graph	45101	14461095
msdoor.mtx.graph	415863	9912536
m_t1.mtx.graph	97578	4827996
nasasrb.mtx.graph	54870	1311227
nd12k.mtx.graph	36000	7092473
nd24k.mtx.graph	72000	14321817
nlpkkt80.mtx.graph	1062400	13821136
offshore.mtx.graph	259789	1991442
oilpan.mtx.graph	73752	1761718
parabolic_fem.mtx.graph	525825	1574400
pdb1HYS.mtx.graph	36417	2154174
pwtk.mtx.graph	217918	5708253
qa8fk.mtx.graph	66127	797226
qa8fm.mtx.graph	66127	797226
s3dkq4m2.mtx.graph	90449	2365221
s3dkt3m2.mtx.graph	90449	1831506
shipsec1.mtx.graph	140874	3836265
shipsec5.mtx.graph	179860	4966618
shipsec8.mtx.graph	114919	3269240
ship_001.mtx.graph	34920	2304655
ship_003.mtx.graph	121728	3982153
Si34H36.mtx.graph	97569	2529405
Si41Ge41H72.mtx.graph	185639	7412813
Si87H76.mtx.graph	240369	5210631
SiO.mtx.graph	33401	642127
SiO2.mtx.graph	155331	5564086
sparsine.mtx.graph	50000	749494
StocF-1465.mtx.graph	1465137	9770126
t3dh.mtx.graph	79171	2136467
t3dh_a.mtx.graph	79171	2136467
thermal2.mtx.graph	1228045	3676134
thread.mtx.graph	29736	2220156
tmt_sym.mtx.graph	726713	2177124
TSOPF_FS_b162_c3.mtx.graph	30798	896688
TSOPF_FS_b162_c4.mtx.graph	40798	1193898
TSOPF_FS_b300.mtx.graph	29214	2196173
TSOPF_FS_b300_c1.mtx.graph	29214	2196173
TSOPF_FS_b300_c2.mtx.graph	56814	4376395
TSOPF_FS_b39_c19.mtx.graph	76216	979241
TSOPF_FS_b39_c30.mtx.graph	120216	1545521
turon_m.mtx.graph	189924	778531
vanbody.mtx.graph	47072	1144913
x104.mtx.graph	108384	5029620

Table A.1.: Full List of Sparse Matrix Collection Instances

Table A.2.: All results for social network instances

Graph	k	KaSPar fast			KaPPa strong			KaPPa fast			KaPPa minimal			scotch			metis		
		best	avg	time	best	avg	time	best	avg	time	best	avg	time	best	avg	time	best	avg	time
coAuthorsCiteseer	2	17855	18003	25.77	21775	26462	12.93	29894	30997	11.98	32678	35492	6.75	34065	34065	5.34	21587	22674	0.30
coAuthorsCiteseer	4	34180	35315	51.75	43778	46540	28.43	44837	47156	17.03	50845	55514	5.78	52277	52277	7.51	39649	41560	0.33
coAuthorsCiteseer	8	49574	50054	85.49	56574	57647	46.35	53838	55686	25.77	61397	62752	5.10	69988	69988	9.61	56289	56996	0.36
coAuthorsCiteseer	16	59574	59915	124.51	66173	66648	55.26	62126	63085	31.04	65681	67007	5.71	83457	83457	11.41	68295	68744	0.39
coAuthorsCiteseer	32	67953	68752	169.78	72331	72736	64.53	71603	72062	30.34	74119	74760	5.49	90807	90807	12.92	77399	78254	0.41
coAuthorsCiteseer	64	76210	77326	193.85	77603	78756	64.45	79411	79872	26.47	81773	82244	5.82	100737	100737	14.20	84538	85426	0.44
citationCiteseer	2	32181	32247	49.44	35122	36696	24.37	34858	48466	15.31	47641	61055	11.51	37175	37175	5.87	33684	34344	0.67
citationCiteseer	4	67194	68371	135.82	76897	79782	47.87	76994	101369	27.40	120656	133916	12.54	79543	79543	11.02	73536	77524	0.76
citationCiteseer	8	103743	105663	297.70	119852	126129	85.92	118505	133337	47.09	188731	196204	13.41	124441	124441	15.37	108655	116082	0.83
citationCiteseer	16	148932	151256	507.69	156984	164984	114.26	156132	160555	57.02	218710	224851	14.18	163941	163941	18.83	153846	157000	0.91
citationCiteseer	32	198757	203173	841.73	205922	207923	147.02	198771	207089	111.12	248894	259090	45.56	210957	210957	21.88	197146	200650	0.98
citationCiteseer	64	255722	258037	1213.93	247462	248994	148.58	240660	241980	115.18	270692	279247	39.04	265971	265971	25.08	242010	244427	1.10
coAuthorsDBLP	2	45292	45650	82.42	54803	56140	23.13	55263	61619	16.96	63305	64872	11.78	63368	63368	8.23	48952	50341	0.53
coAuthorsDBLP	4	80408	81575	144.65	94651	97597	54.59	97007	98865	32.09	123373	126675	9.52	109856	109856	11.73	88513	88734	0.61
coAuthorsDBLP	8	109940	113575	263.08	126261	128129	82.57	116839	118190	46.32	144839	147038	8.62	142749	142749	14.49	115201	117074	0.69
coAuthorsDBLP	16	132067	135259	440.42	144229	145229	98.64	137946	138968	46.24	152803	154368	7.51	169706	169706	16.97	138399	140149	0.75
coAuthorsDBLP	32	152146	154501	787.16	157754	159086	113.74	151883	153606	62.94	160331	161779	14.80	189201	189201	18.95	160842	161565	0.82
coAuthorsDBLP	64	168939	169122	1099.75	169681	170403	123.69	169283	169671	46.08	174708	175679	10.11	207486	207486	20.70	175660	177172	0.88
cnr2000	2	210	236	49.54	2597	3789	25.90	2430	4835	23.21	2422	5053	18.48	20537	20537	3.70	1773	2451	7.44
cnr2000	4	1569	1973	64.02	6089	6971	46.71	6284	6939	27.50	6175	7138	13.34	26809	26809	6.20	4301	6326	8.05
cnr2000	8	4096	4974	75.55	7914	8510	55.07	7378	7911	33.20	7684	8308	12.67	31373	31373	8.18	7899	16951	8.85
cnr2000	16	6943	14824	92.57	8784	10382	61.45	9399	9567	32.72	9805	10003	12.45	34967	34967	10.84	12601	81752	9.47
cnr2000	32	384058	400272	188.69	360661	363687	88.28	368182	372503	48.97	374786	375787	14.89	432813	432813	12.59	368062	409130	9.78
cnr2000	64	713772	723710	483.68	694270	700504	103.42	706366	712434	52.27	722754	727917	15.49	727685	727685	14.06	723221	737874	10.31
coPapersDBLP	2	462530	466947	372.39	512389	527205	80.25	490054	552438	66.76	528953	585647	60.58	622378	622378	42.06	599794	634286	2.33
coPapersDBLP	4	822518	838005	705.59	937267	952505	122.23	1021741	1034181	88.66	1409276	1428094	42.62	1188052	1188052	76.19	1073007	1091355	2.58
coPapersDBLP	8	1188694	1213398	1794.77	1257622	1293223	201.41	1296044	1315313	131.60	1690906	1751688	32.16	1685436	1685436	98.29	1442079	1495943	2.81
coPapersDBLP	16	1534078	1544591	3993.84	1540054	1571957	318.56	1593642	1614871	165.98	1816467	1852634	28.73	2028374	2028374	131.89	1864836	1886340	3.01
coPapersDBLP	32	1789129	1798109	6550.18	1828015	1850535	411.34	1790694	1861113	276.51	1926975	2009450	37.35	2380424	2380424	156.07	2087868	2122569	3.17
coPapersDBLP	64	2039271	2054249	10897.41	2164396	2177596	423.03	2051766	2061702	244.46	2132793	2139541	31.87	2697328	2697328	148.72	2341150	2347850	3.39

Table A.3.: All results for large instances.

Graph	k	KaSPar strong			KaSPar fast			KaPPa strong			KaPPa fast			KaPPa minimal			scotch			metis		
		best	avg	time	best	avg	time	best	avg	time	best	avg	time	best	avg	time	best	avg	time	best	avg	time
fe_tooth	2	3844	3987	5.86	3840	3981	1.16	3951	4336	3.75	3854	4353	2.44	4109	6490	1.59	4259	4259	0.38	4372	4529	0.08
fe_tooth	4	6937	6999	8.54	7034	7146	1.49	7012	7189	5.22	7126	7757	2.97	7780	9157	0.96	8304	8304	0.72	7805	8280	0.08
fe_tooth	8	11482	11564	13.43	11574	12007	1.96	12272	12721	6.83	12215	12678	4.06	13243	13671	0.75	12999	12999	1.09	13334	13768	0.08
fe_tooth	16	17744	17966	21.24	17968	18288	2.79	18302	18570	7.18	18198	18524	3.55	19559	19813	0.65	20816	20816	1.59	20035	20386	0.09
fe_tooth	32	25888	26248	35.12	26249	26592	4.03	26397	26617	5.28	26404	26677	2.92	28070	28391	0.53	28430	28430	2.13	28547	29052	0.10
fe_tooth	64	36259	36469	49.65	36741	40385	5.80	36862	37002	4.71	36795	36992	2.57	38423	39095	0.62	38401	38401	2.69	39233	39381	0.12
598a	2	2371	2384	6.50	2378	2389	1.84	2387	2393	5.64	2391	2401	3.79	2456	2485	2.99	2417	2417	0.39	2444	2513	0.14
598a	4	7897	7921	11.15	7935	7977	2.42	8235	8291	10.24	8291	8385	5.94	9224	9862	2.62	8246	8246	0.95	8466	8729	0.15
598a	8	15929	15984	22.31	15992	16125	3.48	16502	16641	12.21	16461	16598	7.07	17351	17899	2.98	17490	17490	1.63	17170	17533	0.16
598a	16	26046	26270	38.39	26102	26672	5.05	26467	26825	17.74	26670	26887	12.51	27983	28596	6.76	29804	29804	2.37	27857	28854	0.17
598a	32	39625	40019	60.60	40563	40986	7.25	40946	41190	18.16	40928	41186	11.91	43111	43741	7.74	44756	44756	3.21	43256	44213	0.19
598a	64	58362	58945	87.52	58326	59199	10.72	59148	59387	14.15	59026	59233	9.64	61396	61924	6.21	64561	64561	4.11	61888	62703	0.22
fe_ocean	2	317	317	5.55	317	322	1.66	314	317	3.21	314	318	2.11	343	355	1.71	402	402	0.18	540	579	0.11
fe_ocean	4	1801	1810	9.40	1817	1837	1.95	1756	1822	6.30	1754	1822	3.03	1990	2051	1.10	2000	2000	0.44	2102	2140	0.11
fe_ocean	8	4044	4097	14.33	4084	4195	2.51	4104	4252	6.33	4143	4330	2.93	4689	4987	0.73	4956	4956	0.81	5256	5472	0.12
fe_ocean	16	7992	8145	22.41	8120	8359	3.39	8188	8350	5.62	8294	8469	3.04	9457	9553	0.70	9351	9351	1.27	10115	10377	0.13
fe_ocean	32	13320	13518	36.53	13526	13806	5.00	13593	13815	4.34	13618	14042	2.15	15465	15657	0.47	15089	15089	1.83	16565	16877	0.15
fe_ocean	64	21326	21739	62.46	22059	22209	7.78	21636	21859	3.68	21809	21973	2.02	24147	24275	0.51	23246	23246	2.49	24198	24531	0.17
144	2	6455	6507	12.81	6461	6491	3.04	6559	6623	7.45	6563	6638	5.23	6747	6799	3.64	6695	6695	0.66	6804	6972	0.20
144	4	15312	15471	24.73	15717	15774	4.10	16870	16963	13.33	16998	17122	7.00	17364	18101	2.97	16899	16899	1.44	17144	17487	0.21
144	8	25130	25409	38.13	25557	26039	5.54	26300	26457	20.11	26435	26614	10.49	27206	27829	2.93	28172	28172	2.24	28006	28194	0.22
144	16	37872	38404	69.35	38830	39161	8.30	39010	39319	26.04	39266	39492	17.53	40264	41977	6.63	43712	43712	3.12	42861	43041	0.24
144	32	57082	57492	106.40	57353	57860	11.73	58331	58631	24.60	58175	58652	16.03	61774	62171	8.79	63224	63224	4.14	61716	62481	0.26
144	64	80313	80770	144.77	80609	81293	16.05	82286	82452	19.11	82029	82493	12.05	86067	86950	8.16	88246	88246	5.25	86534	87208	0.30
wave	2	8661	8720	16.19	8650	8690	3.25	8832	9132	8.24	8809	9108	4.72	8966	9324	2.84	9337	9337	0.83	9169	9345	0.19
wave	4	16806	16920	29.56	16871	16978	4.39	17008	17250	14.51	17263	17503	6.84	18041	21189	1.81	19995	19995	1.72	19929	21906	0.20
wave	8	28681	28817	46.61	28865	29200	6.01	30690	31419	20.63	30628	31371	9.79	32617	33937	1.50	33357	33357	2.61	33223	33639	0.21
wave	16	42918	43208	75.97	43267	43770	8.31	44831	45048	20.54	44936	45202	10.73	46293	47270	1.47	48903	48903	3.53	48404	49000	0.22
wave	32	63025	63159	112.19	62764	63266	11.88	63981	64390	14.94	64004	64532	8.19	68085	68620	1.02	70581	70581	4.68	68062	68604	0.25
wave	64	87243	87554	150.37	87403	87889	16.88	88376	88964	12.51	88924	89297	6.09	92366	93424	1.03	96759	96759	5.90	92148	94083	0.29
m14b	2	3828	3846	20.03	3845	3870	4.49	3862	3954	11.16	3900	3945	7.80	3951	4208	5.63	3872	3872	0.70	4036	4155	0.31
m14b	4	13015	13079	26.51	13111	13160	5.42	13543	13810	18.77	14104	14211	10.21	14990	15094	4.42	13484	13484	1.71	13932	14560	0.33
m14b	8	25573	25756	45.33	25839	26086	7.27	27330	27393	24.97	27411	27450	11.76	28241	28517	3.78	27839	27839	2.86	28138	28507	0.34
m14b	16	42212	42458	83.25	42727	43365	10.07	45352	45762	28.11	45931	46108	19.27	48769	49397	6.41	50778	50778	4.25	48314	49269	0.36
m14b	32	66314	66991	133.88	66942	68017	14.52	68107	69075	29.94	68715	69223	17.99	72484	73598	8.12	75453	75453	5.72	72746	74135	0.40
m14b	64	99207	100014	198.23	99964	100666	20.91	101053	101455	25.26	101410	101861	17.46	106361	107173	10.24	109404	109404	7.38	107384	108141	0.44
auto	2	9740	9768	68.39	9744	9776	10.99	9910	10045	30.09	9863	10856	18.86	10313	11813	12.12	10666	10666	1.61	10781	12147	0.83
auto	4	25988	26062	75.60	26072	26116	13.35	28218	29481	64.01	29690	29995	33.11	32473	33371	8.93	29046	29046	3.52	27469	30318	0.86
auto	8	45099	45232	97.60	45416	45806	15.98	46272	46652	85.89	47163	48229	46.36	49447	53617	8.42	49999	49999	5.42	49691	52422	0.87
auto	16	76287	76715	153.46	77376	77801	20.81	78713	79769	87.41	79711	80683	58.20	84236	86001	12.25	84462	84462	7.84	85562	89139	0.91
auto	32	121269	121862	246.50	122406	123052	28.12	124606	125500	71.77	124920	125876	46.44	131545	133723	20.23	133403	133403	10.58	133026	134086	0.99
auto	64	174612	174914	352.09	174712	176214	38.76	177038	177595	62.64	177461	178119	44.14	185836	187424	25.39	193170	193170	13.68	188555	189699	1.08
delaunay_n20	2	1711	1731	196.33	1726	1753	12.88	1858	1882	35.43	1879	1898	18.66	1911	1937	13.87	1874	1874	1.18	2054	2194	1.11
delaunay_n20	4	3418	3439	130.67	3460	3480	13.21	3674	3780	64.08	3784	3826	24.34	3857	3900	8.97	3723	3723	2.35	4046	4094	1.15
delaunay_n20	8	6278	6317	104.37	6364	6387	13.71	6670	6854	70.07	6688	6872	41.92	7161	7303	6.15	7180	7180	3.58	7705	8029	1.13
delaunay_n20	16	10183	10218	84.33	10230	10327	13.80	10816	11008	67.92	10882	11061	48.05	11307	11533	6.31	11266	11266	4.77	11854	12440	1.14
delaunay_n20	32	15905	16026	101.69	16211	16236	14.90	16813	17086	42.67	16814	17150	24.44	17993	18179	3.33	17784	17784	6.04	18816	19304	1.18
delaunay_n20	64	23935	23962	97.09	24193	24263	16.40	24799	25179	22.04	24946	25129	12.83	26314	27001	1.79	26163	26163	7.34	28318	28543	1.21
rgg_n_2_20_s0	2	2162	2201	198.61	2146	2217	16.75	2377	2498	33.24	2378	2497	24.66	2400	2530	20.71	2832	2832	1.41	3023	3326	1.57
rgg_n_2_20_s0	4	4323	4389	130.00	4382	4448	17.18	4867	5058	38.50	4870	4973	21.06	5114	5200	11.11	5737	5737	2.82	5786	6174	1.56
rgg_n_2_20_s0	8	7745	7915	103.66	8031	8174	17.81	8995	9391	46.06	9248	9493	25.50	9426	9632	7.83	11251	11251	4.48	11365	11771	1.54
rgg_n_2_20_s0	16	12596	12792	86.19	12981	13148	17.93	14953	15199	35.86	15013	15339	24.61	15039	15442	7.20	17157	17157	6.13	17498	18125	1.53
rgg_n_2_20_s0	32	20403	20478	100.03	20805	20958	18.99	23430	23917	26.04	23383	24222	16.93	23842	24164	3.94	28078	28078	7.96	27765	28495	1.58
rgg_n_2_20_s0	64	30860	31066	97.83	31203	31584	20.50	34778	35354	11.62	35086	35539	9.95	35252	35629	2.09	38815	38815	9.83	41066	42465	1.58
af_shell10	2	26225	26225	317.11	26225	26225	37.00	26225	26225	78.65	26225	26225	65.31	26525	26640	59.76	26825	26825	3.64	27625	28955	2.99
af_shell10	4	55075	55345	210.61	55875	56375	36.59	54950	55265	91.96	54950	55500	51.52	58366	58627	22.11	58500	58500	7.60	61100	64705	3.04
af_shell10	8	97709	100233	179.51	100325	102667	38.47	101425	102335	136.99	102125	103180	61.16	110369	111081	16.03	105375	105375	11.97	117650	120120	3.04
af_shell10	16	163125	165770	212.12	163600	165360	40.47	165025	166427	106.63	165625	166480	69.97	174677	175918	17.00	171725	171725	16.45	184350	188765	3.06
af_shell10	32	248268	252939	191.53	252555	256262	43.14	253525	255535	80.85	252487	255746	52.00	270249	275149	9.25	269375	269375	21.66	289400	291590	3.13
af_shell10	64	372823	376512	207.76	378031	382191	49.38	379125	382923	43.01	380225	384140	29.43	400378	404085	4.82	402275	402275	27.33	421285	427047	3.18
deu	2	167	172	231.47	175	179	58.31	214	221	68.20	230	240	47.55	233	243	38.11	295	295	3.19	268	286	5.38
deu	4	419	426	244.12	427	447	58.84	533	542	76.87	531	545	49.37	544	580	25.65	726	726	6.46	699	761	5.35
deu	8	762	773	250.50	781	792	59.20	922	962	99.76	935	973	45.05	974	1007	19.57	1235	1235	9.84	1174	1330	5.24
deu	16	1308	1333	278.31	1332	1387	61.82	1550	1616	105.96	1556	1618	78.82	1593	1656	21.79	2066	2066	13.11	2041	2161	5.19
deu	32	2182	2217	283.79	2251	2295	62.50	2548	2615	73.17	2535	2641	41.93	2626	2711	11.50	3250	3250	16.28	3319	3445	5.28
deu	64	3610	3631	293.53	3679	3737	64.38	4021	4093	49.55	4078	4146	31.63	4193	4317	5.97	4978	4978	19.41	5147	5385	5.31
eur	2	133	138	1946.34	162	211	792.68										469	469	12.45			
eur	4	355	375	2168.10	416	431	794.41	543	619	441.11	580	646	223.96	657	697	113.35	952	952	25.37	846	1626	29.40
eur	8	774	786	2232.31	823	834	809.21	986	1034	418.29	1013	1034	207.41	1060	1119	80.92	1667	1667	38.67	1675	3227	29.04
eur	16	1401	1440	2553.40	1575	1597	930.59	1760	1900	497.93	1907	1935	295.81	1931	2048	94.56	2922	2922	51.50	3519	9395	30.58
eur	32	2595	2643	2598.84	2681	2761	958.24	3186	3291	417.52	3231	3314	306.52	3202	3386	55.63	4336	4336	65.16	7424	9442	30.81
eur	64	4502	4526	2533.56	4622	4675	868.75	5290	5393	308.17	5448	5538	183.98	5569	5770	29.64	6772	6772	77.14	11313	12738	30.30

Graph	2		4		8		16		32		64	
add20	641	594	1212	1177	1814	1704	2427	2121		2687		3236
data	190	188	405	383	699	660		1162		1865		2885
3elt	90	89	201	199	361	342	654	569		969		1564
uk	**19**	19	**41**	42	92	84	179	152		258		438
add32	**10**	10	**33**	33	**66**	66	**117**	117	**212**	212		493
bcsstk33	10105	10097	21756	21508	34377	34178	56687	54860		78132		108505
whitaker3	**126**	126	382	380	670	656	1163	1093		1717		2567
crack	184	183	370	362	696	678	1183	1092		1707		2566
wing_nodal	1703	1696	3609	3572	5574	5443	8624	8422		11980		16134
fe_4elt2	**130**	130	**349**	349	622	605	1051	1014		1657		2537
vibrobox	11538	10310	19267	19199	25190	24553	35514	32167	46331	41399		49521
bcsstk29	**2818**	2818	**8035**	8159	14212	13965	23808	21768		34886		57054
4elt	**138**	138	325	321	561	534	1009	939		1559		2596
fe_sphere	**386**	386	798	768	1236	1152	1914	1730		2565		3663
cti	**318**	318	950	944	1815	1802	3056	2906	5044	4223		5875
memplus	5698	5489	10234	9559	12599	11785	14410	13241	16340	14395		16857
cs4	378	367	970	940	1520	1467	2285	2206	3521	3090		4169
bcsstk30	6347	6335	**16617**	16622	34761	34604	72028	71234		115770		173945
bcsstk31	2723	2701	**7351**	7444	**13371**	13417	24791	24277	42745	38086		60528
fe_pwt	**340**	340	**704**	704	**1441**	1442	2835	2806		5612		8454
bcsstk32	**4667**	4667	**9247**	9492	**20855**	21490	**37372**	37673	72471	61144		95199
fe_body	**262**	262	**599**	636	**1079**	1156	**1858**	1931		3202		5282
t60k	78	75	213	211	470	465	866	849	1493	1391		2211
wing	803	787	1683	1666	2616	2589	4147	4131	6271	5902		8132
brack2	**708**	708	**3027**	3038	**7144**	7269	**11969**	11983	18496	17798		26557
finan512	**162**	162	**324**	324	**648**	648	**1296**	1296	**2592**	2592		10560
fe_tooth	**3819**	3823	**6938**	7103	**11650**	11935	**18115**	18283	26604	25977		35980
fe_rotor	2055	2045	**7405**	7480	**12959**	13165	21093	20773	33588	32783		47461
598a	2390	2388	**7992**	8154	**16179**	16467	**26196**	26427	**40513**	40674		59098
fe_ocean	388	387	**1856**	1878	**4251**	4299	**8276**	8432	13841	13660		21548
144	6489	6479	**15196**	15345	**25455**	25818	**38940**	39352	58359	58126		81145
wave	8716	8682	**16891**	17475	**29207**	30511	**43697**	44611	**64198**	64551		88863
m14b	3828	3826	**13034**	13391	**25921**	26666	**42513**	43975	67990	67770		101551
auto	**10004**	10042	**26941**	27790	**45731**	47650	**77618**	79847	**123296**	124991	179309	175975

Table A.4.: Walshaw Benchmark with $\epsilon = 1$

Graph	2		4		8		16		32		64	
add20	636	576	1195	1158	1765	1690	2331	2095	2862	2493		3152
data	186	185	379	378	662	650	1163	1133	1972	1802		2809
3elt	**87**	87	199	198	346	336	587	565	1035	958	1756	1542
uk	**18**	18	**40**	40	84	81	158	148	281	251	493	414
add32	**10**	10	**33**	33	**66**	66	**117**	117	**212**	212	509	493
bcsstk33	**10064**	10064	21083	21035	34150	34078	55372	54510	80548	77672	113269	107012
whitaker3	**126**	126	381	378	662	655	1125	1092	1757	1686	2733	2535
crack	**182**	182	**360**	360	685	676	1132	1082	1765	1679	2739	2553
wing_nodal	1681	1680	3572	3561	5424	5401	8476	8316	12282	11938	16891	15971
fe_4elt2	**130**	130	349	343	607	598	1022	1007	1686	1633	2658	2527
vibrobox	11538	10310	19239	18778	24691	24171	34226	31516	43532	39592	52242	49123
bcsstk29	**2818**	2818	**7983**	8045	14041	13817	22448	21410	35660	34407	58644	55366
4elt	**137**	137	**319**	319	533	523	942	914	1631	1537	2728	2581
fe_sphere	**384**	384	792	764	1193	1152	1816	1706	2715	2477	3965	3547
cti	**318**	318	924	917	1724	1716	2900	2778	4396	4132	6330	5763
memplus	5626	5355	10145	9418	12521	11628	14168	13130	15850	14264	18364	16724
cs4	366	361	959	936	1490	1467	2215	2126	3152	3048	4479	4169
bcsstk30	**6251**	6251	**16497**	16537	**34275**	34513	70851	70278	117500	114005	178977	171727
bcsstk31	**2676**	2676	7183	7181	**13090**	13246	24211	23504	39298	37459	60847	58667
fe_pwt	**340**	340	**704**	704	**1416**	1419	2787	2784	5649	5606	8557	8346
bcsstk32	**4667**	4667	**8778**	8799	**20035**	21023	**35788**	36613	61485	59824	96086	92690
fe_body	**262**	262	**598**	601	**1033**	1054	**1767**	1800	**2906**	2947	**4982**	5212
t60k	**71**	71	211	207	461	454	851	822	1423	1391	2264	2198
wing	789	774	1660	1636	2567	2551	4034	4015	6005	5832	8316	8043
brack2	**684**	684	2853	2839	**6980**	6994	**11622**	11741	**17491**	17649	26679	26366
finan512	**162**	162	**324**	324	**648**	648	**1296**	1296	**2592**	2592	10635	10560
fe_tooth	3794	3792	**6862**	6946	**11422**	11662	**17655**	17760	25685	25624	35962	35830
fe_rotor	**1960**	1963	**7182**	7222	**12546**	12852	**20356**	20521	32114	31763	47613	47049
598a	2369	2367	**7873**	7955	**15820**	16031	**25927**	25966	**39525**	39829	**58101**	58454
fe_ocean	**311**	311	1710	1698	3976	3974	7919	7838	12942	12746	21217	21033
144	6456	6438	**15122**	15250	**25301**	25491	**37899**	38478	**56463**	57354	**80621**	80767
wave	8640	8616	**16822**	16936	**28664**	28839	**42620**	43063	**62281**	62743	**86663**	87325
m14b	3828	3823	**12977**	13136	**25550**	26057	**42061**	42783	**65879**	67326	**98188**	100286
auto	**9716**	9782	**25979**	26379	**45109**	45525	**76016**	77611	**120534**	122902	**172357**	174904

Table A.5.: Walshaw Benchmark with $\epsilon = 3$

Graph	2		4		8		16		32		64	
add20	610	550	1186	1157	1755	1675	2267	2081	2786	2463	3270	3152
data	183	181	369	368	640	628	1130	1086	1907	1777	3073	2798
3elt	**87**	87	198	197	336	330	572	560	1009	950	1645	1539
uk	**18**	18	**39**	40	81	78	150	139	272	246	456	410
add32	**10**	10	**33**	33	**63**	65	**117**	117	**212**	212	**491**	493
bcsstk33	**9914**	9914	**20198**	20584	33971	33938	55273	54323	79159	77163	111659	106886
whitaker3	**126**	126	380	378	658	650	1110	1084	1741	1686	2663	2535
crack	**182**	182	361	360	673	667	1096	1080	1749	1679	2681	2548
wing_nodal	1672	1668	3541	3536	5375	5350	8419	8316	12149	11879	16566	15873
fe_4elt2	**130**	130	340	335	596	583	1013	991	1665	1633	2608	2516
vibrobox	11538	10310	19021	18778	24203	23930	34298	31235	42890	39592	50994	48200
bcsstk29	**2818**	2818	**7936**	7942	13619	13614	21914	20924	34906	33818	57220	54935
4elt	**137**	137	318	315	519	516	925	902	1574	1532	2673	2565
fe_sphere	**384**	384	784	764	1219	1152	1801	1692	2678	2477	3904	3547
cti	**318**	318	900	890	**1708**	1716	2830	2725	4227	4037	6127	5684
memplus	5516	5267	10011	9299	12458	11555	14047	13078	15749	14170	18213	16454
cs4	363	356	955	936	1483	1467	2184	2126	3115	2995	4394	4116
bcsstk30	**6251**	6251	**16186**	16332	**34146**	34350	**69520**	70043	114960	113321	175723	170591
bcsstk31	**2676**	2676	**7099**	7152	**12941**	13058	23603	23254	38150	37459	60768	57534
fe_pwt	**340**	340	**700**	701	**1405**	1409	**2772**	2777	**5545**	5546	8410	8310
bcsstk32	**4622**	4644	**8454**	8481	**19678**	20099	**35208**	35965	60441	59824	94238	91006
fe_body	**262**	262	**596**	601	**1017**	1054	**1723**	1784	**2807**	2887	**4834**	4888
t60k	**65**	65	202	196	457	454	839	818	1398	1376	2229	2168
wing	784	770	1654	1636	**2528**	2551	**3998**	4015	5915	5806	8228	7991
brack2	**660**	660	2745	2739	**6671**	6781	**11358**	11558	**17256**	17529	26321	26281
finan512	**162**	162	**324**	324	**648**	648	**1296**	1296	**2592**	2592	10583	10560
fe_tooth	3780	3773	**6825**	6864	**11337**	11662	**17404**	17603	**25216**	25624	**35466**	35476
fe_rotor	**1950**	1955	7052	7045	**12380**	12566	**20039**	20132	**31450**	31576	46749	46608
598a	2338	2336	**7763**	7851	**15544**	15721	**25585**	25808	**39144**	39369	**57412**	58031
fe_ocean	**311**	311	1705	1697	3946	3941	**7618**	7722	**12720**	12746	20886	20667
144	6373	6362	**15036**	15250	**25025**	25259	**37433**	38225	**56345**	56926	**79296**	80257
wave	8598	8563	**16662**	16820	**28615**	28700	**42482**	42800	**61788**	62520	**85658**	86663
m14b	3806	3802	**12976**	13136	**25292**	25679	**41750**	42608	**65231**	66793	**98005**	99063
auto	9487	9450	**25399**	25883	**44520**	45039	**75066**	76488	**120001**	122378	**171459**	173968

Table A.6.: Walshaw Benchmark with $\epsilon = 5$

APPENDIX B

List of Publications

[1] Deepak Ajwani, Ulrich Meyer, and Vitaly Osipov. "Improved External Memory BFS Implementation". In: *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*. SIAM, 2007.

[2] Deepak Ajwani, Roman Dementiev, Ulrich Meyer, and Vitaly Osipov. "Breadth First Search on Massive Graphs". In: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 74 (2009), pp. 291–308.

[3] Timo Bingmann, Johannes Fischer, and Vitaly Osipov. "Inducing Suffix and LCP Arrays in External Memory". In: *Proceedings of the 15th Workshop on Algorithm Engineering and Experiments (ALENEX'13)*. SIAM, 2013.

[4] Marcel Birn, Vitaly Osipov, Peter Sanders, Christian Schulz, and Nodari Sitchinava. "Efficient Parallel and External Matching". In: *Proceedings of the Int'l Conference on Parallel Processing (Euro-Par'13)*. Lecture Notes in Computer Science. to appear. Springer, 2013.

[5] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders. "GPU Sample Sort". In: *Proceedings of the 24th Int'l Parallel and Distributed Processing Symposium (IPDPS'10)*. IEEE Computer Society Press, 2010.

[6] Ulrich Meyer and Vitaly Osipov. "Design and Implementation of a Practical I/O-efficient Shortest Paths Algorithm". In: *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09)*. SIAM, 2009.

[7] Vitaly Osipov. "Parallel Suffix Array Construction for Shared Memory Architectures". In: *Proceedings of the 19th Symposium on String Processing and Information Retrieval (SPIRE'12)*. Vol. 7608. Lecture Notes in Computer Science. Springer, 2012.

[8] Vitaly Osipov and Peter Sanders. "n-Level Graph Partitioning". In: *Proceedings of the 18th Annual European Symposium on Algorithms (ESA'10)*. Vol. 6346. Lecture Notes in Computer Science. Springer, 2010.

[9] Vitaly Osipov and Peter Sanders. “The Filter-Kruskal Minimum Spanning Tree Algorithm”. In: *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09)*. SIAM, 2009.

[10] Vitaly Osipov, Peter Sanders, and Christian Schulz. “Engineering Graph Partitioning Algorithms”. In: *Proceedings of the 11th Int'l Symposium on Experimental Algorithms (SEA'12)*. Ed. by Ralf Klasing. Vol. 7276. Lecture Notes in Computer Science. Springer, 2012.

ANHANG C

Zusammenfassung

Fundamentale Algorithmen umfassen Basiswissen für jeden Bachelorstudenten in Informatik als auch für professionelle Programmierer. Das ist eine Menge von Verfahren, die in jedem (guten) Buch über Algorithmen und Datenstrukturen beschrieben ist. Wenn man das Inhaltsverzeichnis einer dieser Bücher, zum Beispiel von Sanders und Mehlhorn [105], ließt, sieht man, dass wir uns mit den meisten dort beschriebenen algorithmischen Problemen beschäftigt haben. Unter anderem beschreiben wir gegenwärtige Fortschritte für solche klassische Probleme wie Sortierung, Berechnung von kürzesten Wegen, minimalen Spannbäumen, maximale Matchings in Graphen, Breitensuche in Graphen und Partitionierung von Graphen.

Man könnte behaupten, dass diese Probleme längst gelöst sind und die optimalen Schranken für sie schon lang bekannt sind. Das ist soweit richtig, wenn unsere Annahmen über das konkrete Problem und die Architektur stimmen. Im größten Teil der Kursbücher ist es das *Random Access Machine* (RAM) Model. Dieses Model nimmt an, dass die Eingabe in den Hauptspeicher passt, sowie dass die Zugriffszeit auf den Hauptspeicher gleichverteilt ist. Leider stimmt es oft nicht für reale Architekturen, die eine Speicherhierarchie besitzen. Die Speicherhierarchie umfasst mehrere Schichten - vom Cache bis zur Festplatte. Zum Beispiel kann die Breitensuche in einem Graph mehrere Größenordnungen langsamer sein sobald der Graph nicht mehr in den Hauptspeicher passt, so dass das Betriebssystem die Festplatte benutzen muss.

In diesem Fall sagen wir, dass das Problem für die *external memory* Anforderungen formuliert ist.

Ein anderes Problem für klassische Verfahren ist der schnell wachsende Parallelismus. Die Anzahl an Prozessoren variiert von einigen wenigen (bis acht) in einem üblichen Heimcomputer bis zu Hunderten in einem Grafikprozessor (GPU).

Die klassischen Verfahren sind für diese Anforderungen oft ineffizient oder sogar nicht anwendbar.

Die *worst case* Schranken im RAM Model garantieren nicht, dass der Algorithmus eine bessere Laufzeit für praktische Eingaben erzielt. Der Grund dafür ist, dass *worst case* Eingaben

in Praxis selten aufkommen. Deswegen sind theoretisch schlechtere Algorithmen oft besser für praktische Anwendungen. Desweiteren können Heuristiken einen positiven Einfluss auf die Laufzeit haben. In dieser Arbeit versuchen wir die Lücke zwischen klassischen *worst case* Laufzeitschranken und den Anforderungen, die man bei praktischen Anwendungen trifft, wie Eingabegröße, beschränkter Hauptspeicher oder Parallelismus, zu schließen.