Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

Michal Vaner

Cache-oblivious Algorithms

Department of Applied Mathematics

Supervisor of the master thesis:	Mgr. Martin Mareš Ph.D.
Study programme:	Computer Science
Specialization:	Discrete Models And Algorithms

Prague 2012

I would like to thank my supervisor Martin Mareš for his invaluable advice, questions and proof-reading the text. My parents and girlfriend Lucka also deserve my gratitude for their support and patience in times when I was busy working on the thesis. I should thank the Department of Applied Mathematics for providing me with the computers for running the benchmarks. And I shouldn't forget to mention my thanks to the computers in question (Kamenolom and Drahokam from the department and my Hydra), which spent a long time performing tedious computations for me. Finally, the work would not be possible without the good open source software available on the Internet, like `latex` or `git` and more.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague on 2012-04-10

Název práce: Cache-oblivious Algoritmy

Autor: Bc. Michal Vaner

Katedra: Katedra aplikované matematiky

Vedoucí diplomové práce: Mgr. Martin Mareš, Ph.D., Katedra aplikované matematiky

Abstrakt: V této práci se zabýváme výpočetním modelem cache-oblivious algoritmů, který je inspirovaný chováním pamět̆ové hierarchie současných počítačů. V tomto modelu studujeme některé grafové algoritmy a techniky jejich návrhu. Zabýváme se zejména procházením grafu, rozkladem na komponenty souvislosti a hledáni v inkluzi maximálního párování. Taktéž zkoumáme třídění a násobení matic jako podproblémy mnohých grafových algoritmů. Mimo dříve známých algoritmů uvádíme i několik nových. Jejich efektivitu posuzujeme jak asymptoticky, tak experimentálně na reálném hardwaru a srovnáváme je s klasickými algoritmy.

Klíčová slova: Algoritmus, cache-oblivious, výpočetní model, srovnávací test

Title: Cache-oblivious Algorithms

Author: Bc. Michal Vaner

Department: Department of Applied Mathematics

Supervisor: Mgr. Martin Mareš, Ph.D., Department of Applied Mathematics

Abstract: In this work, we study the cache-oblivious computation model, which is inspired by the behaviour of the memory hierarchy of current computers. We study several graph algorithms and techniques of their design in this model. We consider graph searching, identifying connected components and computing maximal matching. We also study sorting and matrix multiplication as subproblems of many graph algorithms. In addition to previously known algorithms, we present several new ones. We study their efficiency both by the means of asymptotic complexity and by benchmarking them on real hardware and we compare them with classical algorithms.

Keywords: Algorithm, cache-oblivious, computation model, benchmark

Contents

1. Introduction

In the recent years, the processor speed increased vastly. Manufacturing memory with corresponding speeds would be very expensive if not impossible. Therefore memory caches were inserted between the main memory and the processor. These caches are faster than the main memory, but also smaller. A cache keeps copy of some of the data from the main memory which were accessed recently. This is because many algorithms often access the same data many times and the cache then can provide it faster.

While the effect of cache is generally good on any algorithm, the speedup is different for different algorithms. Some patterns of memory accesses utilize the cache better than others. A typical example of this effect is traversing elements of a matrix. As we show below, traversing a row of a square matrix is much faster than traversing its column.

Algorithms that are aware of the cache and use it optimally were created. Programs using these algorithms are however written for specific parameters of the cache, which makes them less portable. Fortunately, it was discovered [1] that there are algorithms which use the cache optimally up to multiplicative constants without knowledge of the cache parameters. Such algorithms are called cache-oblivious. Their advantage is that they work optimally with caches of any size. Sometimes a simple modification of a classical algorithm makes it cache-oblivious, many algorithms are based on less straightforward ideas.

We study cache-oblivious algorithms for some graph problems and their common subproblems (sorting and matrix operations). Such algorithms are interesting not only on their own, but many problems from other areas of computer science can be reduced to them.

We analyze the algorithms in theory, by estimating their complexity – run time and space complexity as well as efficiency of their cache utilization. But asymptotic complexity hides many constants which might have a substantial effect on the real speed of the algorithm. We therefore benchmark the algorithms on real hardware in addition to the estimates. This helps us compare the algorithms to each other and also to determine how realistic the cache-oblivious model is.

Apart from previously known algorithms, we propose several new ones, most notably an algorithm for determining connected components in Section 9.2 and maximal matching in Section 10.3. We also examine few hybrid algorithms which combine cache-oblivious algorithms with classical ones. This helps, for example, to eliminate high overhead of recursive algorithms on small problems.

2. Conventions and used notation

There are many generally used notations when it comes to algorithms and bounds of their complexity. This is description of the one used in this text.

We use the well-known symbols $\mathcal{O}$, Θ and Ω for denoting asymptotic estimates of functions. As these are, in fact, sets of functions (e.g., $\mathcal{O}(n)$ is set of all functions growing at most linearly in n), we use the set notation for relations between them. Therefore $f \in \mathcal{O}(g)$ means that f grows at most as fast as g in asymptotic sense and $\mathcal{O}(f) \subseteq \mathcal{O}(g)$ means that all functions growing at most as fast as f are growing at most as fast as g as well.

Generally, n denotes the number of elements or more generally, the size of the given problem.

2.1 Graphs

When we talk about graphs, we usually mean combinatorial graphs – some ***vertices*** (in other works sometimes called nodes) connected by ***edges***. Unless explicitly mentioned, we work with undirected graphs – the edges have no direction. We implicitly assume that all graphs are finite and simple (there are no loops nor parallel edges).

Therefore, a graph is an ordered pair $G = (V, E)$, where V is the set of the vertices and $E \subseteq \binom{V}{2}$ are the edges (a subset of all unordered pairs of the vertices). When we estimate algorithm bounds and there's no room for confusion, we use V and E for sizes of the respective sets, too. That means, instead of writing $\mathcal{O}(|V|)$ (functions growing at most linearly with the number of vertices), we write $\mathcal{O}(V)$ (and similarly for edges).

2.2 Cache and memory

We assume a two-level memory hierarchy consisting of ***external*** and ***internal*** memory (there are often more levels than two in practice, but it would needlessly complicate matters). The internal memory is limited, while the external one is assumed to be infinite. The algorithm works directly with the data in internal memory and moves the data between internal and external memory. However, it is not possible to move the data in arbitrary way.

Both the internal and external memory are split into non-overlapping areas of a fixed size, called ***pages***. It is possible to move the data by whole pages only – load a single page from anywhere in the external memory to any page of the internal memory. But if the page in the internal memory is not free (there are data in it), it must be freed first by storing the data to the page in the external memory where it was originally taken from. This exchange of data in the page of internal memory is called ***memory transfer***.

This terminology is taken from the I/O model (see Section 3.3). It suggests having a RAM as internal memory and a disk as external one. While the cache-oblivious model is applicable for such situation, it is more common to consider the CPU cache hierarchy. Therefore, sometimes, we interchange the terminology, we call the internal memory ***cache***, the external memory just ***memory*** or ***main memory*** and the page is also called a ***cache line***.

The size of the internal memory is denoted by M and the size of a single page is L. Therefore, there are M/L pages in the internal memory.

We assume the cache is ***tall***, which means $M/L \gg L$ – there are more cache lines than how the cache line is wide. This is realistic, as today processors have cache line sizes in order of tens of bytes (most common is 64B) and caches from tens of kilobytes to megabytes (depending on processor model and level of cache when multiple are present). This gives the number of lines in orders of thousands or more.

3. Computational models

Computer science uses asymptotic complexity to distinguish how good an algorithm is. The principles of asymptotic complexity are generally known and therefore we will not describe them here.

We usually estimate three properties of an algorithm we study. If not noted explicitly, the estimate is for the worst-case scenario.

Time complexity is the number of instructions executed during the runtime of the algorithm. This is the time measured in the RAM model (see Section 3.2).

Space complexity is number of memory cells needed to run the algorithm. We count the cells of external memory. We assume we use the whole internal memory, and we want to know how large the external memory needs to be (or how large size of the infinite external memory is used).

Number of memory transfers is just what it says – number of pages exchanged between the internal and external memory during the runtime of the algorithm.

When we evaluate space complexity, we have no serious problems, since it is generally agreed on we count the number of memory cells used during the computation. This can be easily formalised in a way in which it corresponds to the intuitive understanding.

The time complexity is more problematic, since there are multiple definitions of elementary operations (the instructions taking unit time). Since we will be comparing algorithms using multiple criteria, we use several models, each with slightly different elementary operations.

Note that all our models are Turing-complete (they have the same strength as a Turing machine), the models will differ only in how expensive an algorithm is in each one of them. We will also ignore questions regarding where the program is stored, if it is hardcoded in the machine (like in the case of a Turing machine) or stored in the same memory as data (Von Neumann architecture), since it would only complicate matters without providing any interesting differences.

When computing the complexity, we'll not do it in completely formal manner (listing each used instruction). We are interested in the asymptotic estimates only and we can afford little bit of intuition in the estimates. But we need to know what the estimates mean, therefore we list computational models interesting to the work.

3.1 Real hardware

This is not really a model, this is what we would like our programs to run fast on. We could measure it in processor ticks or other time units like seconds. However, we don't analyse the complexity for real hardware for several reasons:

The most obvious one is each computer is different. They have instructions that take different number of ticks on different processors (or completely different instructions), they have memories of different speeds and the speed of program is influenced by almost every other part of the computer.

The other reason is that even if we fixed one computer as given, it would be impossible to analyse. Even single instruction can take different times to run each time. One aspect of this is that processors run multiple instructions at once and they share the computation units inside. The operands might be in registers, or they might need to be loaded. If they are loaded, the time it takes depends on if it is stored in cache, on which level of cache hierarchy, or if the data come from main memory, if the main memory has the corresponding memory line addressed, or if other part of the computer blocks the bus. This complexity can be demonstrated for example by the Intel Architecture Optimization Reference Manual [2].

Therefore we either try to benchmark the implementations of algorithms, which is inaccurate and chooses only some finite set of infinite number of possible inputs (see Chapter 4). The other possibility is to use some simplified model, which gives some estimate of speed, but might ignore some aspects of the computer or turn out to be unrealistic in some of its aspects.

3.2 RAM

RAM, or Random Access Machine, is the most commonly used model for estimating the time complexity and it is usually the abstraction used by procedural languages.

The computer in this model consists of a processor and a main memory. The main memory is infinite array of cells, each containing one integer number. The array is indexed also by integers. The processor executes instructions. Each instruction takes a fixed number of memory cells from the main memory, uses them as operands and stores the result back to the main memory. The location of each operand or result can be either a constant encoded in the program or indexed by a number stored in another memory cell. When indexing by an other cell, the position of this cell must be a constant encoded in the program (depth of indirection can be at most 1).

Each instruction takes unit time, which means that all memory locations are equivalent in their access time (and we completely ignore any of the irregularities of real hardware). This is not unrealistic, though, since a real computer has an upper bound of how long a single instruction takes.

There are several versions of this model differing in the allowed set of instructions. We'll try to stay close to real hardware, and we can say that the unit operations are exactly the primitive operations of the `C` language (which include arithmetics, logical operations, data copies and conditionals).

Some versions of this model include a finite number of registers inside the processor. It is easy to see that this makes no significant difference, since we can reserve a memory cell for each register and use it's index whenever we would like to use the register.

This model has some problems in being too strong. It is possible to compress the whole input into a constant number of memory cells and perform the whole computation on it, usually making it possible to finish the computation in constant time. Some versions of the RAM model try to solve it by limiting the maximal number which can be stored in single cell (which is close to real hardware, but limits the theoretical power to finite automata) or by charging time logarithmic in the size of the operands to the instruction. We'll simply avoid doing the compressions or any other kind of obviously unrealistic misuses of the model. The same problem applies to all of the following models.

3.3 I/O or Cache-aware

This model is inspired by a situation when the internal memory of computer is not sufficient and data must be stored in external memory (today usually magnetic discs or SSD devices).

We have a processor and internal memory, as in RAM. But the internal memory is finite and divided into pages. Each page contains a fixed number of continuous memory cells. To store the data, we have an infinite external memory.

The model allows using of all the instructions in RAM for free. But these instructions can work with the finite internal memory only. To access the external memory, we add two new instructions: ***store*** and ***load***. These transfer a whole page of from internal memory to external memory or vice versa.

We count only the number of loads and stores. This corresponds to the fact that external memory is several orders of magnitude slower than the internal memory on real machines, therefore the whole computation time is dominated by the communication with external memory.

The model considers all external memory locations equivalent in their access time. This is seemingly not true for hard discs, since their seek times depend on the distance between previous access and the current one. But if we use pages large enough, the time to read or write the page (which is located consecutively on the medium) becomes comparatively long as the maximum seek time. This makes it possible to estimate the whole load and store by a constant time. Further justification of this can be found in [3].

The complexity in this model will be a function of the size of the internal memory and page size, as well as the properties of the program's input.

3.4 Cache-oblivious

This is the model which will have most attention in this work. It was introduced in [1].

The model computer has the same parts as in the I/O model. However, the program running on it has no knowledge about the size of internal memory or the size of pages. The program sees only the infinite array of memory cells as in the RAM model (therefore we could say Cache-oblivious and RAM models have the same instruction set).

To make it possible, there's an abstraction layer. The internal memory is used only as a temporary cache. When the program accesses a memory location, the cell may be currently in the internal memory in which case it is simply used. If it isn't in the internal memory, the page containing it must be loaded from the external memory first. If there are no free pages in the internal memory, some page must be transferred to the external memory first (this is called eviction) to make room for it. The abstraction layer chooses optimally which page to evict (it is easy to see that the optimal page to evict is in fact the one which will be needed furthest in the future).

All this happens without the program knowing, while in the I/O model the program moves data explicitly. This is the reason to call the models Cache-aware and Cache-oblivious.

Similarly to the I/O model, we count only the number of pages transferred between internal and external memory. We analyse it relative to the properties of internal memory – while the program itself can't use the sizes, the analysis can.

We sometimes assume some minimal size of the internal memory, either in the number of memory cells or number of pages. In all cases, this will be constant, therefore we can still call the algorithm cache-oblivious (but with some kind of minimal requirements on the hardware it'll run on).

The use of the optimal paging strategy seems unrealistic. However simple LRU strategy (the least-recently used page is evicted) behaves asymptotically the same as the optimal one[1]. The optimal one is usually easier to analyse, especially when an upper bound suffices – we describe any strategy of how the pages are evicted and assume it's not better than the optimal one.

The motivation for this model comes from several areas. The obvious one is the same as with I/O model, when we have too much data to fit to the internal memory. But writing (or designing) programs for I/O is usually complicated, so we can provide a generic abstraction layer to take care of memory transfers and keep the core of the program simple. Something similar is done by many operating systems in the form of virtual memory (with the aid of hardware). If the computer runs out of internal

[1] More precisely, the number of cache misses of LRU is within a constant factor of the optimal strategy with constant-factor smaller cache, as shown by [4].

memory, it stores some data on disk. This is usually called swapping (although this is only inexact an approximation of LRU).

The other motivation is modelling processor caches. Today processors have hierarchy of transparent caches for the data in the memory with various paging strategies (usually based on LRU, but some are enriched by heuristics to provide better performance in usual workloads). The caches are much faster than the memory. If we want to write programs that use the caches effectively, we would be interested in this model, since the caches differ from processor to processor (and from level to level in the hierarchy) and the cache can't be manipulated directly. Also, it can be shown that if an algorithm behaves optimally in a single-level cache hierarchy, it behaves optimally in a multi-level cache hierarchy as well, as proven in [1].

3.5 Streaming model

The reason for its existence is the same as with the I/O model, but comes from the time when computers used magnetic tapes as their external memory. Considering all positions on a tape as equally fast to access is clearly unrealistic.

In this model, we have a finite internal memory and a finite number of streams. Each stream is a possibly infinite sequence of integers. We can read or write data at the current position in stream. Furthermore, we can move one position to the left or right in the stream. It is enough to count the number of moves between positions (we can cache the current cell if we would be reading or writing it multiple times between the moves).

As this model is very limited compared to what today computer can provide, algorithms are no longer directly designed for it. But if an algorithm happens to be fast in this model, it can mean it will perform well on real hardware. Hard disks are faster if data are read and written sequentially. The same holds for memory systems (sequential access is good for both cache evict and prefetch heuristics and for memory modules with matrix indexing – this is explained by Ulrich Drepper [5]).

3.6 Parallel modifications

This is not a model of itself. It is more a modification to other models. The computer has multiple processors. In the PRAM model (parallel extension of the RAM model), they have one main memory each and another main memory shared between all of them (which can be used for communication). In the other models, each processor has its own internal memory and the external memory is shared.

We assume it is possible for multiple processors to read the same memory location at the same time, but they never write to the same location (or page, we assume they have some kind of synchronization to accomplish this) at once. This is known as CREW (Concurrent Read Exclusive Write). Usually only the RAM model is considered in the parallel form called PRAM, but it is possible to extend it naturally to other models.

We can either sum the number of transfers/operations of all processors or take the maximum of them, assuming the one will take the longest to run, therefore the real runtime will depend on it (or, taking the critical path, if they wait for each other). But we'll not generally compute the complexity for parallel models, only note that a given algorithm can be written so it benefits from the parallelism.

The motivation here is to model multi-processor and multi-core computers.

4. Benchmarks

In addition to asymptotic analysis of complexity of our algorithms, we have implemented them and measured their performance on real hardware. To do so, a benchmark framework was created. To understand what was actually measured, here's a description of the framework.

Some of the results are used through the text. More are included in Appendix A. The code is available on the attached CD or online at `http://vorner.ucw.cz/en/thesis/`.

4.1 Pointers and benchmark types

There are several things to measure, each requiring different instrumentation inside the running algorithm. Including all would slow the algorithm down in cases it is not needed, skewing the results. Writing several versions of each algorithm would be repetitive and it would be possible for the versions to differ.

Therefore the algorithms are implemented as `C++` templates which allow replacing classes used as pointers. The pointer class actually used performs the needed instrumentation. This allows building a binary with the correct instrumentation just by specifying the corresponding pointer class.

4.1.1 Measuring runtime

To measure runtime of the whole algorithm, we need no instrumentation – it would actually seriously skew the results. Therefore the pointer class used is just a thin wrapper around the real pointer. The compiler optimises all the calls out, changing it to the primitive CPU instructions, which produces code equivalent or close to an algorithm written in the usual way, with real pointers (or arrays).

It is then possible to simply run the algorithm and compute the runtime by comparing the time before starting it and after it finishes. Sometimes the same test case was run multiple times to minimise random effects of operation system and time measurement inaccuracies. In such case, the total time is reported in the graphs.

It is also possible to run multithreaded. This is not possible with the other benchmark types, mostly because of some low-level technical issues which were not attempted to be solved.

The benchmarks were run on AMD Phenom II CPU at 3.3GHz with 6MB of L3 cache and 6 cores. There were 4 memory banks at 1300MHz frequency. The used compiler was `gcc 4.5` with optimisation flags `-O3 -march-native`. It was run on `Linux 3.2`.

4.1.2 Measuring LRU cache of disk

The goal here is to simulate the case where internal memory is a cache for data stored on a disk. There are two things needed to make the cache work, as suggested in Section 3.4.

The first one is knowing when and what page needs to be loaded. Having a check in each access to a pointer would be needlessly slow. But when the memory is allocated by `mmap` system call, only the address space is reserved. The physical pages of memory are assigned only when first accessed[1].

Now, we can use `mprotect` to make the address space we reserved inaccessible. Whenever the algorithm tries to access such inaccessible page, a segmentation fault signal is generated. If we register a handler for the signal, we can use it to make the page accessible again and load data into it from disk (using direct I/O, to make sure the data aren't kept in some OS cache, which would make the results useless). When evicting a page, we `munmap` it (so the physical page is freed), then `mmap` and `mprotect` the page again, so we can check when it should be loaded.

The other needed thing is to know which page should be evicted. We inject code to all operations with pointers. Whenever data is accessed through a pointer, the access is stored in an auxiliary buffer. The data in buffer is used to compute the page for eviction.

This allows for specifying both the size of the cache and the page (provided it is multiple of the hardware page size) and makes measuring the time the algorithm needs to run when data are loaded and stored on disk. However, the algorithm runs slightly slower, because of the writing to buffer and computation of LRU. It is assumed that this slowdown will be less significant than the time needed for actually storing and loading data.

However, this method has a problem. The data are stored on unpredictable place on the disk, so the results can differ between runs of the benchmark. Empirical observation suggests that this difference is not significant, but the results shouldn't be taken too strictly.

4.1.3 Simulating cache behaviour

This is more theoretical measurement than the previous ones. The goal is to count how many cache misses would happen on a provided cache, if the cache was at the sole use of the algorithm.

To do this, the pointers are modified to output address and length of each access. These

[1] While it doesn't seem to be guaranteed by the POSIX standard, it was experimentally confirmed to be the case on the used Linux system.

addresses and lengths are output through a pipe. A different program reads the data and simulates behaviour of caches with given properties, simply by keeping the list of pages in the cache and information about accesses so it can decide what pages should be evicted. By simulating multiple caches of different sizes or with lines of different lengths we can study both how an algorithm reacts to changes of input size as well as changes to the cache properties.

This is not completely accurate. This doesn't take the local variables in functions into account. But we can assume they would never be evicted from the cache, so the only inaccuracy is they don't take space in the cache, which is negligible. For this to be true, we unroll all recursion and use an explicit stack. This way, there's only a constant number of local variables (or global ones) and everything else is allocated on the heap and accessed through the instrumented pointers.

4.2 Inputs

The algorithms need to be tested on some inputs and the type of an input can influence the results quite substantially. Therefore the inputs are described here.

4.2.1 Sorting inputs

There are three kinds of inputs we have used for benchmarking of the sorting algorithms (in Chapter 6):

- All zeroes. This input is used to make sure that the algorithm works on unusual input.
- Random input. Each value in the input is generated randomly with uniform distribution. This is the method used for graphs and results through the text, but the Apendix A.1 contains other inputs as well.
- Already sorted input. This input checks if the algorithm takes any benefit from the sorted state.

Also, various data types are used (the results differ slightly depending on the size of a single item).

4.2.2 Matrix inputs

The matrix multiplication (in Chapter 7) is measured using random inputs – cells of each of the input matrices are filled by independent random integer with uniform distribution.

4.2.3 Graph inputs

All the inputs for graph algorithms (in Chapters 8, 9 and 10) are graphs with at least a little bit of randomness. However, it is slightly problematic to say what a random graph is. They are listed with the names as they appear in the text (note that this is not the definition of graph property, just a name of the input type).

- ***Dense*** graph here is one where each possible edge is present with a fixed probability, independently of other edges. The probability used in the benchmarks is 1/10,therefore the expected number of edges is $\frac{|V|\cdot(|V|-1)}{20}$.
- ***Sparse*** graph is a graph generated by adding edges until it has $c \cdot |V|$ edges (or if the graph has too few vertices, all possible edges). The edge is chosen by taking two random vertices independently with uniform probability. The edge is discarded if it is already present or if it is a loop. The constant c is 10 unless explicitly stated differently.
- ***Tree-like*** graph is like the above, but with $c = 1$.
- A ***triangulation*** is generated in this way: We start with a triangle. The triangle divides the plane into two faces. We pick one of the existing triangular faces, put a new vertex into it and connect to all three vertices on the border of the face. This breaks the face into three new ones. We again pick a triangular face (not necessarily one of the new ones, but one that was not yet broken) and repeat, until we have enough vertices. Each edge is preserved with probability of $\frac{9}{10}$.

All the generated graphs are simple – they have no loops and no parallel edges. The triangulation one is guaranteed to be planar. Planar or mostly planar graphs are common in practice, they can represent networks, maps and similar (which are often large), so we are interested in how the algorithms perform on them. The treelike and dense graphs are used as extremes in the ratio between edge and vertex numbers.

5. Common techniques

We are going to describe or design several algorithms in the following chapters. Therefore we need to known some common techniques and tricks of designing and analysing them.

5.1 Algorithm approaches

Let us first look at how the cache-oblivious algorithms are usually built. That means there can be other techniques or tricks in some algorithms, but these are quite common.

5.1.1 Scans

The most basic design approach is to have data in an array and access elements sequentially, from left to right or in the other direction. It often happens we need to access all the elements, for example if we want to compute the sum of all of them. In such case, this approach is optimal. We need to access each page of the array at least once, as we need to access an element in each of the pages. This means we have $\Omega(n/L)$ memory transfers in case the array is not present in cache. And the scanning algorithm does $\mathcal{O}(n/L)$ transfers, as each page of the array is processed as a whole and never touched again when a different page is accessed. We can keep this "active" page in the internal memory.

As the scan needs only a single page, there can be a constant number of scans running in parallel (meaning that one array is not scanned at once, but part of it is scanned, then part of another array, then continuing the first one, and so on).

Scanning a constant number of arrays in parallel is also a common technique, and, while simple, it is quite powerful.

One example would be merging two sorted arrays to one. We look at the beginnings of the arrays and compare them. The smaller one is output and we move forward in that array. We again compare the values in the arrays and repeat until we run out of the inputs. As we move only forward both in the input and output arrays, this is equivalent to scanning the three arrays in parallel.

Another example would be finding out what values are present both of two sorted arrays. If the beginnings of the arrays are equal, we found a value which is in both. If not, we skip the smaller one, because we know it is not present in the other array, due to it being sorted. Then we try to compare them again.

Scanning arrays $A_1, \ldots, A_n$ in parallel if n is a constant clearly takes $\mathcal{O}\left(\frac{\Sigma_{i=1}^{n} A_i}{L}\right)$ memory transfers.

Note: We often need to access the elements in some specific order. Therefore many algorithms combine sorts and scans.

Note: Scan is also optimal in the streaming model. Algorithms using only scans and sorts are expected to perform well in the model. There are algorithms designed for the streaming model. We can find some sorting algorithms for the streaming model for example in The Art of Computer Programming, Volume 3 [6], Chapter 5.4.

5.1.2 Divide and conquer

Divide and conquer is a well-known programming technique. It means taking the problem, and by applying some relatively primitive transformation turning it into few smaller problems. The smaller problems are then solved recursively and their results are aggregated to form the result. The recursion stops when the problem turns small or simple enough to solve it trivially.

The advantages for the cache-oblivious model are twofold. First, it is usually possible to design the splitting to smaller tasks and aggregation in a cache-oblivious friendly way, using only sorts and scans. Secondly, sometimes it is possible to prove that the bottom levels of recursion will be free of memory transfers. Since the tasks are getting smaller and smaller, they'd become smaller that the whole cache.

5.1.3 Dividing vertices to conquer edges

This is the Divide and conquer approach applied in a less traditional way. The goal is to split edges in small enough groups so they can be handled (the vertices are not important for many algorithms, they simply "pin the ends of edges to the surface so they don't curl up").

We assume the vertices are identified by integer numbers $0, \dots, |V| - 1$ and the vertices are pairs containing both their endpoint vertex indices.

But instead of taking the array of edges and dividing it into two halves, we take the vertices and form two groups of them. We will call them ***left*** and ***right***. This will create three groups of edges. The ***left*** and ***right*** group of edges have both endpoints in the left or right group of vertices respectively. The edges in the ***middle*** group connect vertices of both groups, so each edge has one left and one right endpoint.

But there's a problem. When we get deeper in the recursion, the vertex groups get smaller and smaller. However, if we look at the graph defined by the middle edge group, we see that the graph potentially contains all the vertices. With a bad choice of how to split the vertices – for example taking the halves with smaller indices and bigger indices, the middle group could stay the same size across any number of levels of recursion. We need a rule how to split the vertices to make sure the number of edges in each group is bounded.

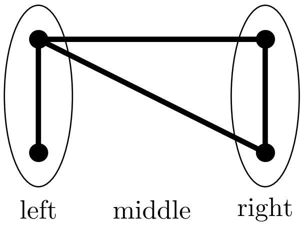

Figure 5.1: Vertex and edge groups

We choose following rule to split the vertices. If we are i levels deep in the recursion, we look at the value of the i^{th} bit of each vertex. The value of the bit decides which group the vertex goes to.

As there are $\log_2 |V|$ bits in the vertex indices, the recursion must after so many levels. This does not give groups of edges of size at most 1, but it provides groups with special properties we can use, as shown bellow in Lemma 2.

Note: We don't need the vertices explicitly, the splitting of edges can be done by scanning through them and just looking at their endpoint numbers.

5.1.4 Van Emde-Boas layout

We want to store a complete binary tree (the tree doesn't have holes and all the branches are of the same length) in memory. We want to traverse it from root to one of the leaves. But the usual approach with pointers is not very cache friendly, so we show a different layout.

We notice that storing data in an array is usually good for caches. The first idea for a layout in an array would be to place it by layers. First, the root is stored. Then the two sons, then the four nodes in depth 2, etc – similar to how an ordinary binary heap is usually stored. However, from some point the levels become longer than a page, so the sons are in a different page than the parent. The best thing we could do would be storing the top of the tree in the cache, but this would still be $\Theta(\max 0, \log n - \log M)$ memory transfers.

Therefore we need a more sophisticated layout. The idea is inspired by the van Emde-Boas tree [7], however, the data structure is different. It can be found for example in Cache-oblivious B-trees [8].

Assume we have a complete binary tree. It would work for different trees as well, but it would be harder to explain. We also ignore rounding for simplicity. There are two reasons why we can do it. One of them is this is a general design approach, not a concrete data structure. Another is it is not important[1].

The tree has approximately $\log_2 n$ layers. We cut the tree in the depth of $\frac{\log n}{2}$ into an upper and lower part. The upper part has $\Theta(\sqrt{n})$ leaves and the bottom part is composed of $\Theta(\sqrt{n})$ trees. All these trees are of approximately the same size.

We store the top tree, followed by the bottom trees into the array. Each of these trees is laid out in the same manner recursively. The layout is illustrated in the Figure 5.2.

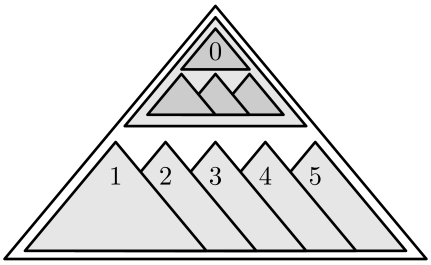

Figure 5.2: The recursive layout (numbers mean order in the array)

The advantage is, as we are splitting the trees into smaller and smaller ones, we eventually get to trees small enough to fit whole in a constant number of pages. This constant number of pages fit into the internal memory, therefore we need to load each of them at most once when finding our path through such a tree. That means we can go from top to bottom of the tree with $\mathcal{O}(1)$ memory transfers.

But the tree has $\Theta(L)$ nodes, therefore it is $\Theta(\log L)$ levels deep. We pay $\mathcal{O}(1)$ memory transfers for each $\Theta(\log L)$ layers. The total cost of traversal of the whole large tree is therefore $\Theta\left(\frac{\log n}{\log L}\right)$.

5.2 Common lemmata

There are several lemmata used through more than one chapter, so they are listed here.

A function f satisfying $f(n_1) + f(n_2) \geq f(n_1 + n_2)$ is said to be ***sub-additive***.

Lemma 1 (Number of memory accesses of a recursive algorithm)**.** *We have a recursive algorithm which splits a task into a constant-bounded number of subtasks of total size no bigger than the original size. The input is of size n. The depth of recursion is $\mathcal{O}(g(n))$*

[1] As mentioned in the original article [8], correct rounding is important for dynamic data structures, but we use the layout in static data structures only.

for some non-decreasing function g and the cost of local work of a task of size m is $\mathcal{O}(f(m))$, where f is a sub-additive function. Then the algorithm needs $\mathcal{O}\left(g\left(n\right)\cdot f\left(n\right)\right)$ memory accesses.

Proof. We would like to take a sum of costs of all the tasks on a single level and multiply it by the number of levels. This would be possible for the sub-additive function f.

The problem is, the $f(n)$ could be smaller than 1 for small enough n. But a task of such size would still require a full memory transfer. Example of this could be scanning an array. The cost is $\mathcal{O}(n/L)$, but if $n < L$, a whole page (or two, in fact, if the data are placed on the border of pages) would need to be loaded. These memory transfers are "hidden" in the $\mathcal{O}$ notation and are not really interesting most of the time. However, summing the cost over many such small tasks would not be correct unless we can prove the needed part of the page would be already present in the cache.

We fix a small constant c. We'll define three kinds of tasks:

- A ***large*** task is such a task of size m, where $f(m) \geq c$. Therefore the cost of the task itself is large enough to amortize the costs of partial pages.
- Then we'll have a ***small*** task. This is a task whose computation (including subtasks) fits into the internal memory. Therefore if the whole input was read into it before, it would cost no memory transfers. Or, if we look at it differently, the only cost paid for this task is to read the input in.
- Last, a ***tiny*** task is one where the parent of the task is small. The tiny task is for free, since it is by definition small and it is already read into the internal memory by the parent.

There's no problem with large tasks, nor with tiny. There's at most one task which is small but not tiny on each path from the root of the recursion tree to a leaf. We claim that this small task has a large parent and we'll amortize the constant number of transfers to the parent (we can, since each large parent has a constant-bounded number of children).

If there wasn't a large parent, there would have to be a task that is neither small nor large. Such task would not fit into the cache completely. Therefore $f(m) > M/L$, as it isn't small. But $f(m) < c$, as it is not large.

However, as we noted already, we assume that the cache is tall. That means specially that it has more than constant number of cache lines, therefore a task that does not fit must need more than constant number of transfers. Therefore a task must be either large or small. □

Lemma 2 (Dividing vertices)**.** *If we have a recursive algorithm dividing the vertices to conquer edges (from Section 5.1.3), no two edges in the same task on the bottom level of recursion share a common vertex.*

Proof. We'll prove this by a contradiction. Assume that there are two edges, e_1, e_2 that share a common vertex v. Since the graph is simple, there are no parallel edges and the other ends of the edges must be distinct, let us call them u and w.

On each level of the recursion, we examined one of the bits in the IDs of vertices. Furthermore, on each level, both edges ended up in the same group – otherwise they would not be in the same task at the bottom of recursion.

We'll denote the i^{th} bit of the ID of the vertex x as $x[i]$.

We look at a single level i, therefore examining the i^{th} bit. If both the edges ended in ether left or right group, the $u[i] = v[i]$ and $w[i] = v[i]$. This means $u[i] = w[i]$.

If they ended in the middle group, $u[i] \neq v[i]$ and $w[i] \neq v[i]$. However, the value of a bit can only be 0 or 1, therefore $u[i] = w[i]$.

As the edges always ended up in the same group, the u and w are equal in all their bits, therefore $u = w$. This is contradiction. □

6. Sorting

This work concentrates mostly on graph algorithms. However, as shown below, many of the algorithms use sorting as their subroutine and this sorting influences their bounds. Therefore we analyze sorting algorithms first, including the number of memory transfers.

First of all, we analyze several well-known sorting algorithms, then we show an algorithm designed for the cache-oblivious model.

6.1 Merge sort

Assume for a while the input array is of size $n = 2^i$. It'll simplify the description and we'll show how to get rid of it later.

This algorithm first divides the input into n blocks each of them consisting of a single element, so all of them are trivially sorted. Then, in each iteration, it takes two sorted blocks of the same size and ***merges*** them together. The way the blocks are chosen is described below, as there are multiple possibilities.

Merging them means we look at the first element of each block. We takes the smaller one, output it and advance in the respective block to the next element. We then repeat this action until we get to the end of one of the blocks. Then we append the rest of the other one to the result. We can notice the result will be also sorted.

When we have only one block, the array is sorted.

If we want to sort an array of a different size than 2^i, we can do a trick. We extend the array with virtual elements which are larger than anything, so the input has the correct size. This will make it at most twice larger, therefore it won't hurt the complexities. After sorting, we can remove these elements from the end of the array.

Furthermore, we notice these would always be at the end of a block. Therefore we don't need to store them, we simply have some smaller blocks and pretend the blocks are of the full size – this only means less work and used memory.

However, we still use the assumption in the analysis.

Theorem 1. *The algorithm runs in $\mathcal{O}(n \cdot \log n)$ RAM time.*

Proof. When we merge two blocks to create a block of size s, it clearly takes $\mathcal{O}(s)$ time. This is $\mathcal{O}(1)$ per an element. Whenever an element takes part in a merge, it gets into twice as large block, because the other block is of the same size. Therefore an element can participate in $\mathcal{O}(\log n)$ merges. So the total is $\mathcal{O}(n \cdot \log n)$. □

Theorem 2. *The algorithm needs $\mathcal{O}(n)$ memory.*

Proof. A merge can be done with linear amount of additional memory. And we can discard the smaller blocks once we merged them (or reuse the memory). □

Theorem 3. *The algorithm would require $\mathcal{O}\left(\frac{n}{L} \cdot \log n\right)$ memory transfers, if there were no partial pages at the beginning or end of the blocks.*

Proof. This is very similar to the proof of theorem 1. We need to notice that a merge can be done in $\mathcal{O}\left(\frac{n}{L}\right)$ memory transfers, as we read the inputs sequentially and write the output sequentially (so we can have a memory page for each of them, holding the active area). □

There are two common ways in which the merge sort is implemented. We look at a merging tree. Each node is a block which existed during the runtime of the algorithm. Sons of block b are the blocks used as inputs to create b.

6.1.1 Layered

First, all the blocks of size 1 are merged together to form blocks of size 2. Then, all these are merged to form blocks of size 4, and so on. We go from bottom up in the merging tree.

The merges on a single layer are performed from the left to the right. We can use a queue for the blocks, each time taking two blocks to merge from the beginning and putting the result to the end of it.

Theorem 4. *The layered version of merge sort requires $\mathcal{O}\left(\frac{n}{L} \cdot \log n\right)$ memory transfers.*

Proof. We can imagine that the whole input is scanned two times in parallel and the whole new level is written sequentially as the output. Therefore the merging on a single layer takes $\mathcal{O}\left(\frac{n}{L}\right)$ memory transfers – there can be only one partial page at the beginning and one at the end of the level.

The queue operations take $\mathcal{O}\left(\frac{q}{L}\right)$ memory transfers for q operations (as we can have a page for the beginning and a page for the end of the queue, effectively having a writing scan and a reading scan). There are $\mathcal{O}(n)$ operations with the queue, therefore this part of the cost is smaller.

Therefore we can use the theorem 3 and get the result. □

6.1.2 Recursive

If we consider the merge tree again, this version goes through it in a DFS order. We call the algorithm on the whole array to obtain the final result. That call would like to merge the two halves of the array together, but they must be sorted first. So it calls the algorithm recursively on the left and right half. Once they return, the final result is obtained by merging their results.

Theorem 5. *The recursive version of merge sort takes* $\mathcal{O}\left(\frac{n}{L} \cdot \max\left(1, \log n - \log M\right)\right)$ *memory transfers.*

Proof. To merge two blocks, we need $\mathcal{O}\left(\frac{s}{L}\right)$ memory transfers if the size of the result is s. This function is sub-additive and the size of the levels is the same through the recursion. There are two children of a task. There are $\log n$ levels of recursion, but the tasks of bottom $\log M$ levels fit into the memory completely, so we can simply count $\log n - \log M$ levels of recursion. We can use the Lemma 1 and get this result.

We need to at least read the input in if $n < M$ at least, which costs n/L – there are enough pages for the whole input, so the order in which we access them does not matter. □

6.2 Quicksort

Another sorting algorithm commonly used on the RAM is Quicksort [9].

While this algorithm doesn't guarantee the optimal RAM bounds in the worst case scenario, it achieves them in the expected case. It is not true in the cache oblivious model. However, as practice shows, it is very fast on real hardware.

The algorithm is defined in recursive way as follows:

1. If the input array is at most 1 element large, it is trivially sorted, so it is returned.
2. Otherwise, an element from the array is selected. We call this element a ***pivot***.
3. The array is split into two parts containing smaller and larger elements than pivot respectively.
4. Each part is sorted recursively.
5. The smaller part, pivot and larger part are concatenated together and returned.

Note: In case the input array may contain duplicate elements, it could happen there are elements equal to the pivot. In such case, we can put these elements in either part or create a third part of elements equal to pivot. The third part would not be sorted recursively.

The usual implementation splits in following way. It scans the input array from both ends in parallel. We first find an element larger than the pivot on the left side. Then we find an element smaller than the pivot on the right side. These two elements are then swapped. This is repeated until the scanning positions meet somewhere in the middle. The place where they meet is the boundary between parts, as now the left side contains only elements smaller than the pivot and the right side the larger ones. Some care must be taken to make sure the pivot ends up in the middle in between and is not included in either of the parts. The position of pivot can, of course, change.

This implementation has three advantages: First, we don't need an auxiliary array, all the data are kept inside the input one. Second, the splitting is done by a scan, which is cache friendly. And third, the concatenation step is not needed, as the parts are already concatenated in the right order.

Theorem 6. *The algorithm can be implemented in a way it needs $\mathcal{O}\left(\log n\right)$ additional memory.*

Proof. All data are kept inside the original array. The only additional memory needed is for the stack for the recursion. Every time we split the array, we can put the task with more elements onto the stack and start working on the smaller part immediately. This ensures that each time we put something onto the stack, the size of current task drops at last twice and it can grow again only when taking something out of the stack. □

Lemma 3. *An element participates in $\mathcal{O}(n)$ splits.*

Proof. As the pivot is not included in either of the parts, the total size drops at last by 1 on each level. Therefore we run out of elements after n splits. □

Lemma 4. *It takes $\Theta\left(n\right)$ time to split array of n elements.*

Proof. In each constant-time step, we move the left position one step to right or the right position one step to left. This makes the positions one element closer to each other. They were at distance $n - 1$ at the beginning so after so many steps they meet. □

Theorem 7. *The worst-case RAM time complexity is $\mathcal{O}\left(n^2\right)$.*

Proof. Directly follows from Lemmata 3 and 4, as each level of recursion contains at most n elements. □

Lemma 5. *It takes $\mathcal{O}\left(n/L\right)$ transfers to split array of n elements.*

Proof. The splitting is equivalent to scanning the array in parallel from left and right at the same time. It is enough to keep one active page on the left side and one active on the right – each page will be transfered into the cache at most once. □

Theorem 8. *The algorithm needs $\mathcal{O}\left(n^2/L\right)$ memory transfers in the worst case.*

Proof. We simply combine Lemmata 1, 3 and 5. The size of levels is decreasing, the n/L function is sub-additive, there are 2 children to a task and there are $\mathcal{O}(n)$ levels. □

Theorem 9. *Quicksort runs in $\mathcal{O}(n \cdot \log n)$ time on average.*

The proof of this well-known fact is somewhat technical and not interesting for the purpose of this work. Therefore we omit it here. A complete analysis including average number of comparisons was presented by Ian Parberry [10].

Theorem 10. *Quicksort requires $\mathcal{O}\left(\frac{n}{L} \cdot \log n\right)$ memory transfers on average.*

Proof. We would like to use Lemma 1 to compute the cost. However, we don't know the correct average recursion depth.

But we can notice the proof of the Lemma uses the recursion depth to compute the total cost of tasks over all the recursion levels only. Fortunately, we can compute the total cost differently. We notice the RAM run time of task sized m is $\Theta(m)$. The whole $\mathcal{O}(n \cdot \log n)$ run time from Theorem 10 is composed only from these tasks. The task sized m needs $\mathcal{O}(m/L)$ memory transfers, as shown in Lemma 5. Therefore the sum of all tasks will be also L times smaller than the RAM time bound, which is $\mathcal{O}\left(\frac{n}{L} \cdot \log n\right)$.

We notice the rest of the proof of Lemma 1 can be applied as the Quicksort algorithm satisfies all the assumptions. □

6.3 Funnel sort

The Funnel sort algorithm is designed directly for cache-oblivious model. It was introduced by H. Prokop [1]. It was later made lazy by Brodal and Fagerberg [11], which makes it slightly simpler to implement.

It is inspired by merge sort, but instead of merging two streams in one step, it uses a device called funnel to merge more than two inputs at once. Overall, it splits the input into $\sqrt[3]{n}$ parts of equal size, sorts each of them recursively, then it builds a funnel on these $\sqrt[3]{n}$ inputs to merge them into output. To solve the small cases where 8, we can sort the input in some other way.

We will ignore rounding errors in the rest of the explanation, as they are not interesting.

6.3.1 Funnel

The crucial part in the algorithm is how the funnel looks like. The purpose is to build a device that merges all its inputs together. If there are k inputs, we'll call it a ***k-funnel***.

The funnel is a tree of mergers – places where two streams are merged together. The mergers are connected by intermediate buffers.

The tree and the buffers are laid out in the van Emde-Boas way (see Section 5.1.4). Therefore a k funnel is composed of $\sqrt{k} + 1$ $\sqrt{k}$-funnels. One is at the top and the outputs of the rest are connected through the buffers to the inputs of the top one. These $\sqrt{k}$-funnels are build of $\sqrt[4]{k}$-funnels, etc. An illustration can be seen in Figure 6.1.

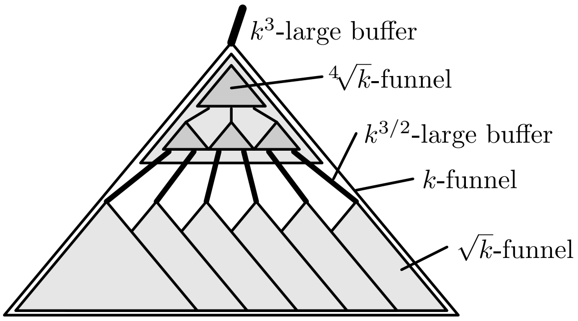

Figure 6.1: k-Funnel consisting of $\sqrt{k}$-funnels

Each funnel is connected to buffers. They are either the buffers containing inputs and a buffer to hold the whole output in case of the biggest funnel, or some intermediate buffers in case of the smaller funnels. The intermediate buffers always connect two k-funnels together. Such buffers are k^3 elements large. This goes in the way of the van Emde-Boas layout, as smaller funnels contain smaller buffers.

This looks like a merge sort, we have a tree of mergers to merge longer and longer streams together to form the final one. However, there are differences. The first one is, the tree in the merge sort was implicit, but it is build explicitly in the form of a funnel here. The other one is the merge sort merges the whole inputs at once before moving to a different streams. We'll be switching between the mergers during the work of a funnel.

Each time a funnel is invoked, it tries to fill its output buffer. This is delegated to the top sub-funnel A recursively, as that one shares the output buffer. The funnel A either finishes because the output buffer is full, in which case we are done. The other possibility is it runs out of data in one of its inputs. The funnel A suspends the work and the funnel B below the buffer is invoked to fill it up. After B finishes, the work of A is resumed.

If the input buffer can not be refilled, because it is the input buffer and there's no funnel below it, or because the funnel below it already run out of all its inputs, the input is no longer used.

6.3.2 Estimates

Theorem 11. *The funnel sort takes $\mathcal{O}(n \cdot \log n)$ RAM run time.*

Proof. For this proof, we will assume that all the mergers exist for the whole life of the algorithm. This is not true, as they are created and destroyed during the recursive calls of the algorithm, but it won't change the run time needed to create them all beforehand and destroy them all after the algorithm finishes.

We look at the complete tree of mergers, connected together from all the funnels. The top-level merger processes n elements. The ones directly below this merger process $n/2$ elements each. Generally, a merger processes half the number of elements processed by the merger above it. This means there are $\mathcal{O}(\log n)$ mergers on the path of each element. As processing a single element in a merger takes constant time, this is $\mathcal{O}(n \cdot \log n)$ over all the elements.

We also need to look at the work needed to construct the funnels. We notice the funnels correspond one-to-one with mergers, as each funnel has a merger at its very top. The tree of mergers have n leaves, therefore it contains $\mathcal{O}(n)$ mergers. Building a funnel costs a constant time, including the buffer. This is smaller than the actual merging. □

Lemma 6. *A k-funnel takes $\Theta(k^2)$ memory.*

Proof. There are some mergers and buffers in the funnel. As each merger has a buffer of at last one element (except the topmost one), we can count only the size of buffers, the sizes of mergers will amortize in this.

There are $\sqrt{k}$ buffers of $k^{\frac{3}{2}}$ elements, together taking k^2 elements. Additionally, we need to count the buffers of the $\sqrt{k}+1$ smaller funnels. Therefore we would need to solve this recurrence:

$$S_k = \left(1+\sqrt{k}\right) \cdot S_{\sqrt{k}} + k^2. \tag{6.1}$$

If we look down to the smallest funnels (e.g., 2-funnels), there are $\mathcal{O}(k)$ of these (as there are clearly $\mathcal{O}(k)$ mergers total), each having a buffer of $\mathcal{O}(1)$ size. The leaves therefore take $\mathcal{O}(k)$ memory, while the root of recursion takes $\Theta(k^2)$. Therefore the root size dominates and the total memory consumed by k-funnel is $\Theta(k^2)$. □

Theorem 12. *The algorithm needs $\mathcal{O}(n)$ additional memory.*

Proof. We need an additional memory for three purposes. One of them is the stack for the recursion and traversing the active funnel. It is easy to see that $\mathcal{O}(\log n)$ stack is enough.

The other reason is, when we are merging by a funnel, we need to preserve the inputs while already writing the output. But if we allocate a temporary buffer of corresponding size for each level of recursion, we are safely within $\mathcal{O}(n)$, as the size of input decreases exponentially within the recursion.

The last kind of memory is the one occupied by the funnels. But we notice there's only one funnel constructed at each time and the biggest one in existence will be the top-level one with $k = \sqrt[3]{n}$ inputs. Therefore it takes $\Theta\left(n^{\frac{2}{3}}\right)$ memory. □

Theorem 13. *The algorithm needs $\mathcal{O}(\frac{n}{L} \cdot \log_{\frac{M}{L}} \frac{n}{L})$ memory transfers. This is optimal.*

First, let us have a look at it intuitively. With merge-sort, we got the lowest levels for free, because the subtasks fit entirely into the cache. Now, we get some free levels in each depth of the recursion, as there are small funnels everywhere. We pay only when we're working with large funnels, but these are built of small funnels as well.

Proof. As shown by Aggarwal and Vitter [3], the lower bound for permuting n elements in the I/O model is the amount stated here. As sorting performs a permutation, it must cause at least as many memory transfers. It is easy to see that a lower bound for the I/O model is also a lower bound for a cache-oblivious model. Therefore when we prove the upper bound, we know it is the best possible result we can get.

The rest of the proof is inspired by the one written by Demaine [12].

First, we'll reformulate the tall cache assumption to $M \geq L^2$ for the rest of the proof.

Lemma 7. *A k-funnel causes $\mathcal{O}\left(\frac{n}{L} \cdot \log_{M/L} \frac{n}{L} + k\right)$ memory transfers during processing of n elements, provided $n \geq k^3$.*

Proof. Consider the biggest l such that the l-funnel fits into $M/4$ memory. As the funnel takes $\Theta(l^2)$ space, the $l < \frac{\sqrt{M}}{2}$. And because of the tall cache assumption, there are at least $\sqrt{M}$ pages in the internal memory, therefore we have enough free pages to keep a page for each of the l input buffers and another page for the output. This means that merging by this funnel is equivalent to scanning the inputs in parallel in the number of cache misses. This is m/L for m elements.

To load the entire l-funnel into internal memory, we pay $\mathcal{O}(l^2/L + l)$ transfers. As the funnel is the biggest fitting, $l \in \Omega(\sqrt[4]{M}) \subseteq \Omega(\sqrt{L})$. From this, we conclude that $\mathcal{O}(l^2/L + l) \subseteq \mathcal{O}(l^3/L)$. Therefore, if $m \geq l^3$, the cost of reading the funnel in is amortized in the cost of merging the elements.

This is in the case the input buffers of the l-funnel contain enough elements. If we run out of an input buffer, we invoke the funnel below it. When we return to the current l-funnel, we might need to read it in again. However, as the input buffer below is at least l^3 elements large, we can amortize the cost of reading the funnel in from those. This means, each block of at least l^3 elements pays for three things. Loading the l-funnel below the buffer when we start producing the block, loading the l-funnel above the buffer after we finished the block and the actual processing of the elements.

This means the amortized cost of moving one element through one l-funnel is $\mathcal{O}(1/L)$.

Because $l \in \Omega(\sqrt[4]{M})$, there are $\mathcal{O}\left(1 + \frac{\log k}{\log M}\right) \subseteq \mathcal{O}(1 + \log_M k)$ l-funnels on the path of an element. So, to merge n elements, we would pay $\mathcal{O}(\frac{n}{L} \cdot \log_M k)$.

We also need to pay a full memory transfer for each of the k input buffers, even in case they contain less than L elements.

The rest of the proof is only formula bashing to get the desired form.

First, $M \in \Omega(L^2)$, therefore $\log M \geq 2 \cdot \log L$. This leads to:

$$\log_{M/L} k = \frac{\log k}{\log M/L} = \frac{\log k}{\log M - \log L} \in \frac{\log k}{\Theta(\log M)} \subseteq \Theta\left(\log_M k\right). \tag{6.2}$$

Therefore we can exchange the base of the logarithm.

If $k \in \Omega(L^2)$, then similarly $\log_M \frac{k}{L} = \log_M k - \log_M L = \Omega(\log_M k)$. On the other hand, if $k \in \mathrm{o}(L^2)$, then $\frac{k}{L} \cdot \log_M k \in \mathrm{o}(L \cdot \log_M L) \in \mathrm{o}(k)$, therefore the $+k$ term dominates.

This together would allow us to write the result as $\mathcal{O}(\frac{n}{L} \cdot \log_{M/L} \frac{k}{L} + k)$. As $n > k$, we can replace k with n in an upper-bound estimate. □

To sort the n elements, we need to split the input into $\sqrt[3]{n}$ blocks, sort each of the blocks and merge them together using the $\sqrt[3]{n}$-funnel. This would give a recurrence:

$$T(n) = \sqrt[3]{n} \cdot T\left(n^{2/3}\right) + O\left(\frac{n}{L} \cdot \log_{M/L} \frac{n}{L} + \sqrt[3]{n}\right). \tag{6.3}$$

If the computation does not fit completely into the internal memory, the $n \in \Omega(M) \subseteq \Omega(L^2)$. In other words, $L \in O(\sqrt{n})$, therefore the n/L is larger than $\sqrt[3]{n}$ and we can forget the $\sqrt[3]{n}$ part for simplicity.

We consider the tasks of size $\Theta(L^2)$ the leaves of recursion. Anything below fits into the internal memory and is therefore for free.

Such task can be read into the internal memory in $\mathcal{O}(L)$ memory transfers and solved inside it, causing no more transfers. As there are $\mathcal{O}(n/L^2)$ such tasks, the total cost over all of them is $\mathcal{O}(n/L)$.

The root of the recurrence dominates. Therefore the total cost is $\mathcal{O}(\frac{n}{L} \cdot \log_{M/L} \frac{n}{L})$. □

To save some typing in the rest of the work, we'll use $\mathrm{sort}(n)$ instead of the function $\frac{n}{L} \cdot \log_{\frac{M}{L}} \frac{n}{L}$.

6.4 Experimental results

After having looked at the theory, let's examine the practice. The tested algorithms include:

- Both merge sorts.
- Funnel sort as described here.
- The well known heap sort with binary heap.

- The Quicksort. The used one is slightly modified hybrid version, which uses bubble sort to handle inputs of size at most of 10 elements. This is inspired by the implementation of `qsort` function in the GNU C Library [13]. That one, however, uses Insertsort.
- The `funnelHybrid` is a funnel sort that switches to the Quicksort with inputs smaller than 2^{16}.
- The `funnelFullHybrid` does the same, but it also replaces small funnels (those with just a few input streams) by classical multi-mergers (mergers with possibly more than 2 inputs) with trees on the inputs, to avoid the recursion overhead.

As we see below, the hybrid versions perform slightly better. This is probably because the divide and conquer benefits are remarkable on large inputs, but they are outweighed by the constant overhead of recursion calls on the small subtasks. The overhead is relatively large compared with the work spent on a small task.

The hybrids perform slightly better, as the pure algorithms, while good on large inputs, have constant overhead for each task, which is relatively high on very small tasks.

We show several graphs of memory and cache accesses and run times. Most of the test cases are sorting 4-byte integers.

The first experiment shown in the Figure 6.2 is measuring the number of cache misses during the sorting. This is on a simulated 8MB large cache split into 64B lines.

As we can see, the funnel algorithm perform badly on small inputs, but it wins in large ones. Quicksort is reasonably good on all inputs and heap sort performs well only on small inputs.

One of the reasons for funnel sort to perform badly on small inputs is because it uses more auxiliary memory, therefore other algorithms fit into the cache a little bit longer.

But if we look at the real measured run time of the algorithms in Figure 6.3, we find out that funnel sort and its hybrid modification are rather slow, only the heap sort is slower than them. Quicksort is by far the fastest of the tested algorithms.

As illustrated in the graph in Figure 6.4, a possible reason might be that the funnel sorts have by far largest number of accesses to memory (both internal and external). In the cache-oblivious model, the accesses to internal memory is for free, but it is not so in the reality. Accesses to the cache is faster by orders of magnitude, but the funnel sort also perform orders of magnitude more accesses. This highlights a weak spot in the cache oblivious model.

However, the model is not completely useless. We can see that the heap sort has almost as much less accesses than the funnel sort as Quicksort. But as the accesses are in the wrong order, it creates far more cache misses and makes it slower than the funnel sort.

Therefore, if we had data large enough so the smaller asymptotic complexity in cache misses would win over the large constant factor in the RAM run time (as the number

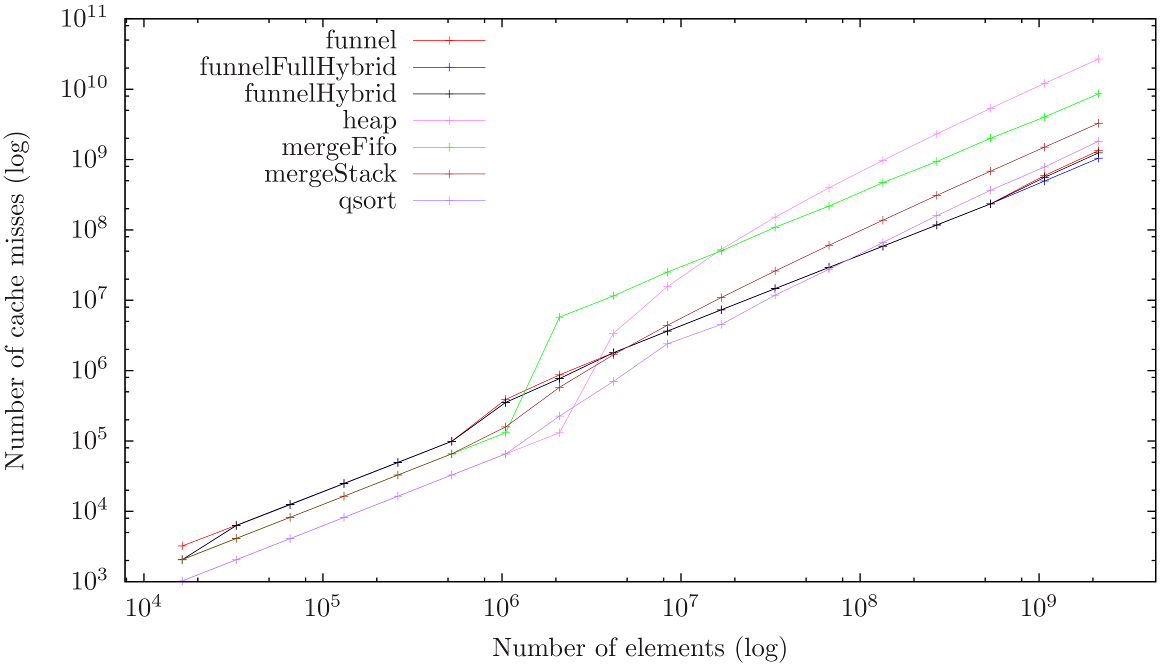

Figure 6.2: Number of cache misses during a sort

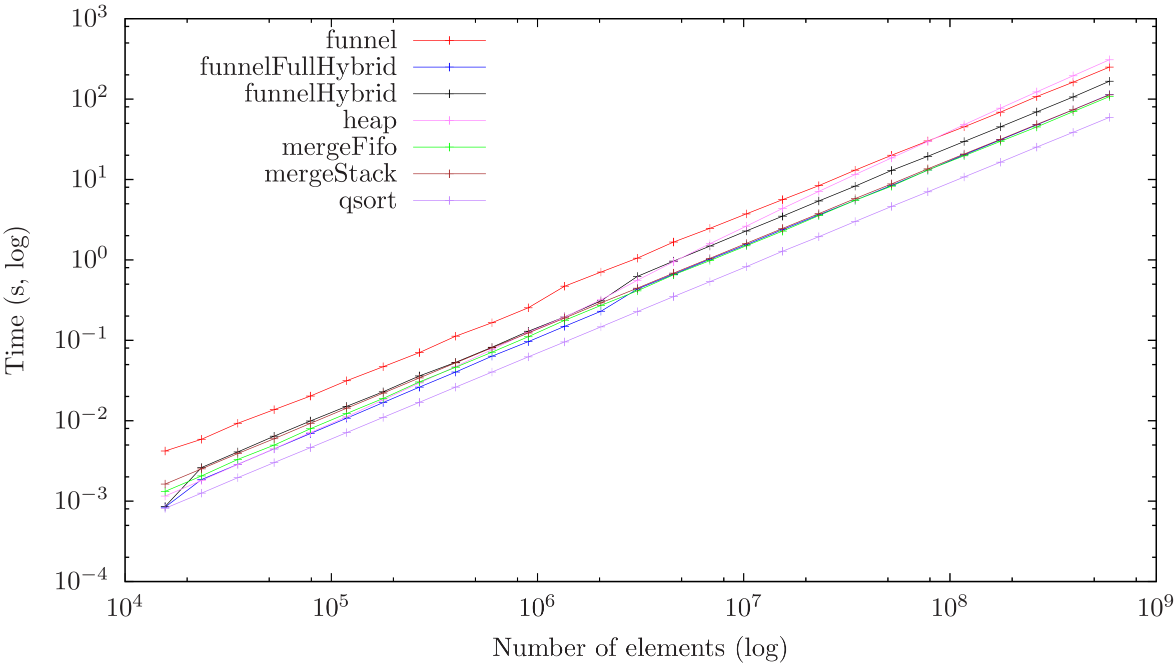

Figure 6.3: Run time of sorting

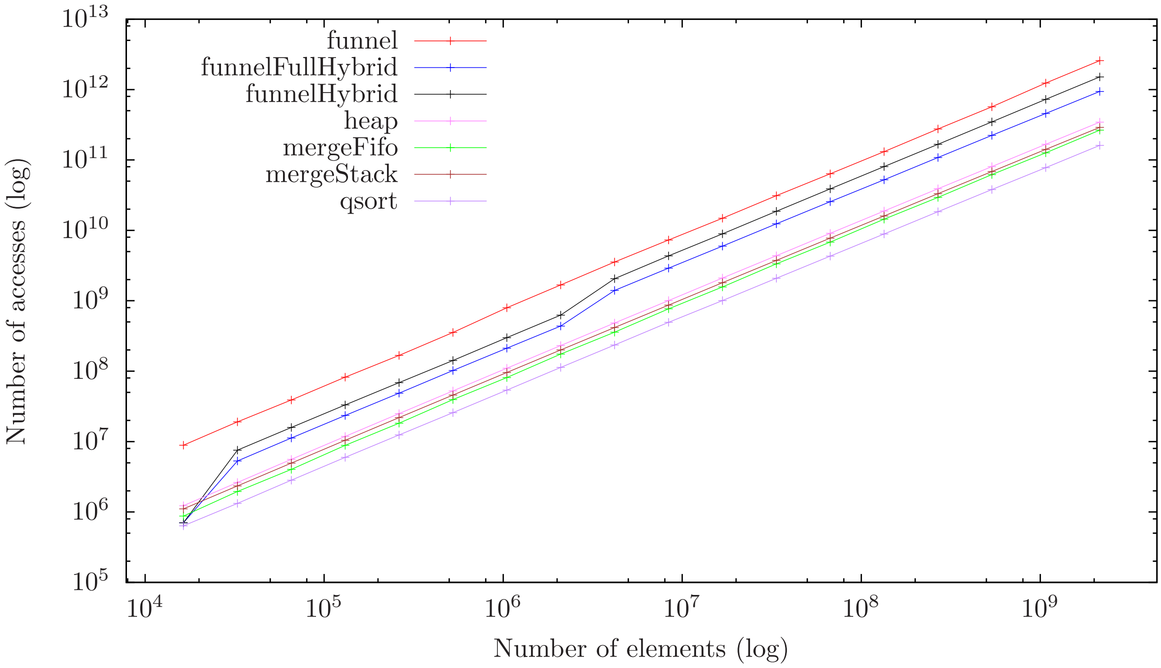

Figure 6.4: Number of total data accesses during a sort

of memory accesses is limited by it), funnel sort might turn faster than Quicksort.

To illustrate that, we run an experiment when the external memory was a disk and the internal memory RAM. Elements 200 bytes long with integer keys were sorted. It is shown in 6.5. We see that the funnel sort performs better, due to the larger difference between internal and external memory speeds. The tendency looks like it would become faster than Quicksort with large enough data.

We also show the run times of several parallel implementations on a 6-core CPU. The funnel sorts can run recursive subtasks in separate threads, but the final merge by the huge funnel on the top is performed single-threaded. It is in the Figure 6.6.

Note: As Quicksort performs better in practice, we'll use it instead of funnel sort as the subroutine in the practical experiments with algorithms in the following chapters.

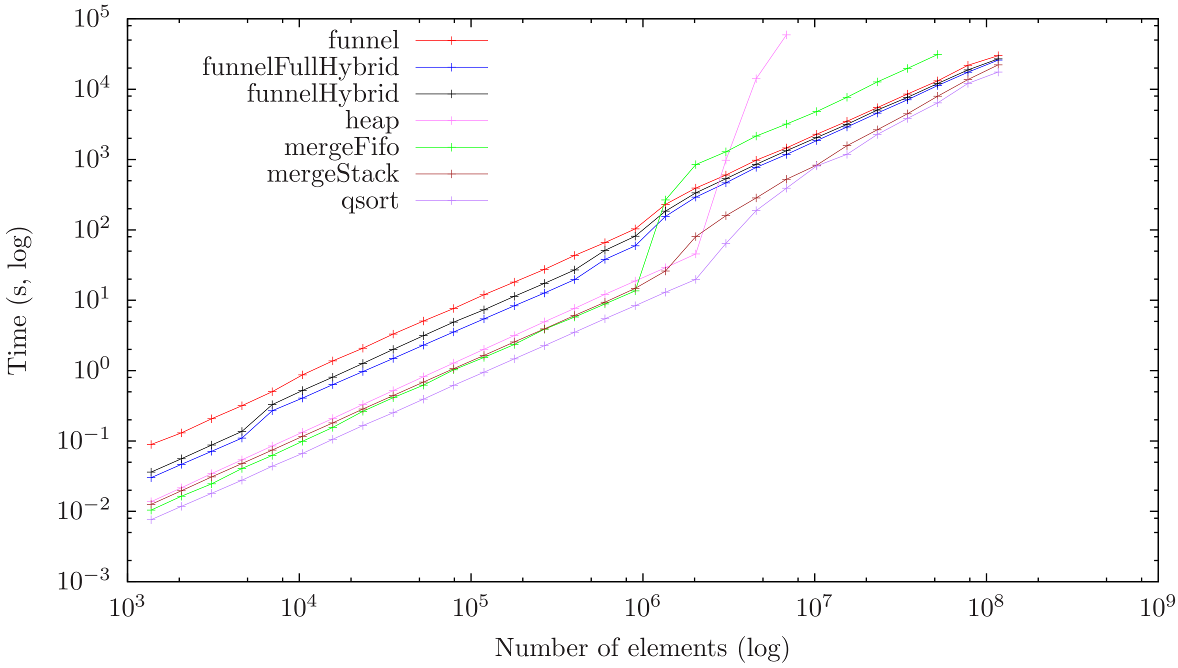

Figure 6.5: Run time of sorting with external memory being disk

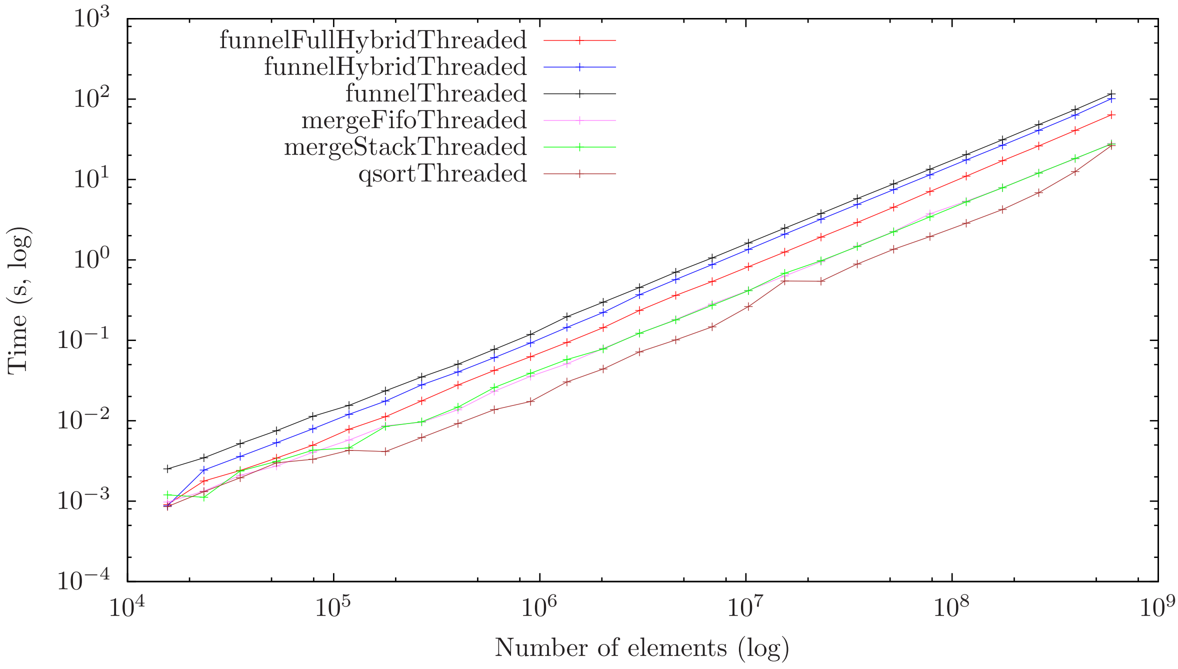

Figure 6.6: Run time of parallel sorting

7. Matrices

Many graph algorithms work by representing the graph as a matrix and reducing graph operations to matrix operations.

The usual representation is an adjacency matrix. It has $|V| \times |V|$ elements and the one on the position (i, j) representing an edge from vertex i to j. This means that adjacency matrix for an undirected graph is symmetric. We can represent an existence of edge with zero and non-zero number, or we can store the length (using ∞ in case of nonexistence of the edge) or store whatever other information.

Therefore we'd like to have a look at operations with matrices. Clearly, if the matrix is in any representation where the elements are stored in memory consecutively , adding, subtracting, assigning and comparing matrices is not interesting, as we can simply scan both matrices in their natural memory representation.

The interesting operation is matrix multiplication. First, let us look at the usual multiplication algorithm as done on paper. Then we show several ways to improve its speed.

To make the analysis and description easier, let us consider square matrices with size n equal to some power of two (therefore the matrices have n^2 cells and they can be repeatedly divided to quarters). In cases where it matters, it is easy to generalize the algorithm and its analysis to arbitrary matrices.

7.1 Straightforward multiplication

The matrix multiplication is defined using scalar products of vectors. Each cell is computed from a row of the left matrix and column of right one. This gives a natural and simple implementation by three nested cycles:

```
for i in 0 .. n - 1:
  for j in 0 .. n - 1:
    result[i, j] = 0
    for k in 0 .. n - 1:
      result[i, j] += a[i, k] * b[k, j]
```

Also, the canonical representation of matrix in computer memory is by two-dimensional array. This leads to so-called row-major[1] order of the elements. Each line is stored consecutively and the lines are concatenated together to form one long sequence.

[1] Or column-major, which differs only in the order of array indices. But it is trivial to see these representations are equivalent in their properties.

The RAM run time is clearly $\mathcal{O}(n^3)$ and we need only $\mathcal{O}(1)$ additional memory except for the inputs and the output.

But the number of transfers is not optimal. When we compute the value of single result element, we need to traverse a row in the left matrix and a column in the right one. Traversing the row is a scan. However, traversing the column is suboptimal. Suppose the matrix is large enough, more precisely $n \in \Omega(L)$ and $n \in \Omega(M/L)^2$. This means that each element of the column resides in a different page. And there are not enough pages in the internal memory, so we get some cache misses on each column. If there are $P = M/L$ pages in the internal memory and the $n \geq c \cdot P, c > 1$, we get $\Theta(n)$ memory transfers on each column traversal.

This leads to $\Theta(n^3)$ memory transfers for the whole multiplication – which is the worst possible for an algorithm running in $\mathcal{O}(n^3)$ time.

7.2 Divide and conquer

The following algorithm was shown by Prokop [1]. We notice that the problem with columns happen only on large matrices. So we try to use the divide and conquer approach to get smaller ones.

We take the matrix and split it into four quarters by cutting it in half both horizontally and vertically. If we cut matrix A, we will call these ***blocks*** as $A_1, \ldots, A_4$ as shown in Figure 7.1.

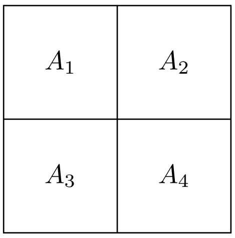

Figure 7.1: Splitting of a matrix to 4 smaller ones

We notice we can build the result by pretending the blocks are in fact elements of matrix 2×2. If we had $R = A \cdot B$, we could get the result as:

$$\begin{aligned} R_1 &= A_1 \cdot B_1 + A_2 \cdot B_3 \\ R_2 &= A_1 \cdot B_2 + A_2 \cdot B_4 \\ R_3 &= A_3 \cdot B_1 + A_4 \cdot B_3 \\ R_4 &= A_3 \cdot B_2 + A_4 \cdot B_4 \end{aligned}$$

[2] In fact, the first could be deduced from the second by the tall-cache assumption.

The multiplications of the blocks are solved recursively and summing them together is straightforward.

When we get to inputs of size 1×1, we solve these trivially.

Instead of computing the RAM complexity and proving correctness, we prove the following theorem and get these two as corollaries.

Theorem 14. *The algorithm sums the same addends as the straightforward algorithm.*

Proof. We prove it by induction on the size of the matrix. If we multiply matrices 1×1, it clearly holds.

Now, we multiply a bigger matrix. If we multiply it the straightforward way, we take a row from the first matrix and a column from the second and perform a scalar products of vectors, which is a sum of some products of elements.

We can cut both vectors in half and look at the halves separately. These are exactly the multiplications that happen in corresponding quarters of the matrix, if we use the divide and conquer algorithm (from the previous step of induction). And, finally, in the local summing of the corresponding quarters, these larger sums are put together. □

Corollary 1. *The algorithm performs matrix multiplication of given operands.*

Corollary 2. *The RAM run time of this algorithm is $\mathcal{O}(n^3)$.*

Theorem 15. *The algorithm needs $\Theta(n^2)$ memory.*

Proof. We need $\mathcal{O}(\log n)$ of memory for the recursion stack.

When splitting the input matrices, it is enough to store the ranges only. As the inputs are never modified, we don't need to copy the elements.

But we need to store the intermediate results. When we are solving a task, we need to have $\Theta(n^2)$ of space to store our result and we need to get all the 8 results of our subtasks, which is $\Theta(n^2)$ as well.

The size of the task is decreasing exponentially with the recursion depth. Therefore the size of the top-level task dominates the bound. □

Theorem 16. *The algorithm needs $\mathcal{O}\left(\frac{n^3}{L}\right)$ memory transfers.*

Proof. The tasks become small enough to fit completely into the internal memory at some point. Anything below this point in the recursion is done for free. As we multiply elements only in tasks of size 1×1, this is below the point, so we don't need to care about the memory transfers caused by the multiplications. We can count the transfers caused by matrix additions only.

As there are at most $\mathcal{O}(n^3)$ total operations, as shown above, there must be at most as many additions. The additions happen in groups – we add up a whole square matrix at once each time.

There are three kinds of sizes of the tasks. Some of them are small enough to be somewhere deep under the level where they fit into the memory. These are for free.

Then there are the large tasks that don't fit. Looking at a large task, we first notice that the size of the matrix $m \in \Omega(L)$. If it wasn't, then the task would need only $\mathcal{O}(L^2)$ internal memory, but this is $\mathcal{O}(M)$ because of the tall cache assumption. Such task would fit inside the internal memory, therefore it wouldn't be large.

When adding the large matrices, we scan each row. As the row is $\Omega(L)$ long, we can simply sum their costs together, so we get to $\Theta(m^2/L)$ of memory transfers while performing $\Theta(m^2)$ additions.

And last, there are the tasks which do fit, but their parent doesn't. These need to be read into the internal memory first. But as the parent has a constant number of subtasks and reading each of them must be no more expensive than processing the whole parent, we can amortize the constant number of such tasks to the parent. Therefore we can count these tasks as if they were for free.

The total number of additions is composed of large tasks and tasks which are for free. Each free task performs some of the $\mathcal{O}(n^3)$ additions, but causes no memory transfers. Therefore assuming all the tasks are large could only make the final estimate larger, which is correct for an upper-bound estimate.

Therefore the total number of transfers is $\mathcal{O}(n^3/L)$. If there were more, the large tasks would perform too many additions. □

Note: In case when the matrix isn't square, we can split asymmetrically. We take the three sizes that are relevant for the multiplication (height of the left matrix, width of the right matrix, width of the left matrix, which is equal to the height of the right one) and split only the biggest one of them in half. This doesn't change the estimates of the algorithm (only the constants) and makes it more flexible.

7.3 The Z representation

We'll introduce one more improvement. This again was introduced by Prokop [1]. We notice that the original input matrix has at most two pages not completely filled with data. This may not be true for the smaller blocks, there may be two incomplete pages on each row. We want to eliminate this.

Therefore, we change the order in which we store the elements. We take the four blocks and take representation of each of them. We concatenate these representations together in a fixed order, let's say in the order of their as in the Figure 7.1. The representations of the blocks are built recursively the same way.

This ensures that each a block of any size is stored in a continuous part of memory. Therefore there are no incomplete pages in the middle, only at the ends.

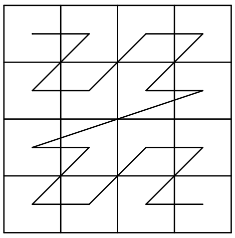

Figure 7.2: The Z representation of a matrix

The fact that each matrix has at most 2 incomplete pages does not improve the bounds. It does however allow us to prove the bounds without the tall cache assumption, as we needed it only to show the large matrix has small number of incomplete pages.

But, as shown in Section 7.5 below, it makes the algorithm run faster in practice. There are two possible explanations. The total number of cache misses might be lower. And the order in which the pages are accessed might be better for cache prefetch heuristics.

7.4 Strassen's algorithm

The following improvement was proposed by Strassen [14]. While the cache-oblivious model was not considered when designing this algorithm[3], it still makes it faster. This is because the algorithm lowers the RAM run time and the run time limits the number of memory transfers.

The divide and conquer algorithm performed 8 matrix multiplications in each tasks. This algorithm is able to compute the result using only 7 multiplications, at the cost of more additions. However, additions are asymptotically cheaper. The algorithm computes 7 auxiliary matrices as follows:

$$\begin{aligned}
M_1 &= (A_1 + A_4) \cdot (B_1 + B_4) \\
M_2 &= (A_3 + A_4) \cdot B_1 \\
M_3 &= A_1 \cdot (B_2 - B_4) \\
M_4 &= A_4 \cdot (B_3 - B_1) \\
M_5 &= (A_1 + A_2) \cdot B_4 \\
M_6 &= (A_3 - A_1) \cdot (B_1 + B_2) \\
M_7 &= (A_2 - A_4) \cdot (B_3 + B_4)
\end{aligned}$$

[3] Simply because the algorithm is much older than the model.

And the final result C is then computed as follows:

$$\begin{aligned}
C_1 &= M_1 + M_4 - M_5 + M_7 \\
C_2 &= M_3 + M_5 \\
C_3 &= M_2 + M_4 \\
C_4 &= M_1 - M_2 + M_3 + M_6
\end{aligned}$$

It can be trivially verified this produces the correct result.

The space complexity is the same as of the divide and conquer algorithm.

Theorem 17. *The run time complexity of the Strassen's algorithm is $\mathcal{O}(n^{\log_2 7})$.*

We'll omit the proof here and refer the reader to the original article [14].

Theorem 18. *The algorithm needs $\mathcal{O}\left(\frac{n^{\log_2 7}}{L}\right)$ memory transfers.*

Proof. We could take the proof of number of memory transfers of the divide and conquer algorithm (Theorem 16) nearly verbatim to prove this one. The only difference is the total number of additions, which is not $\mathcal{O}(n^3)$, but $\mathcal{O}\left(n^{\log_2 7}\right)$. □

Note: Algorithms with even better asymptotic complexity (at least of the run time) exist. Example of these could be the Coppersmith-Winograd algorithm [15] or a more recent one of Williams [16].

7.5 Experimental results

As we can see below, the processor cache influences the speed of matrix multiplication substantially. The reason might be that the number of operations grows relatively quickly with the size of the data. The number of cache misses is in good correlation with the run time of the algorithm.

However, plain recursive algorithms perform rather badly. This is probably because the stack manipulation and extra complexity in the low levels of the recursion outbalance the benefits, as there's little data to work on. The hybrid versions of algorithms perform significantly better, which indicates this might be the case.

The tested algorithms are:

- **classic** – the straightforward algorithm from the Section 7.1.
- **inverse** – the same algorithm, but the matrix representation are transposed, they are stored in column-major order instead of row-major.

- **divide** – the divide and conquer algorithm with asymmetrical splitting from the note at the end of the Section 7.2.

- **square** – the divide and conquer algorithm from the Section 7.2.

- **strassen** – the Strassen's algorithm from the Section 7.4, with row-major matrix representation.

The hybrid versions of algorithms switch to the classical algorithm for matrices of sizes at most 20 × 20 in order to reduce the work with stack and recursion when the benefits of the divide and conquer approach are small. This has a surprisingly high effect on the result.

The Z versions use the Z representation from the Section 7.3. The impact on the speed is good, but only by a small factor. The effect is less visible with the Strassen's algorithm, as that one doesn't traverse the quarters of the matrix in the order of them being stored in the memory.

The fastest of the tested algorithms is the hybrid Strassen's algorithm with the Z representation. The classical algorithm is reasonably fast for small input, but gets a lot worse when they no longer fit into the cache. The non-hybrid versions of recursive algorithms are generally slow.

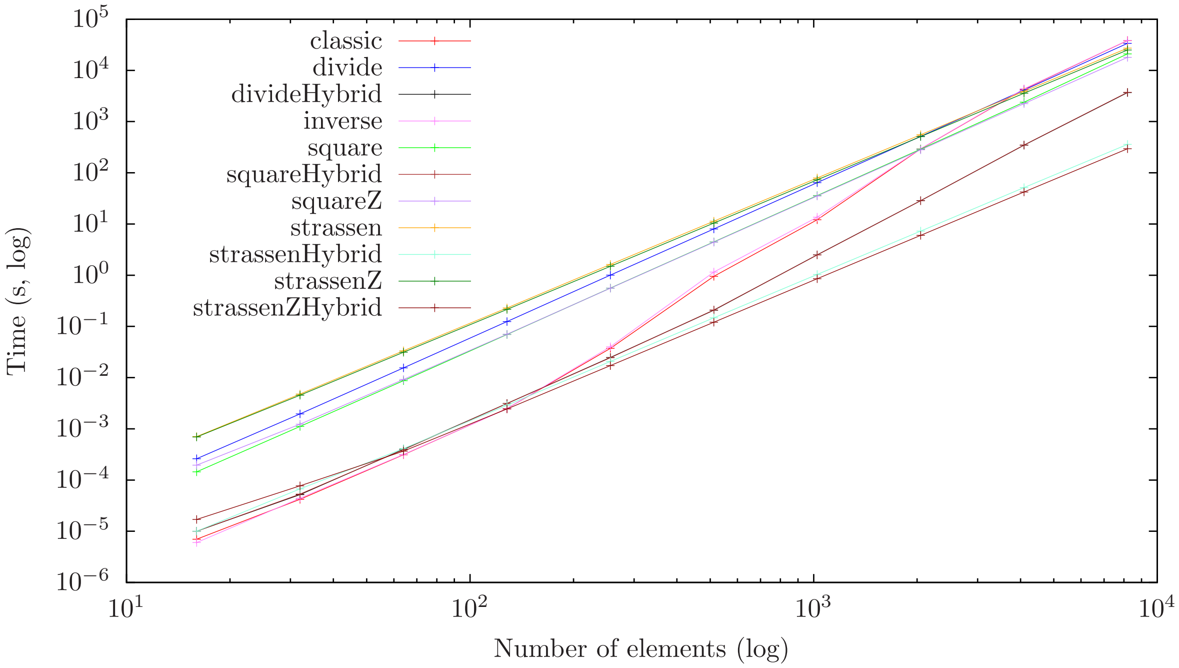

Figure 7.3: Runtime of matrix multiplication algorithms

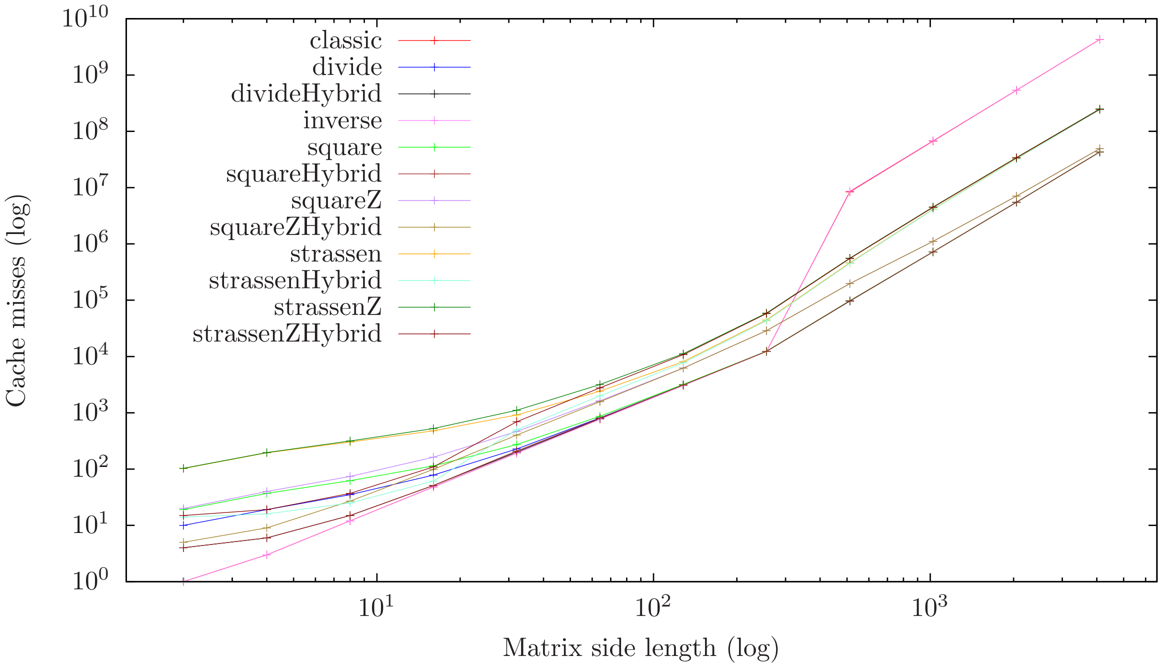

Figure 7.4: Number of cache misses on 1MB cache with 64B long lines

8. Toolbox of graph algorithms and data structures

Here we will show several building blocks which can be used to design many cache-oblivious algorithms for graph problems.

All of them expect the graph to be in the two arrays representation. There's an array of edges (or edge halves in case of an undirected graph – each edge is represented as two directed edges), sorted by their first endpoint. The second array contains vertices. Each vertex has a pointer to its first edge in the array of edges. Therefore we can easily scan all edges going from a single vertex (the vertex has the first edge and the following vertex knows the edge one after the last).

To illustrate why we need special algorithms for the cache-oblivious model, we have a quick look at the classic DFS algorithm. We repeatedly enter a vertex, mark it as visited and scan all its neighbors. We recurse to the first yet unmarked vertex we find. When we return from the vertex, we find another yet unmarked neighbor and continue, until we run out of them.

But we access a vertex on the other side of every edge. This would effectively make a memory transfer for each edge, giving us an estimate of $\mathcal{O}(E)$ memory transfers total.

The theoretical lower bound for DFS is $\Omega(E/L+\text{sort}(V))$, as we need to read the whole input of size $\mathcal{O}(E + V)$. Also, if we were able to visit the vertices in the DFS order in smaller number than $\Omega(\text{sort}(V))$ memory transfers, we would be able to write their numbers to output in smaller number of memory transfers. But the DFS is a permutation of the vertex numbers and $\text{sort}(V)$ is therefore a lower bound, as mentioned in the Funnelsort chapter. Similar lower bound is applicable to many graph problems.

Unfortunately, we know no algorithm that good, but we still improve the upper bound.

Therefore we first show two data structures which will help us eliminate the access on the end of each edge.

8.1 Buffer repository tree

This is a data structure introduced in the article *An Optimal Cache-Oblivious Priority Queue and its Application to Graph Algorithms* [17].

It works with a static set of keys $1, \ldots, k$. This structure will hold tuples $(key, value)$ which we'll call ***messages***. The key is a number from the range $1, \ldots, k$, while value could be arbitrary. However, in our case it'll be in the same range as well. The data structure has two operations. One is adding a new message. The other one is removing and returning all the messages with given key.

Usually, we'll use it in a way where the keys are vertices of a graph. This will allow us to leave a message for a vertex without actually accessing the vertex itself. The vertex will be able to retrieve all the messages addressed to it later in a batch.

The data structure is a binary tree, where the leaves are the keys we use, while each internal node represents an interval of all the keys in its subtree (the root represents the whole set of keys, its left son the lower half of them, the right son the upper half of them).

Each node keeps a list of some messages that belong to the interval. Each message is in exactly one such list. Therefore, messages belonging to a given key reside on the path from root to the leaf corresponding to the key.

To add a new message to the structure, we just put it into the list at the root node.

When we want to extract all keys for a given key, we walk from the root to the leaf. However, we don't only scan the messages in each node, but we take the list of messages and sort them to the two sons of the node by the corresponding key. So, when we finally get to the leaf, all messages we want are located in the leaf and the nodes on the path from the root are empty – we pushed the ones we wanted ahead of us and the others aside.

Before we start with the analysis of complexity, we need to fill in a few technical details, namely how the lists are represented. The root list is an array which is reallocated to twice its size when it gets full. The other lists are different. They are linked lists of arrays. When we're sorting one node to its sons, we count how many messages go to each of the sons, create an array of corresponding size for each of them and write the results to the arrays. The arrays are then attached to the end of the linked lists.

Theorem 19. *The tree uses $\mathcal{O}(n+k)$ memory when storing n messages.*

Proof. We have approximately $2k$ nodes in the tree $(k + \frac{k}{2} + \frac{k}{4} + \ldots)$.

The messages are either in the root buffer or in other buffers. The root buffer is at most twice as large as the number of messages in it. The other buffers have space for the exact number of messages and have some pointers to the next array in the linked list. If we don't store empty arrays, we have at most as many pointers as messages. So, the total size of memory used for the messages is $\mathcal{O}(n)$. □

Theorem 20. *It'll take $\mathcal{O}(n \cdot \log k)$ RAM run time to insert and extract n messages.*

Proof. Each message is first put into the root buffer, then it traverses $\log_2 k$ nodes down to a leaf and then it is taken out. Inserting into the root buffer takes $\mathcal{O}(1)$ amortized time (it might need to reallocate the buffer, but it does so infrequently enough). Moving to a lower level is a constant-time operation as well and picking it at the bottom. So there's $\mathcal{O}(\log k)$ of work per one message. □

Theorem 21. *It takes $\mathcal{O}\left(\frac{\log k}{L}\right)$ amortized memory transfers to insert a message and $\mathcal{O}(\log k)$ amortized memory transfers to extract messages of a single key.*

Proof. It takes $\mathcal{O}\left(\frac{1}{L}\right)$ amortized memory transfers to insert an message into the root buffer, as we can keep the last page of the buffer inside the internal memory at all times and the reallocation can be done by a scan. However, we prepay $\Theta\left(\frac{\log k}{L}\right)$ memory transfers for the future moves of the message.

To scan one array in the list, we pay $\Theta(m/L)$ memory transfers if there are m elements. But there can be as much as 2 pages not completely filled with data, and we need to pay $\mathcal{O}(1)$ transfers for these. Therefore we need $\mathcal{O}\left(o + \frac{p}{L}\right)$ to sort p messages from one node into the sons, if they are stored in o arrays in the linked list.

Each message has $\mathcal{O}\left(\frac{\log k}{L}\right)$ transfers prepaid from the time it was inserted. It uses $\mathcal{O}\left(\frac{1}{L}\right)$ of it on each level to get one level lower, which pays for the whole $\frac{p}{L}$ part of the sum, as we are moving p messages.

Each array in the list will have $\mathcal{O}(1)$ transfers prepaid for its deconstruction. As we create only 2 arrays on each level when going down, we can prepay these from the $\mathcal{O}(\log k)$ transfers we use for extracting elements of one key. □

8.2 Buffer priority tree

Another data structure we'll need represents a set of keys. It gets created with k distinct keys (they can be arbitrary, not necessarily $1, \ldots, k$). It supports single compound operation. The operation takes a list of keys and deletes them from the data structure. It then deletes and returns the key with lowest key still present in the structure, or informs there's no key left to return. We will assume no key is deleted more than once.

The purpose will be to keep list of neighbors which are still alive. From time to time, we'll remove the dead ones and pick another one to examine. Therefore, we'll create a buffer priority tree for each vertex and initialize it with the list of neighbors of the vertex.

This data structure is introduced in the same article as the Buffer repository tree. It also has a very similar structure.

The leafs represent the keys. They are sorted, so we can easily decide to go to the left or to the right when walking from the root. An internal node, if it is still alive, holds a buffer of all the keys in its subtree to be deleted and the number of keys still alive in its interval.

When we perform the operation, we put the set of keys to delete to root and decrease the number of living keys in the whole tree. Then, if it is still positive, sort the keys to be deleted to the left and right son, as with the buffer repository tree, and update their counts. If the count in the left son is positive, we move there, as the lowest living key is there. If it drops to zero, the whole left subtree becomes dead and we ignore it from the time on. We move to the right son and continue.

After some time, we get to a leaf which has a count of 1. We remember it as a result, set the count to zero and decrease the counts on the path to the root, to account for removing the result.

Theorem 22. *The buffer priority tree takes $\mathcal{O}(k)$ memory.*

Proof. As the structure is the same as with the buffer repository tree, the same bounds as in the Theorem 19 apply. We just notice that the elements we put into the tree are the keys to be deleted and a key can be present at most once, so we have $\mathcal{O}(k)$ for the base structure and $\mathcal{O}(k)$ for the elements present. □

Theorem 23. *It takes $\mathcal{O}(k \cdot \log k)$ time and $\mathcal{O}(\mathrm{sort}(k))$ memory transfers to create the structure.*

Proof. The tree structure can be laid out in continuous array of memory. When we build it from the bottom, we simultaneously scan a lower level and write an upper level. When the level is finished, we move one level up, but the scan continues on the following place in memory. So the construction can be seen as two scans from the end of the array.

The limiting operation is therefore creating the lowest level. We need to sort the keys for that and it is where both bounds come from.

Note: If it is guaranteed the input is already sorted, we can build it in $\mathcal{O}(k)$ time and $\frac{k}{L}$ memory transfers. □

Theorem 24. *It takes $\mathcal{O}(k \cdot \log k)$ time to delete all the keys, either as inputs or results.*

Proof. Each element to be deleted travels $\mathcal{O}(\log k)$ levels down from root to be deleted. An element is returned after the algorithm walks the $\mathcal{O}(\log k)$ levels from the root. So, we have an upper bound of $\mathcal{O}(\log k)$ operations per key and each key can be removed at most once. □

Theorem 25. *We need $\mathcal{O}\left(\left(\frac{m}{L}+1\right) \cdot \log k\right)$ amortized memory transfers to perform the operation with m elements to delete.*

Proof. We perform the same set of operations as if we first inserted m elements into a buffer repository tree and then looked up elements of one key. Therefore using the proof of the Theorem 21 and summing the operations together gives this result. □

8.3 Searches

We will show the BFS and DFS. They are modifications of their usual RAM versions. All algorithms in this chapter come from [17].

As we showed at the beginning of this chapter, the classic DFS does not utilize the cache very well. This is because it accesses a vertex at the end of each examined edge. We'll get rid of the problem here.

In the following, we'll assume there's at last so many edges as vertices, otherwise we could use the RAM algorithm and get a better result.

Instead of having a flag in each vertex to signal if it was visited, each vertex will remember if each of its neighbors is visited. This duplicates the information and looks counter-intuitive, but it'll actually make the algorithm better in the sense of memory transfers, because we'll batch the updates to the information.

We use the above two data structures. Each vertex gets its own buffer priority tree, constructed from the numbers of its neighbors. This will hold the yet unvisited neighbors of the vertex (but won't be up to date at all times).

Furthermore, we'll have a single global buffer repository tree, serving the purpose of a message board between the vertices. Each message there means a vertex A tells vertex B that it (vertex A) was visited and B should update its copy of the information. It is stored under the key B, so B can read all messages directed to it at once.

Whenever we put a vertex number onto a stack, we mark it as visited. But instead of noting it inside the vertex, we do it by notifying all its neighbors it was visited, by inserting a message directed to each of them, containing the current vertex number.

Single step of the algorithm looks at the vertex whose number is on top of the stack (but keeps it there). First, we take the messages directed to this vertex out of the buffer repository tree. These list the neighbors which already got onto the stack, being visited. We remove these from the buffer priority tree belonging to the current vertex and take the first yet unvisited neighbor out of it. We place this vertex onto the stack, marking it as visited in the way outlined above. If there's no more unvisited neighbor (the buffer priority tree didn't give us anything), we remove the current vertex from the stack.

If we want to visit the vertices in a BFS order, we use a queue instead of a stack. The rest of the algorithm is exactly the same.

You may have noticed the above works for undirected graph, but we talk about neighbors in both directions interchangeably. This was to make the idea simpler to grasp at first. It works for directed graphs as well, but some care must be taken to distinguish the directions. In fact, to enumerate the neighbors we should notify, we need a copy of the graph that has all its edge reversed, because we actually need to notify the vertices whose neighbor we are, not our neighbors (we need the vertices from which an edge leads to us).

Theorem 26. *The algorithm visits (marks) the vertices in DFS, resp. BFS order.*

Proof. The algorithm is different from the usual RAM version in two details. One is, we use explicit stack, while the RAM version usually uses recursion (but it is implemented with explicit stack sometimes as well). It is easy to see that this change makes no difference to the output.

The other detail is marking strategy. In the RAM version, a flag is held inside each vertex. In the cache oblivious version, each vertex has a list of not yet visited neighbors, effectively duplicating the information multiple times. However, we show that the marking strategies give the same answers.

In the beginning, a vertex is in the lists of all its neighbors, so from the point of the neighbors, the information is correct. No other vertices will be interested if it is visited.

When we mark a vertex as visited, we send a message to each of the neighbors. If any of the neighbors want to know if the vertex is visited, it first reads all the messages and removes it from the list, therefore it'll give the correct result. □

Theorem 27. *The algorithm runs in $\mathcal{O}(E \cdot \log E)$ runtime.*

Proof. We need to build three things during the initialization. The BRT is built in linear time ($\mathcal{O}(V)$). Each buffer priority tree with k elements takes $\mathcal{O}(k \cdot \log k)$ time to build and $\sum_{v \in V} k_v = |E|$, the function is sub-additive, therefore this is $\mathcal{O}(E \cdot \log E)$ total. To build the reversed graph, we generate a new array of (reversed) edges, sort them by their start vertex and scan through them to build the pointers from vertices to them. This can be done in $\mathcal{O}(E \cdot \log E)$.

Each edge generates at most one message for the BRT and each neighbor is extracted at most once from each BPT it is in. So we have at most $\mathcal{O}(E)$ messages and $\mathcal{O}(E)$ extractions from BPTs in the lifetime of algorithm. One message respective extraction takes $\mathcal{O}(\log V)$ respective $\mathcal{O}(\log E)$ (which is the same, because $V \leq E \leq V^2$). So this takes $\mathcal{O}(E \cdot \log E)$ total. □

Theorem 28. *The algorithm needs $\mathcal{O}(E)$ auxiliary memory.*

Proof. This is trivial, as the BRT takes $\mathcal{O}(V + E)$ memory (for the structure and possible stored messages), each BPT takes $\mathcal{O}(k)$, which is $\mathcal{O}(E)$ total and the reversed graph can reside in $\mathcal{O}(E)$ memory. □

Theorem 29. *The algorithm needs $\mathcal{O}\left((V + \frac{E}{L}) \cdot \log V + \operatorname{sort}(E)\right)$ memory transfers.*

Proof. The initialization phase can be done with $\mathcal{O}(V + \operatorname{sort}(E))$ memory transfers. We could prove it the same way as in the proof of Theorem 27 above.

Each vertex is marked as visited at most once. To mark a vertex, we scan the list of neighbors, which takes $\mathcal{O}(1 + \frac{n_v}{L})$ if vertex v has n_v neighbors. Then we generate n_v messages and insert them into the BRT, which takes $\mathcal{O}\left(\frac{n_v \cdot \log V}{L}\right)$ memory transfers. As there are E neighbors total over all the vertices, this takes $\mathcal{O}\left(\left(V + \frac{E}{L}\right) \cdot \log V\right)$ transfers.

Now we notice that a vertex at the top of stack is accessed $\mathcal{O}(V)$ times. We divide these accesses into two types. One type is when the vertex has no unmarked neighbors. Then it is removed from the stack and never gets there again. So there are at most V accesses of this type. The second type is when we find an unmarked neighbor. We mark this neighbor at once, and each vertex is marked at most once. This means there are at most V accesses of the second type.

During each access, we extract m messages from the BRT, then give them to our BPT to remove the corresponding vertices and extract a vertex from it. We pay $\mathcal{O}(\log V)$ transfers to extract them $\mathcal{O}\left(\left(\frac{m}{L}+1\right)\cdot\log V\right)$ to delete them from the BPT. As we said before, there are at most E messages during the lifetime of the algorithm, and we do $\mathcal{O}(V)$ accesses to a vertex on top of the stack, this takes $\mathcal{O}\left(\left(\frac{E}{L}+V\right)\cdot\log V\right)$ memory transfers over the whole algorithm.

There are at most $\mathcal{O}(V)$ operations with the stack or queue, which is small enough we don't need to pay attention to implement them in any special way.

Now we just sum the estimates together to get the final result. □

8.3.1 BFS for undirected graphs

We can get a better result for BFS on undirected graphs. This also comes from the same article [17].

The first observation we make is that the BFS processes the graph in layers. The first vertex is the source one. Then there's a group of vertices in the distance 1 from it, then a group of vertices in distance 2, etc. We call the layer in distance d from the source R_d[1]. Furthermore, the order of vertices in the same layer isn't important for most applications.

Another observation is, a vertex living in a layer R_d has neighbors only in the layers R_{d-1}, R_d and R_{d+1}.

The algorithm will produce a new layer in each step of computation. Let's assume we have a layer R_d. We take all the unique neighbors of vertices in the layer, which produces a list containing all the vertices from layers R_{d-1} and R_{d+1} and some vertices from layer R_d. So we remove the vertices of layers R_{d-1} and R_d (we already computed these before) and get all the vertices of layer R_{d+1}. We bootstrap the computation by having a layer R_{-1} empty and the layer R_0 containing the source vertex.

The implementation is rather simple. To get all the neighbors of a layer, we go through all the vertices and concatenate their lists of neighbors together. Then we sort them and remove the duplicates by a scan of the array. We can remove the vertices of previous

[1] We would use L_d, but it could get confused in the estimates with L which means the size of singe memory page.

layers by scanning through all the 3 arrays in parallel and removing the correct vertices (all 3 arrays are sorted by now).

Theorem 30. *This algorithm produces BFS ordering.*

Proof. The first and crucial observation is that if there's an edge $f = \{u, v\}$ between vertices in layers R_d and R_e (u being in R_d), then $|d - e| \leq 1$. If (without loss of generality) $d < e - 1$, then v has distance at most $d + 1$ – since u is in R_d, it has distance d and by going through the edge f, we get $d + 1$. This would contradict $d < e - 1$.

The layers R_{-1} and R_0 trivially contain the correct vertices. We'll continue through the other layers by induction.

So, layers R_{d-1} and R_d are correct. We need to check the computed layer R_{d+1} is complete and doesn't include any extra vertices.

First assume it contains a vertex v_e that belongs to a different layer. It can't belong to layers R_d nor R_{d-1}, because we removed these. So, it can belong to a closer layer (layer $R_c, c < d - 1$). But in this case, there's a neighbor in R_d and $d - c > 1$, which contradicts the observation.

So, now assume it belongs to a layer R_e, where $e > d + 1$. But we found it as a neighbor in distance d, so there's an edge between R_d and R_e and $e - d > 1$. This again contradicts the observation.

Now, to show the layer is complete, let's say we missed a vertex v_m. But because it is in distance $d+1$ from the source vertex, it must have a neighbor in distance d. Because R_d is correct, it is contained in it. But then, we must have found v_m as the neighbor of such vertex and we couldn't remove it, as it wasn't in the layers R_{d-1} nor R_d. □

Theorem 31. *The algorithm runs in $\mathcal{O}(E \cdot \log E)$ time and it needs $\mathcal{O}(\mathrm{sort}(E) + V)$ memory transfers.*

Proof. Each endpoint of an edge enters an array at most once. The array is then sorted and scanned 4 times (once to remove duplicates, once when removing elements from it and twice when used as a source of elements to remove from another array).

The V corresponds to both partial pages when copying the neighbors of each vertex to the array and for initializations of sorts (there's one sort in each layer and there can be no more layers than vertices).

As the sorting estimates for both runtime and number of memory transfers are subadditive, we can estimate the total sorting times by $\mathcal{O}(E \cdot \log E)$ and $\mathcal{O}(\mathrm{sort}(E))$ respectively. The scanning bounds are smaller than these. □

Theorem 32. *The algorithm needs $\mathcal{O}(E)$ auxiliary memory.*

Proof. A vertex is in at most one layer, so $\mathcal{O}(V)$ memory would be enough to store the layers. But there are duplicates at the array which will become the next layer at the beginning of the computation step. However, as it is a concatenation of the neighbors of certain vertices, it is bounded by the number of edges. □

8.3.2 Experimental results

We have a look at how the algorithms perform. The tests were on both dense and sparse graphs (see Chapter 4). The `Classic` algorithms are the RAM ones (with marks directly in the vertices). The `BRT` ones are the BRT/BPT based ones (from Sections 8.1 and 8.2). The `bfsSort` is the sort-based BFS from Section 8.3.1.

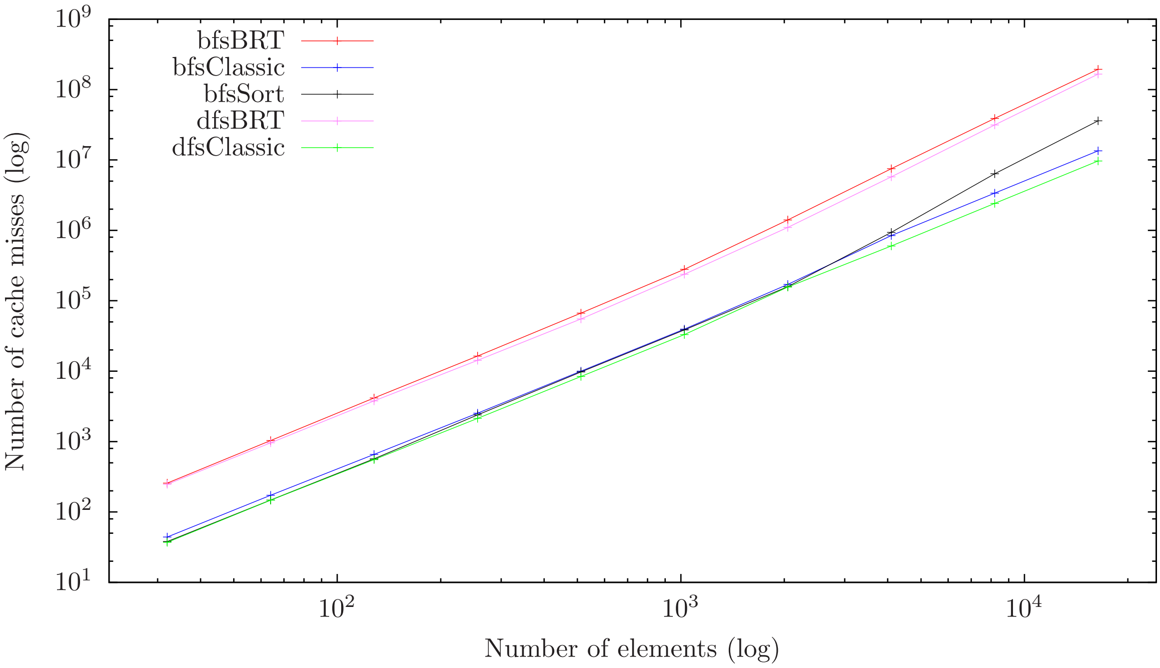

Figure 8.1: Cache misses for searches on a dense graph

As we see in Figures 8.1 and 8.2, the BRT based algorithms have too much overhead for practical use. The sorting algorithm have worse time complexity, especially when there are many edges, but in case of sparse graph the number of misses is approximately the same as with the classic algorithms.

When measuring the run time, we also include the threaded version of the sort based BFS. The only change here is that it uses a threaded sort to speed the computation up. The others are not threaded, since there's no intuitive way to make the algorithms parallel.

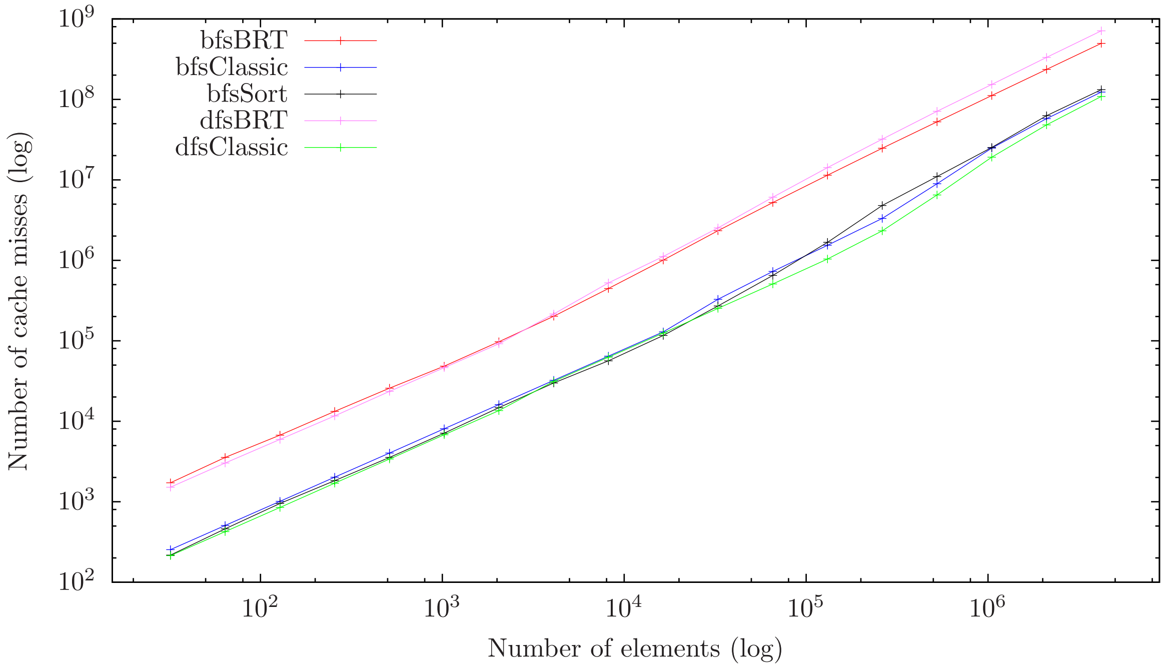

Figure 8.2: Cache misses for searches on a sparse graph

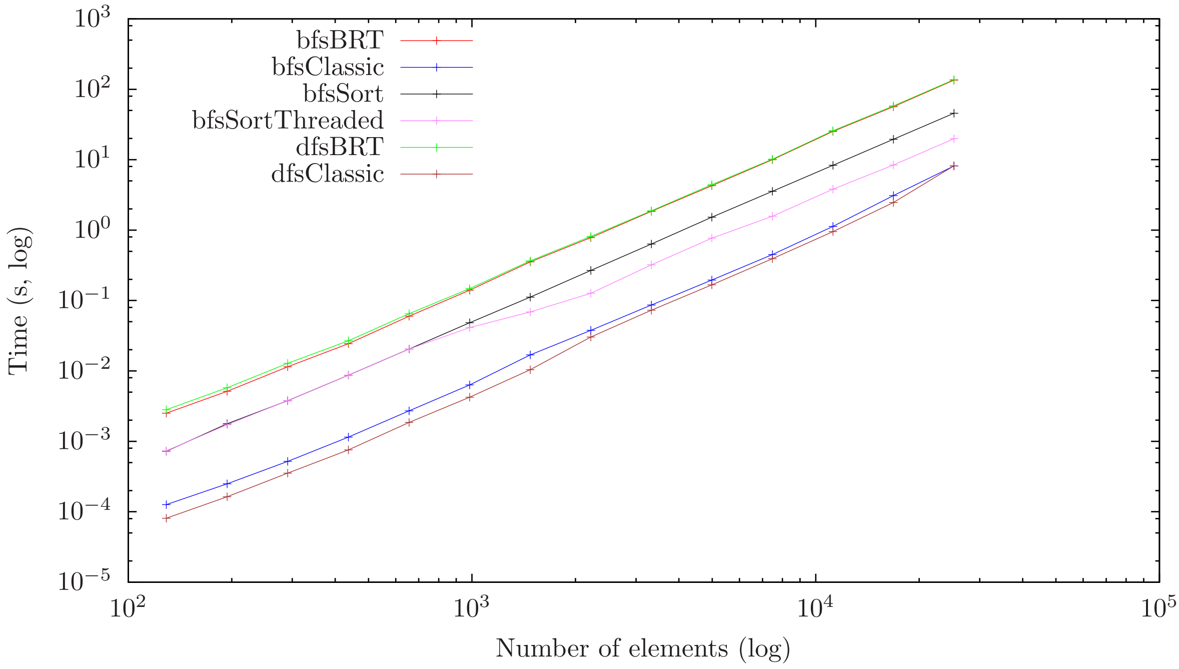

Figure 8.3: Time of search on a dense graph (10 iterations)

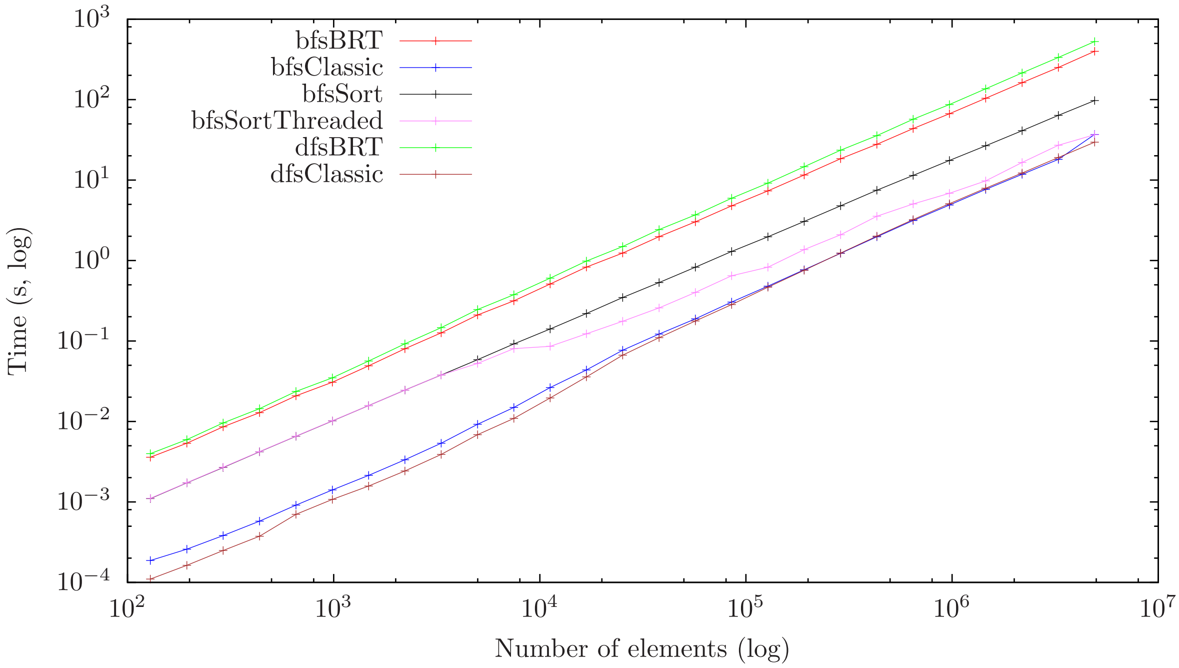

Figure 8.4: Time of search on a sparse graph (10 iterations)

While it is no surprise the BRT based searches are slow (as they have a lot of cache misses), it is interesting to note that in the case of sparse graph the sort-based BFS actually performs reasonably well and in the threaded version it is comparably fast to the classic BFS and DFS on large enough data.

9. Components

One of the well-known graph problems is finding connected components. We'll start with undirected graphs for simplicity.

9.1 BFS

First, we show an algorithm which is a straightforward modification of the usual RAM algorithm. We're scanning the vertices in any order. Each time a yet unvisited vertex is discovered, a search is started from that vertex and all vertices found from this startpoint are marked to be in the same component.

Theorem 33. *The algorithm performs $\mathcal{O}(\text{sort}(E) + V)$ memory transfers.*

Proof. As we do not care about the order of vertices in the search and the graph is undirected, we can use undirected BFS from Section 8.3.1. For each component C_i, the search cost for that component is $\mathcal{O}\left(\text{sort}\left(E_{C_i}\right) + V_{C_i}\right)$ memory transfers. Because $\forall i; V_{C_i} \geq 1 \forall i$, we don't need to consider the $\mathcal{O}(1)$ memory transfers for initialization of the BFS.

The function is sub-additive, each edge or vertex is contained in exactly one component, so we can just sum their costs together to $\mathcal{O}(\text{sort}(E) + V)$.

When looking for a yet unvisited vertex, we can do it in any order, because we already paid the $\mathcal{O}(V)$ in the estimate. However, scanning them is probably the simplest way. □

Similarly, we can show that the algorithm runs in $\mathcal{O}(V + E \cdot \log E)$ RAM time. It is easy to see it takes $\mathcal{O}(V + E)$ memory.

9.2 Divide and conquer

In this chapter, we introduce a completely new algorithm. This one works by the divide and conquer principle described in Section 5.1.3.

9.2.1 Overview

In each step, we first divide the vertices into two groups (left and right) and edges in three groups (left, right and middle). The graph with only left, right or middle edges will be called left, right or middle graph respectively.

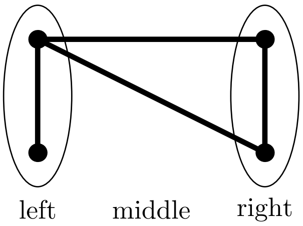

Figure 9.1: Vertex and edge groups

Then we find the connected components of the left and right graph by calling the same algorithm recursively.

Then we take the middle graph and contract components returned from the left and right graph. We call the algorithm recursively on this graph. This gives us the final components, but with some vertices contracted together.

So we finally expand the vertices to components again to get components in the original graph.

```
components(graph, depth):
  left, middle, right = split(graph, depth)
  lComponents = components(left, depth + 1)
  rComponents = components(right, depth + 1)
  middle.contract(lComponents)
  middle.contract(rComponents)
  result = components(middle, depth + 1)
  result.expand(lComponents)
  result.expand(rComponents)
  return result
```

The recursion stops when we can be sure that the input graph is simple enough. It happens when each vertex has degree at most one – in that case each edge defines a separate component. This trivially happens when there's at most one edge in the input graph. But we detect the situation in more complicated cases as well, as described below.

9.2.2 Technical details

We need to provide further implementation details to be able to prove correctness and estimate the complexity of the algorithm.

We first note that we never handle vertices explicitly. The input will consist of edges only and we expect the vertices to be numbered from 0 to $n - 1$. The edges can be in any order, but each one is there only once (unlike many representations, we don't store two copies of each edge, one in each direction, but only one instance). We pass the recursion depth and the number of vertices in the original graph with the input.

The result of each recursive call will be function (in the mathematical sense) assigning a component number to each vertex number. We represent the function as a list of pairs, each containing the vertex and component number. We keep the result sorted by the vertex number.

We'll keep an invariant that a component with number ℓ contains the vertex with number ℓ. This gives us two advantages. For one, we can omit the pairs where the vertex and component numbers are equal, which still preserves all the needed information. This means we can ignore singleton vertices from now on, as they produce no output. Also, it means the output of a problem is never larger than the input. If there's a pair in the output with vertex ℓ, it means it is mapped to different component, therefore it is connected to something by an edge.

The other advantage is, the component numbers returned from the left and the right graph can not collide. If a vertex has at least one edge in one problem, it is singleton in the other and no component with that number is produced (except the implicit singleton one, obviously).

To solve the trivial subproblem (one where each vertex has degree at most one), we notice that each edge represents a separate component. So, in each component, we need to produce one pair, mapping one of the vertices to the other (the other vertex is implicitly mapped to the same number). But we can notice that the input edge is the exact pair we need to produce. So we just sort the input and return it as an output.

If we are not sure the subproblem is trivial yet (we have more than one edge), we split the vertices, as described in Section 5.1.3. When we are unable to split the vertices, because we already used all the bits in their IDs, we need to do a thorough examination if the problem is trivial. This is described below.

To simplify the future contraction step, we "flip" the edges in the third set so that their first endpoint is in the left vertex group and the second endpoint in the right group (as they are undirected, the order does not matter in the input).

When performing the contraction, we sort the edges in the middle subproblem by the first endpoint and scan both the result from the left subproblem and the middle edges in parallel. While we do so, we renumber the endpoints in the edges so they contain component numbers instead of vertex numbers. The parallel scan is very similar to merging two sorted sequences. Then we sort the edges by the second endpoint and renumber them using the right result in the same way. Finally, we sort them using both endpoints and remove duplicates by performing a scan over them.

After receiving the result of the middle edge set, we need to perform expansion of the contracted components again. We start with the left result, sort it by the component, while we keep the middle result sorted as it is (because the vertices in the middle result correspond to the components in left and right ones). Now we scan them in parallel and renumber the components in the left result by the result of middle subproblem. We do the same with the right result. Finally, we concatenate all three results together and sort them.

9.2.3 Trivial subproblem

A trivial problem is such where each vertex has degree at most 1. In such case, as we said, we can return the input as output after sorting. But how do we recognize such subproblems? If it has only a single edge, it is trivial. We check for this kind of triviality every time we start a task.

Each time we have no more bits to split by, we check for problem triviality. We create an auxiliary array where we put all the endpoints of input edges (we go through the edges and write two numbers for each edge – the vertex numbers of the endpoints – to the array). Then we sort the array and, by a single scan, we keep only one occurrence of each number that occurred multiple times before. This will give us the numbers of vertices with degree larger than 1. If this array is empty, the problem is trivial, we sort the input and return as result.

If the problem is not trivial, we split the edges to two groups, the ones adjacent to any of the vertices with degree larger than one, and the rest (these are adjacent to two vertices of degree one). We call the whole algorithm recursively on the first part, and when it is solved, concatenate with the second part and sort the result.

To split the edges to the two groups, we sort them by first endpoint, scan in parallel with the array of vertices we found, and mark the ones with first vertex in the array. Then we do the same with the second endpoint. Last, we just split the edges to two groups by the mark (for example by sorting them).

9.2.4 Correctness

It is not obvious the algorithm gives the correct result. We therefore need to prove it.

Theorem 34. *The result returned from the algorithm corresponds to a function describing components in the input graph. Furthermore, the result satisfies the format described above.*

Proof. We'll prove this by induction on the recursion depth. Some of the Lemmata here use the Theorem (by assuming the result of subproblem is valid), but always with fewer levels of recursion to the bottom.

Lemma 8. *The result is a function (there are no two pairs with the same vertex number).*

Proof. In the case of a trivial problem, it holds (by definition of a trivial problem). When we concatenate the result of recursive call when we found out the problem was not trivial after running out of bits to split, we concatenate one function from the smaller instance of the problem (which is a correct function from induction) with a trivial part of the problem and both parts had distinct vertex numbers on their input. The Lemma still holds.

The renumbering in the expansion step can't create two pairs with the same vertex number (because it doesn't change vertex numbers, only component numbers), nor can concatenation of the left and right subproblems, as left and right graphs are vertex-disjoint (we ignore the singleton vertices). It remains to handle concatenation with the middle subproblem.

We notice that there are no pairs in the subproblem results where vertex number would be equal to any component number. There's only one vertex which is equal to the number of any given component. But that vertex must be contained in that component and the pair would have the same vertex and component number, therefore it isn't included in the result.

But the input of the middle subproblem contains only vertices with numbers equal to some components from the left and right results (the rest of vertices are implicitly singleton vertices, forming single-vertex components). The result therefore can't contain any other vertices. But according to the previous paragraph, such vertices have no pair in the results of the left and right subproblems. Therefore it is safe to concatenate them. □

Lemma 9. *The result never contains a pair with both the vertex and the component numbers equal.*

Proof. If the subproblem is trivial, we return the edges we got as an input. As the input graph is a simple graph, there are no loops.

Concatenation can't create such pair in principle, but the renumbering in expansion step could. This could happen if we got a pair (a, b) from the left or right subproblem and an inverse pair (b, a) from the middle subproblem. However, after contraction, the middle subproblem can't contain the vertex a (as noted in the proof of the previous Lemma), and since each component must contain a vertex with the same number, such inverse pair can't be returned. □

Lemma 10. *The result function corresponds to components and each component contains a vertex with the same number.*

Proof. This holds for for singleton vertices through the whole run of the algorithm – no edge mentions them, so they won't get included in any output, implicitly mapping to a component of the same number.

In the case of a trivial problem, it is also easy to see. There are only singleton vertices and single edges. In the case of an edge, one end of it maps to itself (producing no pair) and the other one is mapped to it as well, through the pair representing the edge.

Next we look at the case when we called the algorithm recursively, because the input was a non-trivial problem after running out of the bits to split by. The result of the call is correct (by induction). Concatenation with the trivial part can't break anything, since the trivial part is not connected by any edge to the non-trivial part, therefore the parts are independent.

Otherwise, we consider three kinds of components.

- A component that is completely inside either the left or the right graph. As there are no edges contained in the middle graph, the whole component is a singleton vertex in the middle subproblem (after contraction). Therefore it is not included in the result of the middle subproblem and it is copied unchanged to the result during the concatenation.
- A component that is completely inside the middle graph. A similar argument applies here, the component's vertices are singletons in the left and right graphs (as no edge connects them there). As well, it is just copied unchanged to the result.
- The component lies across multiple subproblem graphs. As the middle graph "connects" the other two, the middle one must be one of those. The results of the left and right graphs return parts of the component. During the contraction step, the edges inside the middle graph are relabeled and they connect the component parts together. These form a component returned from the middle graph. During the expansion step, vertices of the parts are relabelled from the original part number to the final component number. The pairs from the middle part are the vertices which mapped implicitly in the left and right subproblems. Note that there's still one part which is not renumbered, because the number was chosen as the number of the final component.

□

Let us return to the proof of Theorem 34. We proved the result is a function, that it doesn't contain the pairs with vertex and component numbers equal and that it actually describes components. We don't need to prove the result is sorted, we sort it just before returning. □

9.2.5 In-place implementation

Before we start with the estimates, we show a modification of this algorithm that works mostly in-place. It will simplify the complexity analysis.

We notice that after splitting the input into the three subproblems, we no longer need the original input. Therefore we can just reorder the input in the same array without using a new one. Similarly, when we call a subproblem, we no longer need its input, so it can return its result in the same array where the input was. We can to do this, as the result is at most as large as the input.

So we need to show we can reorder the input inside the array and we can do the expansion inside the array (the other operations are working on the array they are given quite naturally).

The splitting can be done in two ways. The simpler one is sorting the edges by the subproblem they belong to and scanning the result to find the places where one subproblem ends and other begins. The more complicated, but slightly faster is the following solution. We first scan the array one time, counting the sizes of each future subproblem. Given the sizes, we know the boundaries of future subproblem inputs. Then we start scanning the array from the beginning again, until we find the first edge that is in the wrong interval. We take it out (leaving a "hole" in the array for a while) and start scanning in the interval of subproblem where it belongs. When we find an edge which does not belong into that subproblem, we take it out, which creates a hole for the edge we took out previously, so we put the previous edge there. We do the same with the new edge. If the new edge belongs to the subproblem where we have a hole already, we fill it up and scan for a new edge we can take out.

The expansion step is simple. We can do the renumbering easily. Then we can compact the three results together by scanning and moving them to the left over any gaps which could appear due to results being smaller than inputs.

Note that this algorithm isn't truly in-place. We need a stack for the recursion and the sorting would likely use some additional memory (there are sorting algorithms that work in-place, for example the well-known heap sort, but they are not optimal in the cache-oblivious model), and the auxiliary array for trivial problem identification. But except for the stack, these are all "ephemeral" – the array is needed to perform some operation, but it doesn't store data for a long time. There's at most one such buffer at a time and each is $\mathcal{O}(E)$, so it won't change the space complexity.

9.2.6 Analysis of complexity

Lemma 11. *There are $\mathcal{O}(\log^2 V)$ levels of recursion.*

Proof. Because we keep splitting by the divide and conquer principle described in Section 5.1.3 until we run out of bits and there are only $\log V$ bit positions, we check for trivial problem every $\log V$ levels of recursion.

Now imagine for a while we don't do any contractions. In this case we would have a trivial input after running out of the bits by the first time (see Lemma 2).

However, if we find out the problem is not trivial, this must have been broken somehow. The only place this can be broken is when we do contractions – we can change the number on an end of the edge, which changes even the bits we already split by.

We do the contractions only when going into the middle subproblem. And, to change a single original number to a different final one, we need at least one edge to be present in some of the left or right subproblems on that level (or, we need a result from the subproblem, but that originates from the edge). Therefore, for each edge that proceeds to the non-trivial part, there's another edge somewhere else that does not happen to be in our input. This means that the number of edges decreases at least by factor of 2 between two checks for triviality. Therefore there are at most $\log E$ such checks, and as $\log E \in \mathcal{O}(\log V)$, we can say there are $\mathcal{O}(\log^2 V)$ levels of recursion total. □

Theorem 35. *The algorithm needs $\mathcal{O}(E + \log^2 V)$ memory.*

Proof. The input is $\Theta(E)$ large. We can use a sorting algorithm that has a linear space complexity (for example Funnelsort from Section 6.3). We already discussed how to reuse the space where the input was in the previous section.

The $\log^2 V$ is for the stack of recursion. Note that the input can be actually smaller than this, for example two edges in a really large graph, that end up in the same recursion leaf would need $\Theta(\log V)$ levels of recursion. □

Theorem 36. *The algorithm's run time $\mathcal{O}(E \cdot \log^3 V)$ in the RAM computing model.*

Proof. We have $\mathcal{O}(\log^2 V)$ levels of recursion. As each edge is in at most one task on one given level, the size of all the tasks on one level is $\mathcal{O}(E)$.

Each task consists of several sorts and scans. So, time needed for single task is $\mathcal{O}(E_t \cdot \log E_t)$, where E_t is the size of the task. As this function is sub-additive, each level takes at most $\mathcal{O}(E \cdot \log E)$ time.

As $E \in \mathcal{O}(V^2)$, we can write that the time for single level is $\mathcal{O}(E \cdot \log V)$. Therefore, the total time over all levels is $\mathcal{O}(E \cdot \log^3 V)$. □

Theorem 37. *The algorithm requires $\mathcal{O}\left(\mathrm{sort}(E) \cdot \log^2 V\right)$ memory transfers.*

Proof. As proven by Lemma 11, there are $\mathcal{O}(\log^2 V)$ levels of recursion. A task of size n takes $\mathrm{sort}(n)$ memory transfers (as argued in the proof of Theorem 36, it consists of several scans and sorts). A task is split into 3 subtasks of total size no bigger than the input. We can use the Lemma 1 and get the result. □

This algorithm is expected to have a reasonable complexity in the streaming model as well, as it only scans and sorts. However, we do not compute the complexity here.

It would be possible to run the left and right subproblems in parallel, however the middle subproblem must wait for them to finish. We can run the sorts with multiple threads.

9.2.7 Comparison

It might look like $\mathrm{sort}(E) + V$ is usually smaller than $\mathrm{sort}(E) \cdot \log^2 V$. It is clearly so with dense graphs. When the graph density $\rho = E/V$ is larger than L, the BFS algorithm is obviously faster. However, it might not be so with sparse graphs.

So we look where the border lies.

$$\begin{aligned}
\frac{E}{L} \cdot \log^2 V \cdot \log_{\frac{M}{L}} \frac{E}{L} &= V && (9.1)\\
\frac{E}{V} \cdot \log^2 V \cdot \log_{\frac{M}{L}} \frac{E}{L} &= L && (9.2)\\
\rho \cdot \log^2 V \cdot \log_{\frac{M}{L}} \frac{E}{L} &= L && (9.3)
\end{aligned}$$

If we assume that the density is bounded by a constant (which isn't uncommon in practice, consider for example planar graphs), then the break-even point is when $\log^2 V \cdot \log_{\frac{M}{L}} \frac{E}{L}$ is within constant factor of L. As $\log E \leq 2 \cdot \log V$ and the base of logarithm is only a constant factor, we can estimate the point by $\log^3 V$. In case of a typical processor cache, where the size of page is at the order of bytes, the BFS-based algorithm is clearly better. But when we use the main memory as a cache for the disks and the block size is at the order of megabytes, we need graph with around 2^{100} vertices or more for the BFS algorithm to win.

9.3 Reachability

Reachability is an analog of components for directed graphs. But as the graph is directed, a different set of vertices is reachable from each vertex. Therefore the result can be either list of vertices for each vertex or a matrix (two-dimensional array) of booleans, where the value at index (i, j) says if vertex j is reachable (a path exists) from i. This matrix is also called the transitive closure of the graph.

We could run a DFS or directed BFS from each vertex to create the result. This would result in $\mathcal{O}\left(\left(V + \frac{E}{L}\right) \cdot V \cdot \log V + \mathrm{sort}(E)\right)$ memory transfers. We can simply run the DFS algorithm from Section 8.3 V times. Notice we need to sort the edges only once at the beginning of the algorithm and reuse this sorted array for each DFS run.

We can take another approach based on matrix multiplication. The algorithm is described by Mareš [18].

The graph itself can be represented by its slightly modified adjacency matrix. The matrix will be $V \times V$ large and contain 1 on the position (i, j) if there's an edge from vertex i to j or if $i = j$. It contains 0 otherwise. Let's call this matrix A.

Theorem 38. *The value $\left(A^k\right)_{i,j}$ is non-zero if and only if there's a walk of length at most k from i to j.*

Proof. We prove it by induction. The A^1 case holds, edges are walks of length 1 and the diagonal holds walks of length 0 from each vertex to itself.

Now, assume it holds up to A^{l-1} and we want to prove it for A^l. $A^l = A^{l-1} \cdot A$. If we look at the value at (i, j), it is $\sum_{v \in V} \left(A^{l-i}\right)_{i,v} \cdot A_{v,j}$. The $(A_{l-1})_{i,v}$ is non-zero if there's a walk from vertex i to vertex v of length at most $l - 1$. The $A_{v,j}$ is one if there's an edge from v to j or if $v = j$. So, together it represents a walk that goes to the vertex v and then by single edge continues to the vertex j, or in case $v = j$, it stays there. This is clearly a walk of length at most l.

If the $A_{v,j} = 0$, then there's no way to get from v to j by at most one edge. Therefore the result correctly doesn't get increased by a walk through this vertex. Similarly if $A_{i,v} = 0$.

The sum iterates through all the vertices, therefore we try all the possible last edges of the walk.

To complete the proof, we notice the values in the matrix are never negative, therefore summing the values can produce zero only if all values are zero. □

We can use this to build an algorithm. We take A^k, where $k \geq |V|$ (if there's a walk, there must be a path and that one can't be longer than $|V|$ vertices) and change each non-zero number to one. This will produce the result.

To lower the number of multiplications we need to do, we can compute $\left((A^2)^2\right)^{\cdots}$. This way we need to do only $\lceil \log |V| \rceil$ multiplications to get large enough power of A.

To avoid working with large numbers, we reset the non-zero values to 1 after each multiplication. This can be done by a single scan, which can't worsen any of the bounds.

It is clear that the algorithm needs $\mathcal{O}\left(\text{mult}_t\left(V\right) \cdot \log V\right)$ memory transfers, where $\text{mult}_t(n)$ is the number of memory transfers needed for multiplication of square matrices of size n. Similarly, it takes $\mathcal{O}\left(\text{mult}_r\left(V\right) \cdot \log V\right)$ RAM running time (where $\text{mult}_r\left(n\right)$ is time needed to multiply matrices). It takes $\mathcal{O}\left(V^2\right)$ memory, as we don't need to store the intermediate results, only the operands and the result of the current multiplication.

If we used the Strassen's algorithm, the algorithm would take $\mathcal{O}(V^{\log_2 7} \cdot \log V)$ time and it would cause $\mathcal{O}\left(\frac{V^{\log_2 7} \cdot \log V}{L}\right)$ memory transfers.

9.4 Experimental results

We tried several algorithms for computing the connected components. We do it on undirected graphs. The reachability algorithm from previous section is not included, as it would be an unfair comparison.

The tested algorithms are:

- `componentsDfs` – this is the classical algorithm using classic RAM DFS with a mark in each vertex. The marks are reset to unused only once per the whole run of algorithm, not each time the DFS is started.
- `componentsDfsBRT` – this is the classical algorithm, but the used DFS is based on the BRT/BPT, as described in Section 8.3. The BRT and BPTs can be reused – they can be created just once, the search will use only the BPTs in the current component, leaving the others intact.
- `componentsBfsSort` – this is again the classical algorithm, but it uses the sort-based BFS from Section 8.3.1 to identify a component.
- `componentsDC` – the divide and conquer algorithm from Section 9.2.

The `Threaded` versions use multithreaded version of quicksort to sort data. Running the left and right subproblems in parallel was not implemented.

As we see in Figures 9.2 and 9.3, the RAM time complexity plays more important role than the number of memory transfers. The classical algorithm with classical DFS performs best. It is not surprising that both the BRT based algorithm and the divide and conquer one perform rather badly. The sorting BFS version is slightly better (especially in the threaded version with less dense graphs).

However, as expected, the divide and conquer algorithm performs better on sparse graphs than the BRT based one.

When we look at the number of cache misses in Figure 9.4, we see that the divide and conquer algorithm is pretty good at smaller inputs, together with the sorting BFS one and RAM one. However, unlike them, it gets worse when the depth of recursion increases, which is to be expected. Both the divide and conquer algorithm and the BRT-based algorithm benefit from long pages more than from the size of the internal memory. As we see in the Figure 9.5, which is number of cache misses in case of cache lines of 1024 bytes instead of 64, they get slightly better. However, the sorting BFS one still wins in this case.

To provide an additional benchmark, we look at the runtime when the memory is a cache for disk with 8192 pages per 4096 bytes (in Figure 9.6). The used input is a sparse graph with $c = 3$ (see Section 4.2.3). In this case, the divide and conquer and the sorting based algorithms are reasonably good (considering large inputs). The

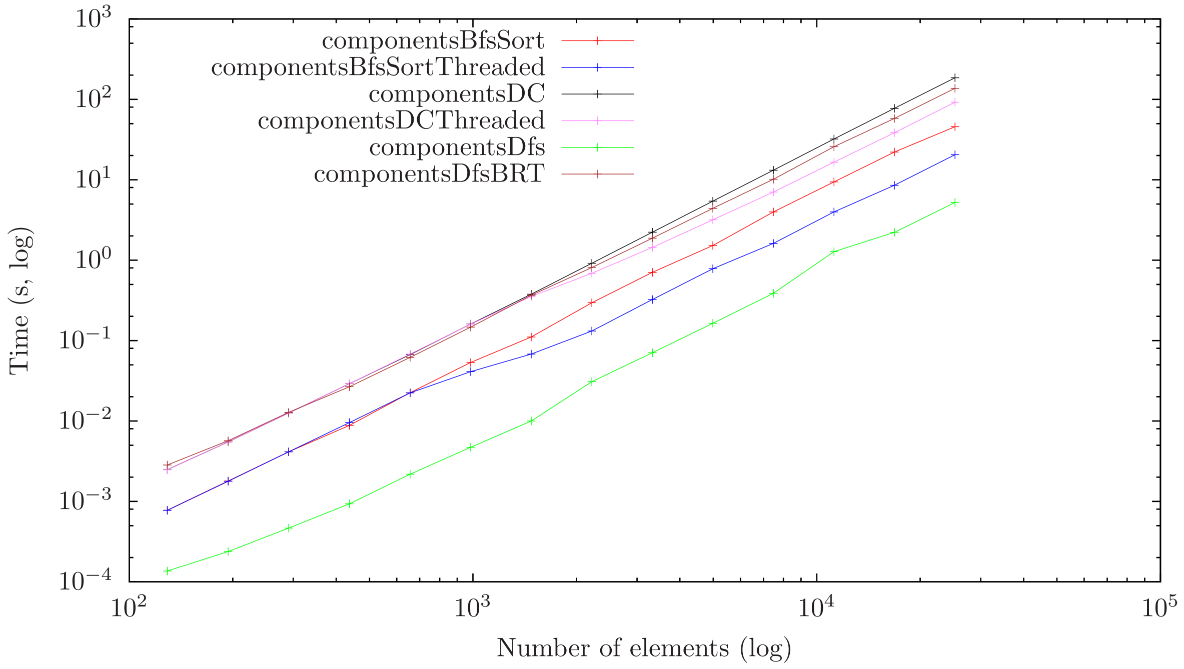

Figure 9.2: Time of components computation on a dense graph (10 iterations)

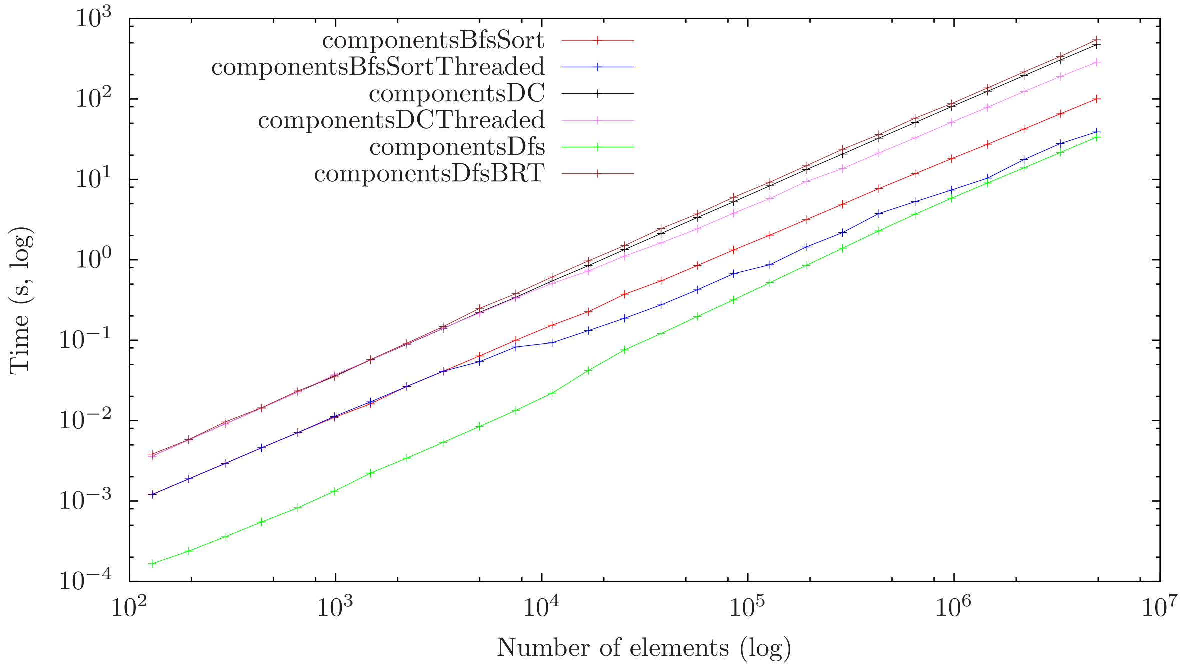

Figure 9.3: Time of components computation on a tree-like graph (10 iterations)

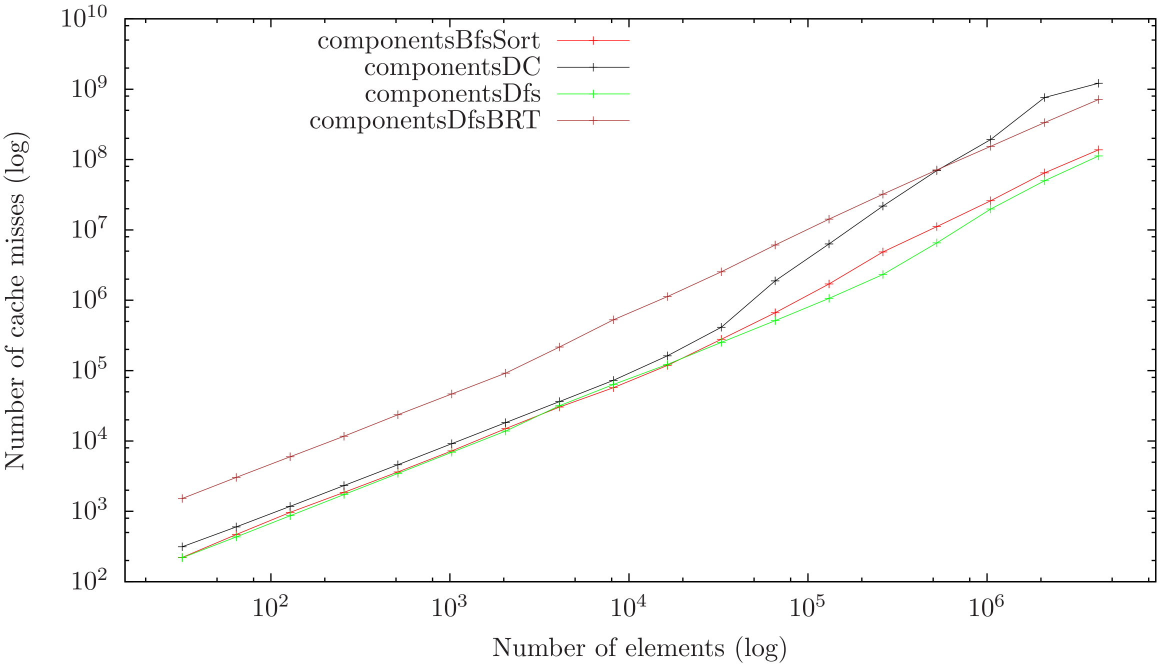

Figure 9.4: Number of cache misses on a sparse graph

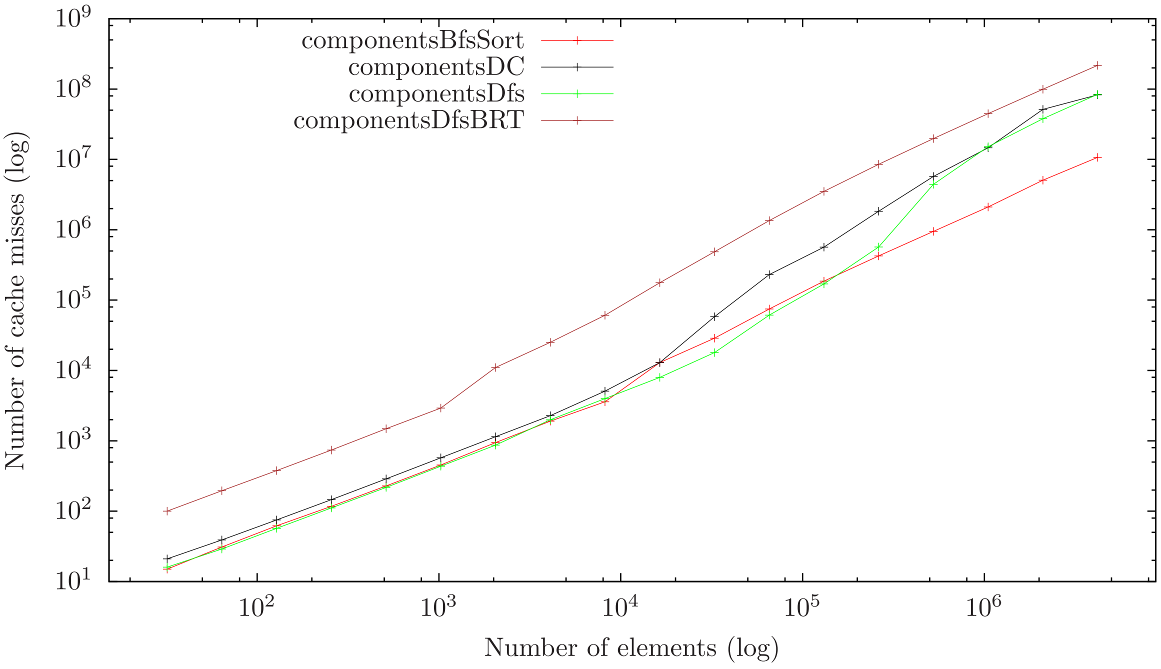

Figure 9.5: Number of cache misses on a sparse graph with long cache lines

other two algorithms get significantly slower when the vertices don't fit into the internal memory. We can guess that, while the sorting based algorithm does have a $|V|$ additive term in the bound, the list of vertices is effectively scanned once per a produced level and the number of levels is reasonably small, therefore it performs better than the algorithms accessing the vertices in a chaotic order.

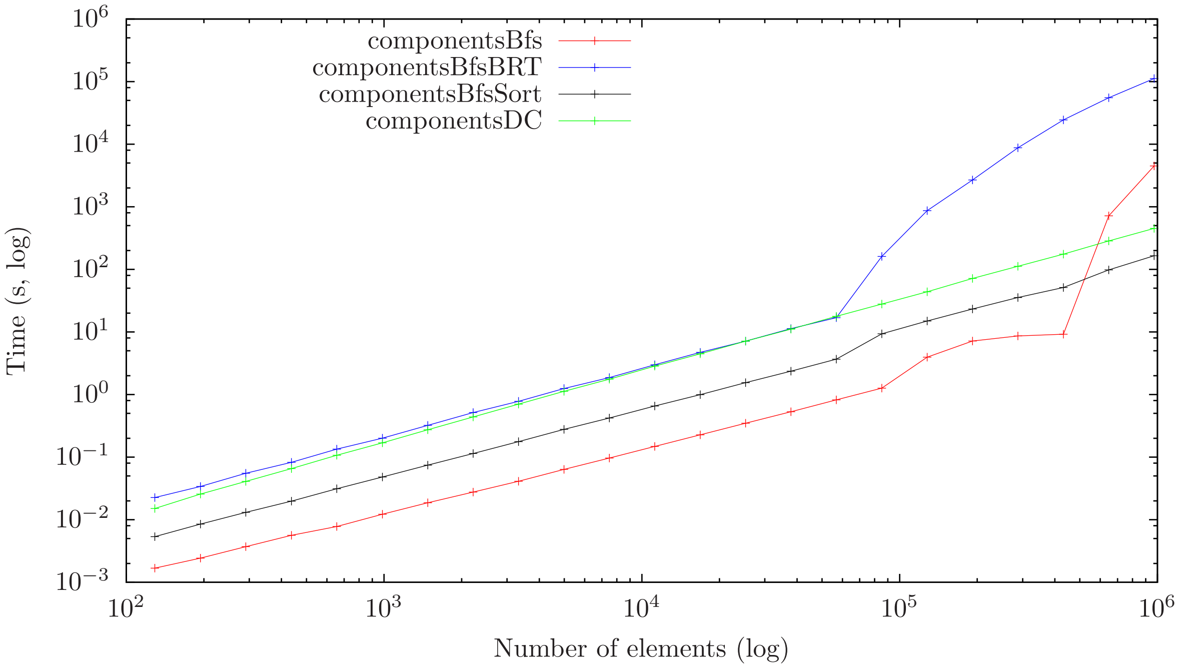

Figure 9.6: Runtime of components when external memory is disk

10. Maximal matching

A ***matching*** is a subset of edges of an undirected graph such that no two edges share a common vertex.

A ***maximal matching*** is such a matching which is subset-maximal. Adding any other edge would break the condition and the edge would share a vertex with a different edge. Not to be confused with ***maximum matching***, which is a matching of the biggest possible cardinality. The maximal matching is easier to compute, as each maximum matching is also maximal but not necessarily the other way around.

10.1 RAM approach

The problem is trivial in the RAM model. We will compute the matching greedily. Each vertex will contain a variable where it'll keep its degree in the matching we produce. As we start with an empty matching, all the variables will be initially zero.

Then we go through all the edges (in any order) and try adding each of them. If both endpoint vertices have degree zero, we output the edge as part of the matching and increase the degrees in the endpoint vertices. If at least one of them has non-zero degree, we skip this edge.

It is obvious this computes a maximal matching, as each edge we haven't included would create a vertex of degree two.

It takes $\mathcal{O}(V+E)$ RAM time and $\mathcal{O}(V/L+E)$ memory transfers. We need to initialize the array of degrees and each lookup into a vertex might cause a cache miss. It needs $\mathcal{O}(V)$ additional memory for the degrees of vertices, the input graph is not included in the estimate.

10.2 With BRT

We notice we spend a lot of time trying out edges which are not used later. We'll look into this problem here.

We keep a flag if a vertex was used in each of the vertices. We also add one global buffer repository tree (from Section 8.1). The tree will be used to notify vertices we would scan in the future about neighbors which are already used.

All vertices start as unused and the buffer repository tree is empty. Then we scan all the vertices in the order of their indices.

If the current vertex is used, we ignore it and continue with another one. When it is unused, we try to find an edge with the other side still unused. We will ignore all the edges going to vertices with lower indices (because such edge was examined in the other direction already). We extract all the already used neighbors from the BRT. Then we sort both our edges and the neighbors and by scanning them in parallel find the first unused one or conclude all of them are already used.

We output the edge and mark the other vertex as used. The current vertex does not have to be marked, since it'll never again be examined. At the end, all the neighbors of the other vertex which will still be scanned need to be notified the other vertex is already used. So we walk the edges and put a message for each of the recipient vertices into the BRT about the vertex being used, similarly to how we did in Section 8.3.

This is a simple high-level pseudo-code of the algorithm.

```
messageboard = BRT(graph.vertexCount)
result = emptyList
for each vertex from graph.vertices:
  vertex.used = false
for each vertex from graph.vertices:
  if not vertex.used:
    usedNeighbors = messageboard.get(vertex)
    unusedNeighbors = vertex.neighbors - usedNeighbors
    removeAllSmallerThan(unusedNeighbors, vertex)
    if not unusedNeighbors.empty:
      other = unusedNeighbors[0]
      other.used = true
      result.append({vertex, other})
      for each neighborToNotify in other.neighbors:
        messageboard.insert(neighborToNotify, other)
```

Theorem 39. *This algorithm computes maximal matching.*

Proof. We need to prove two things. First, if we add any edge to the final result, there'll be a vertex with degree bigger than one. The second is that there's no vertex with degree greater than one in the result.

Imagine we can add an edge $e = \{u, v\}, u < v$ which is not in the result. Why didn't we output the edge in the first place?

There are several possibilities. The first one is that we never really examined the edge. That could have happened if we skipped the vertex u (we never look at the other direction of edge). But we could have skipped u only because it already had an edge, therefore adding e would mean u would have degree bigger than 1.

Another possibility is that we looked at u and we used a different edge going out of u. But adding e now would again mean u would be shared by two edges.

And the last possibility is that we looked at u and we didn't choose this edge because we got v from the BRT. But that means v already had an edge at the time, so adding e would mean v is shared by two edges.

Now imagine we got a degree at least two. We were adding one of the edges with shared vertex, $e = \{u, v\}, u < v$. Let's look into the possible cases.

- The vertex u is the one with the big degree and the e was added as first. But that means we left u and continued scanning vertices with bigger indices. Any edge leading "back" to u would be skipped.
- The vertex u is the one with the big degree and there was an edge $f = \{u, w\}$ already when we added e. But as we added the edge before visiting u, $u > w$. In this case, u would have been marked, so we would skip u completely.
- The vertex v is the one with the big degree and e is added as first. This means we mark v (and no edge $f = (v, w), v < w$ will be added) and all the relevant neighbors get a message through the BRT not to use v (therefore no edge $f = (v, w), v > w$ will be added).
- The vertex v is the one with the big degree and e is added when there's an edge $f = (v, w)$ already. If $v < w$, the f would not be examined yet when adding e, so $v > w$. But in this case, the BRT contains a message for u not to use v, so we can't add e.

□

Theorem 40. *The algorithm needs $\mathcal{O}(V + E)$ additional memory.*

Proof. We need the flags signalling whether a vertex is used. These take $\mathcal{O}(V)$. We also need a buffer repository tree with vertices as the keys and at most E messages inside (we can store only one direction of an edge). The buffer repository tree takes $\mathcal{O}(V + E)$ memory by 19. □

Theorem 41. *The algorithm takes $\mathcal{O}(V + E \cdot \log V)$ RAM time.*

Proof. The scanning through vertices alone takes $\mathcal{O}(V)$ time. Each vertex then sorts its own edges, picks up the messages from the BRT (there can be at most as many as it has edges) and sorts them. Because sorting takes $\mathcal{O}(n \cdot \log n)$ time and that function is sub-additive, the total time spent for this over the runtime of the algorithm is $\mathcal{O}(E \cdot \log E)$. Because $E \in \mathcal{O}(V^2)$, we can write this is $\mathcal{O}(E \cdot \log V)$. The scanning of edges takes less time than the sorting. If there's a yet unused neighbor, the neighbor is marked (which takes constant time) and at most one message is inserted for each of its edges. As a vertex is marked as used at most once, we'll insert $\mathcal{O}(E)$ elements to the BRT, which will take $\mathcal{O}(E \cdot \log V)$ time, by Theorem 20. □

Theorem 42. *The algorithm issues* $\mathcal{O}\left(\left(V+\frac{E}{L}\right)\cdot\log V\right)$ *memory transfers.*

Proof. We'll use Theorem 21.

The scanning of vertices take $\mathcal{O}\left(V/L\right)$ transfers.

The sorts of local edges takes $\mathcal{O}(\mathrm{sort}(E))$ over the runtime of the algorithm (if we put the partial pages aside for a while). When we extract the used vertices from the BRT, there are at most as many as the edges, so sorting them takes $\mathcal{O}(sort(E))$. We observe that $\mathrm{sort}(E)\in\mathcal{O}\left(\frac{E}{L}\cdot\log E\right)\subseteq\mathcal{O}\left(\frac{E}{L}\cdot\log V\right)$. We add the partial pages back – but there are at most constant amount of them for each vertex – and get we $\mathcal{O}(\frac{E}{L}\cdot\log V+V)$.

We pay $\mathcal{O}\left(\log V\right)$ for a single extraction, so we pay $\mathcal{O}(V\cdot\log V)$ over the whole runtime.

Each marking of the neighbor as used is a potential cache miss. Inserting the edges of each vertex into the BRT takes $\mathcal{O}\left(1+\frac{E_v\cdot\log V}{L}\right)$ transfers. So, that is $\mathcal{O}\left(\frac{E}{L}\cdot\log V+V\right)$ over all vertices.

Now we just need to sum these parts together to get the claimed bound. □

10.3 Divide and Conquer

We introduce yet another new algorithm, based on the divide and conquer technique described in Section 5.1.3.

10.3.1 Overview

Again, recursive algorithms can be quite complicated, so we start with a high-level overview.

In each step, we divide the vertices as described in Section 5.1.3 and get the three groups of edges, defining three edge-disjoint subgraphs (left, right and middle one).

First, we'll recurse on the left and right edges. This will produce some edges to the final output. It'll also return a sorted list of vertices used in the subproblems.

Using the list of used vertices, we'll remove all the edges from the middle list that have at last one endpoint in the result lists. Then we'll recurse on the remaining edges, to produce the rest of the final output and produce some more used vertices.

While the final output can be just appended to a global array, we need to merge the lists of used vertices to keep them sorted.

The recursion stops when we are sure there are no vertices with degree bigger than 1.

To make the understanding of the algorithm easier, we provide a symbolic pseudo-code.

```
result = emptyList

matching(graph, depth):
  if depth > log(graph.vertexCount):
    result.append(graph.edges)
    return graph.vertices
  else:
    left, middle, right = split(graph, depth)
    leftUsed = matching(left, depth + 1)
    rightUsed = matching(right, depth + 1)
    middle.removeVertices(leftUsed + rightUsed)
    middleUsed = matching(middle, depth + 1)
    return leftUsed + rightUsed + middleUsed
```

10.3.2 Implementation details

As with the algorithm for connected components from Section 9.2, we will not keep explicit list of vertices in the input.

We'll try to avoid sorting in the middle levels of the recursion. To reach the goal, we'll keep two copies of each edge, one in each direction. Each copy of the edge has an "active" flag, which is set to `true` initially. We sort the copies by the their vertex before starting the recursive part of the algorithm.

When we delete edges based on the used vertices, we scan the edges and used vertices in parallel. Both are sorted now, the edges by their first vertex. We don't delete the edges, we just set the active flag to `false`. We consider only the first when scanning the arrays in parallel, which marks only one direction of the edge. It means that an edge can have one direction marked as active, while the other is not.

In the bottom level, when we are sure the vertices have degree at most 1 (as proven in Lemma 2), we flip the edges, so their vertex with the lower index is put first and sort them. This brings the two copies of each edge together, so we can unify them and keep only the ones where both copies are active. If we deleted neither of the copies, both endpoints are still unused. These edges are output as a part of the final result. Then we take all the vertices of all these edges, sort them and return them to the higher level of recursion.

10.3.3 Correctness and complexities

Theorem 43. *This algorithm produces maximal matching.*

Proof. Let us restate the claim. The algorithm finds a maximal matching in the graph defined by the active edges in the input. This is equivalent with the claim of the

Theorem at the top level of the recursion, because the input has no inactive edges.

We'll prove our claim by induction on the depth of recursion. At the bottom level, we have a graph without any vertices of degree greater than one by Lemma 2. Therefore if we take only the active edges, we still have no common vertices. Obviously, the maximal matching in such a graph is the whole set of edges (we have no edge to add and no two edges share a vertex).

When we are at a higher level of recursion, we assume that the claim holds for all the lower levels. We clearly never produce anything that wasn't active in the input, because we just concatenate the outputs of the subproblems.

If we produced two edges sharing a vertex, they can come from one subproblem or from two different ones. The first option is not possible, it would contradict the induction assumption. In the second case, one of the edges must come from the middle group (because the left and right group share no vertices with each other). But we deactivated all edges with a vertex used in the left or right result, so the middle subproblem could not have produced any such edge.

The last thing we need to prove is that we can't add another edge to the result. Assume for a while that we can add an edge e. The edge was active in our input (otherwise it would not be expected in the output). If we put it to the left or right group, it would violate the induction assumption, because the edge could be added to the result of the subproblem. Similarly if we put it in the middle group, but didn't deactivate it. So the last case is when we deactivated the edge. But we deactivate an edge only if at least one of its vertices is already used, so if we deactivated e, it can't be added to our result. □

Theorem 44. *This algorithm needs $\mathcal{O}(E + \log V)$ memory.*

Proof. We need to store the recursion stack, intermediate inputs, and lists of used vertices. We also need to store the result somewhere, but the result is never larger than the input by the very definition of the problem, so it fits into the amount stated.

The recursion is $\Theta(\log V)$ levels deep, because we stop when we have split by all bits in the vertex indices.

We probably could achieve an implementation where we would just keep rearranging and overwriting the input array to store the intermediate inputs and results, like in case of the divide and conquer algorithm for components, but that would be unnecessarily complicated. Instead, we will have several buffers and we'll be switching between them as we go deeper into the recursion.

We can have two buffers of size $\min(2 \cdot |E|, |V|)$ to keep and merge the lists of used vertices. They are swapped on each level of recursion. One of them will be our output buffer (where the current task will store the result), the other auxiliary, where our subproblems will output their results. As the outputs are called one by one, they can concatenate their outputs (we remember where one ends and the other one starts, so a single auxiliary buffer for their output is enough). As all merging is done at the very end, our output buffer can be used the whole time as an auxiliary buffer for our subproblems.

We could do the same with the input if we knew how large the groups of edges would be. For simplicity, we'll have three buffers besides in addition to the input array and we'll split the input into them. Once we do, we can reuse the unused ends of the buffers as auxiliary buffers for the lower levels of recursion. If we recurse on the left subproblem, the rests of the middle and right buffers have enough space, because they don't contain the left input and our input buffer is large enough, which makes three buffers which can be used. □

Theorem 45. *The algorithm runs in $\mathcal{O}(E \cdot \log V)$ time in the RAM model.*

Proof. We can generate the list of edges from the graph in linear time (if the input graph is in a reasonable format), and we are able to sort it in $\mathcal{O}(E \cdot \log E) \subseteq \mathcal{O}(E \cdot \log V)$ time.

We also sort at the bottom level of recursion. However, each edge gets to only one problem at the bottom level and the sorting complexity is sub-additive, so this takes $\mathcal{O}(E \cdot \log V)$ as well. We sort the list of used vertices, but there can be at most twice as many vertices as the input edges.

Finally, there are internal levels of recursion. All the operations there (splitting, making edges inactive, merging the used vertices) can be done in linear time in the size of the input. The size of inputs across all the tasks in one level of recursion is $2 \cdot E$ and we have $\mathcal{O}(\log V)$ levels, so this takes $\mathcal{O}(E \cdot \log V)$ time as well. □

Theorem 46. *The algorithm needs $\mathcal{O}\left(\frac{E}{L} \cdot \log V\right)$ memory transfers.*

Proof. The proof is almost the same as the proof of previous Theorem, but we use the estimates of scan and sort for memory transfers instead of their run time (since the work in single task is done only by scanning).

However, we need two further things. First, $\operatorname{sort}(E) \in \mathcal{O}(E \cdot \log V)$ (since the difference is only that the base of the logarithm gets higher when we increase the size of cache).

Second, we need to use the Lemma 1 to get the result. □

10.4 Experimental results

We again measure the performance of the algorithms. The names are obvious – `maxMatchClassic` is the classic RAM algorithm described in Section 10.1. The algorithm called `maxMatchBRT` is from the BRT-based one from Section 10.2 and `maxMatchDC` is the divide and conquer form Section 10.3.

As our graph representation keeps edges in sorted order, no sorting in the preprocessing phase of the divide and conquer algorithm is needed.

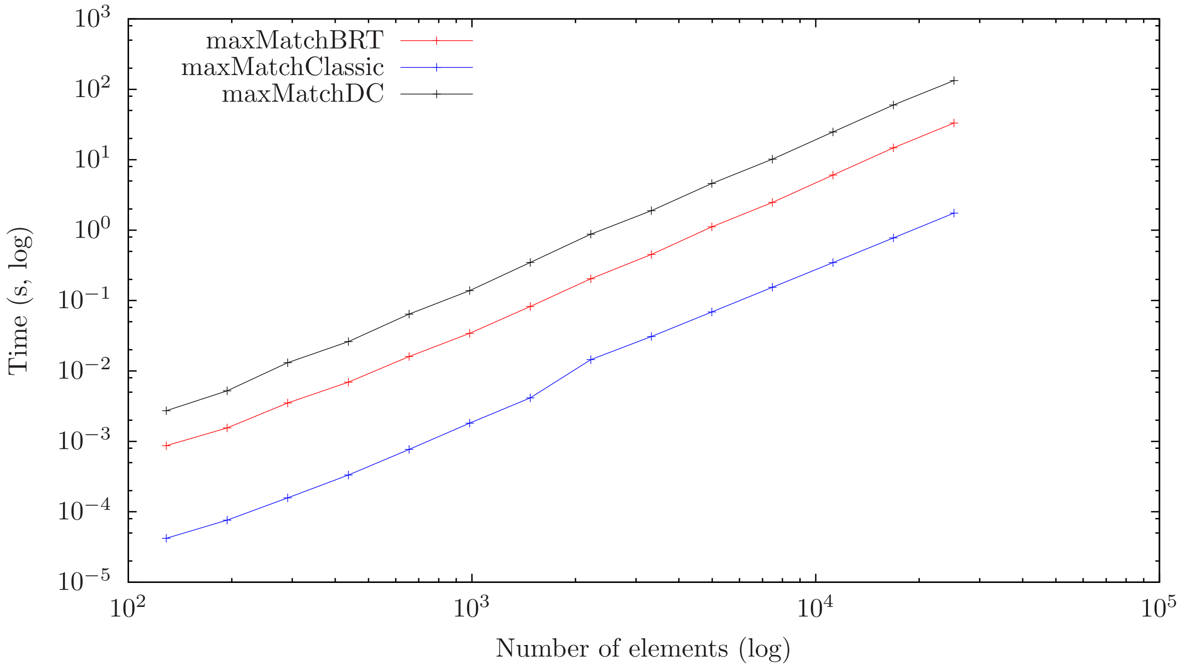

Figure 10.1: Time of run on a dense graph (10 iterations)

As we see in the Figures 10.1 and 10.2, the classical algorithm performs the best. It is expected, as it has better RAM time complexity, even when the theoretical number of cache misses is higher.

However, as we see on the graphs of cache miss counts (in Figures 10.3 and 10.4), the theoretically better bound functions don't help either, as they include additional overhead. Longer pages don't help here either, as can be seen in Figure 10.5.

The classical algorithm seems to win even in the case when the internal memory is used as a for a disk, as can be seen in Figure 10.6.

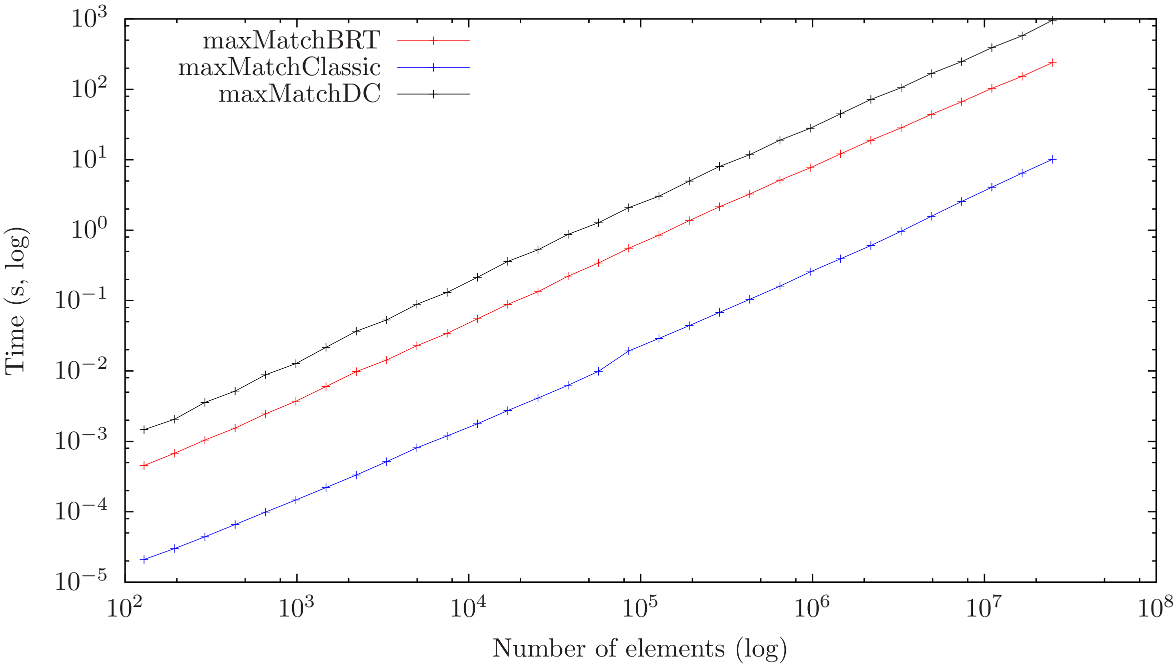

Figure 10.2: Time of run on the triangulation graph (10 iterations)

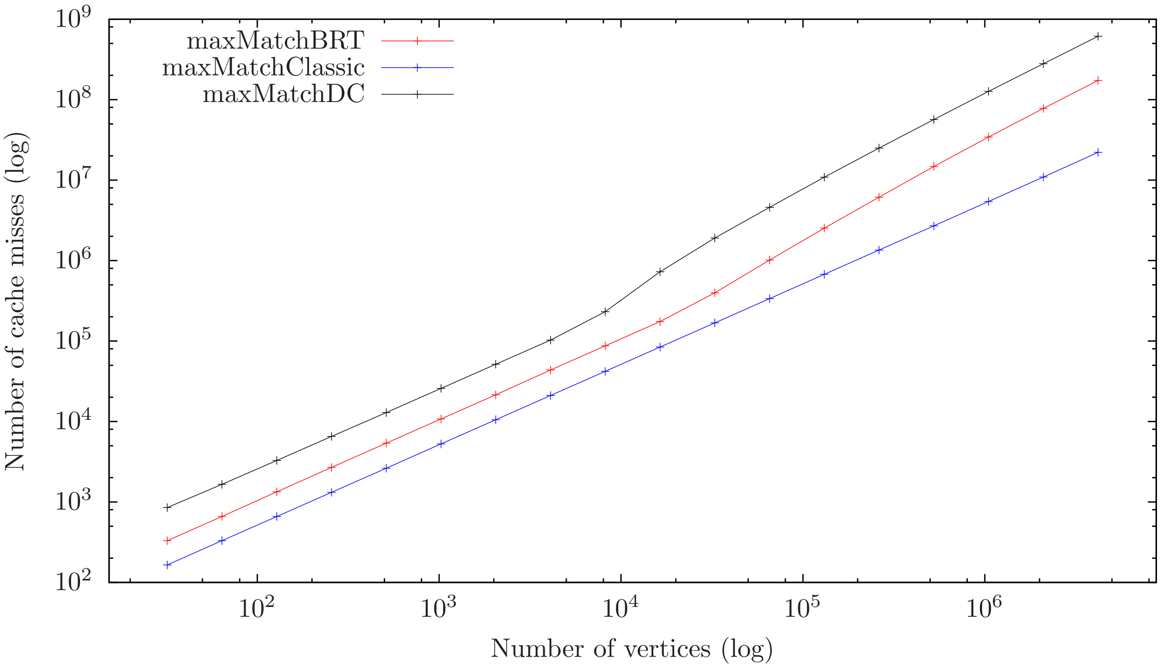

Figure 10.3: Number of cache misses on a dense graph

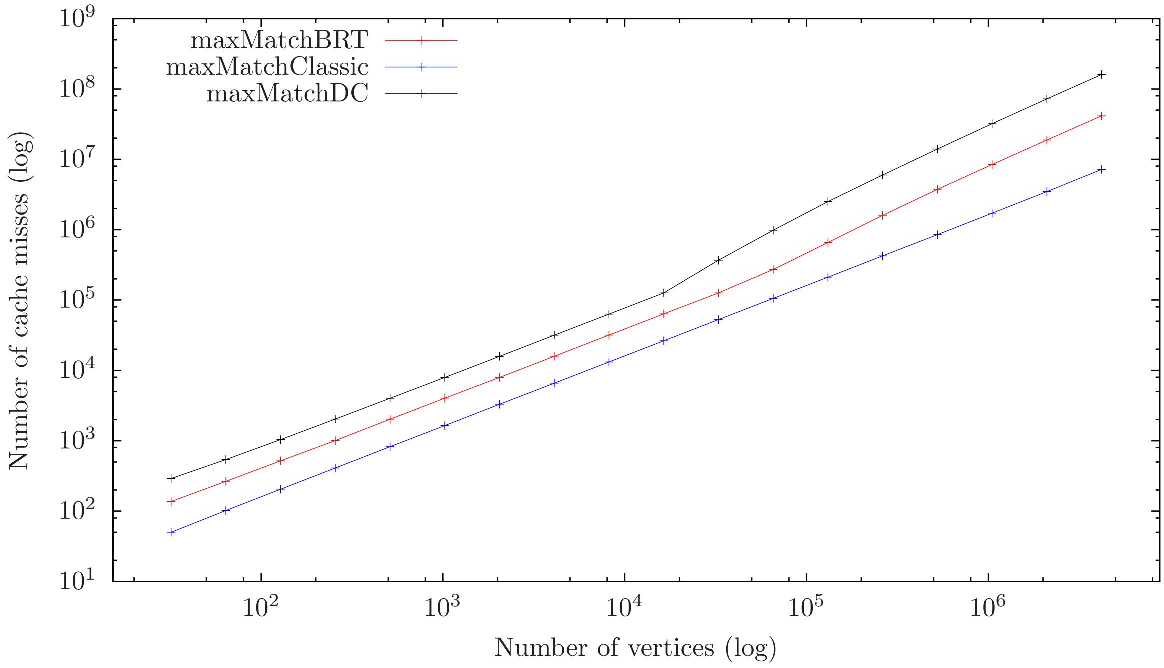

Figure 10.4: Number of cache misses on the triangulation graph

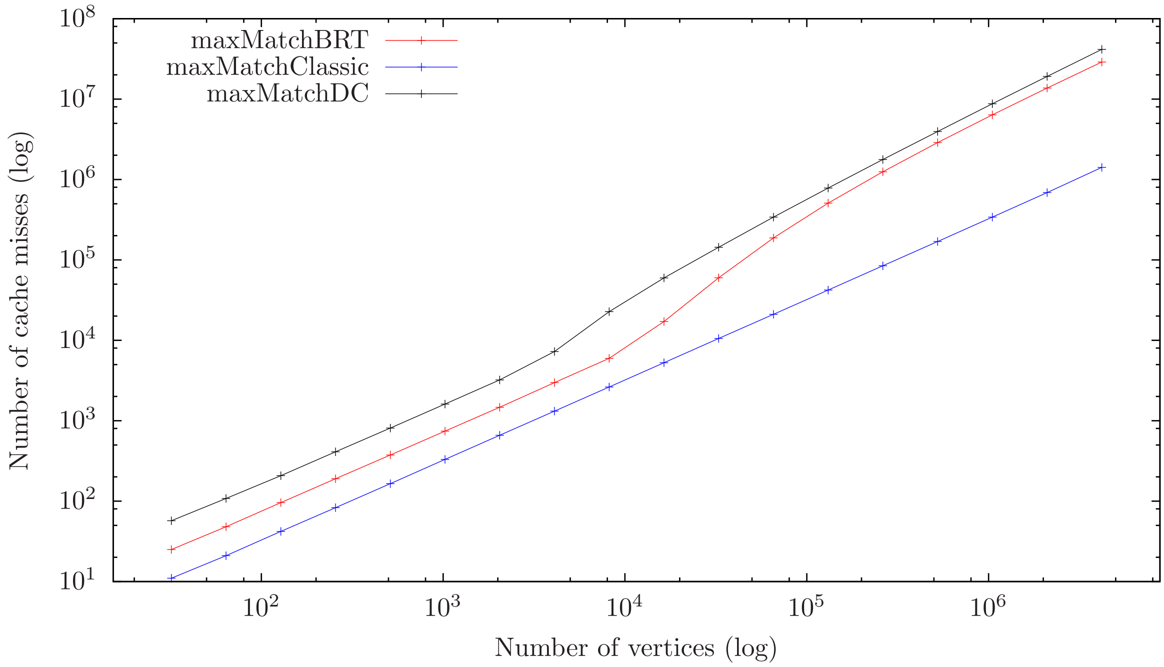

Figure 10.5: Number of cache misses on a dense graph with 1024 bytes long pages

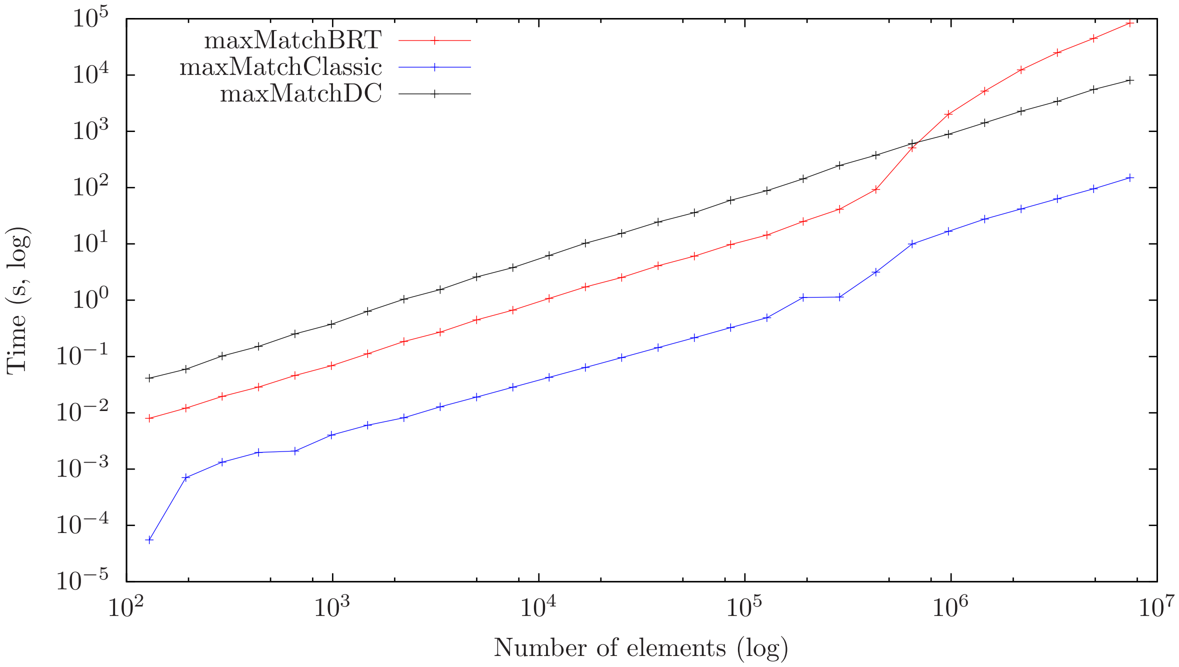

Figure 10.6: Time of running on disk with the sparse graph with $c = 3$

11. Conclusions

We analyzed the asymptotic properties of several algorithms, most importantly their behaviour towards the cache. We compared the estimated behaviour with real-life measurements, both in the case of a cache between the main memory and the processor and in the case where the main memory caches data from disk.

In case of matrix operations, the cache-oblivious algorithms are simple and faster than the classical ones in practice. The case of sorting is slightly more complicated. Funnelsort is slower than Quicksort and it is more complicated. But as shown, Quicksort is also reasonably good at cache utilization, even if it is a classical algorithm. The divide and conquer approach therefore seems to be a good way to design algorithms for the cache-oblivious model, especially if they are hybridized and small subproblems are solved in the usual way.

However, graph algorithms designed with the cache-oblivious model in mind are much more complicated than traditional ones. And, even if their asymptotic number of cache misses is smaller, they are slower. This is because they either have worse runtime complexity or because the slowdown by waiting for data loaded from the memory instead of from the cache is still smaller than the multiplicative constants we omit in the estimates.

Our new algorithms run faster than the previously known cache-oblivious algorithms on some cases of graphs. But this depends on the characteristics of their input, in many cases our algorithms are significantly slower.

In case where the data are stored on a disk and the main memory serves the purpose of a cache, the situation is much better for the cache-oblivious algorithms. As we have both the cache and the lines larger, the difference in the number of cache misses between classical and cache-oblivious algorithm is more visible. Also, the speed difference between cache miss and cache hit is bigger.

Therefore the behaviour in the cache-oblivious model is a good secondary indicator of an algorithm quality, but considering only the memory transfers seems not to produce usable algorithms. It is better to design the algorithm primarily for low run time complexity, but keep the existence of cache in mind.

Bibliography

[1] H. Prokop. Cache-Oblivious Algorithms. Master's thesis, Massachusetts Institute of Technology, June 1999.

[2] Intel Corp. Intel$^{®}$ 64 and IA-32 Architectures Optimization Reference Manual. June 2011.
`http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html`, Retrieved on 2012-04-02.

[3] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. Communications of the ACM, pp. 1116–1127, 1988.

[4] E. Horowitz, D. D. Sleator and R. E. Tarjan, Amortized Efficiency of List Update and Paging Rules. Communications of the ACM, pp. 202–208, February 1985.

[5] U. Drepper. What Every Programmer Should Know About Memory. Red Hat, Inc. November 21, 2007.

[6] D. E. Knuth. *The Art of Computer Programming : Sorting and Searching*. Addison-Wesley, 1997. 780p. ISBN 0-201-89685-0.

[7] P. van Emde-Boas. Preserving order in a forest in less than logarithmic time. Proceedings of the 16th Annual IEEE Symposium on Foundations of Computer Science, Berkeley, California, pp. 75–84. 1975

[8] M. A. Bender, E. D. Demaine and M. Farach-Colton, Cache-oblivious B-trees. SIAM Journal on Computing, volume 35, number 2, pp. 341–358. 2005.

[9] C. A. R. Hoare. Quicksort. Computer Journal, 5(1):10–15, 1962.

[10] I. Parberry. Analysis of Quicksort. Technical Report, Department of Computer Sciences, University of North Texas. 1997.
`http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.52.8765`, Retrieved on 2012-04-07.

[11] G. S. Brodal, R. Fagerberg. Cache Oblivious Distribution Sweeping. Lecture Notes in Computer Science, Volume 2380/2002. 2002.

[12] E. D. Demaine. Cache-Oblivious algorithms and data structures. Lecture Notes from the EEF Summer School on Massive Data Sets, BRICS, University of Aarhus, Denmark. 2002.

[13] U. Drepper, R. McGrath, et al. Source code of GLIBC, the GNU C Library. Free Software Foundation. `http://www.gnu.org/software/libc/`. Retrieved on 2012-04-02.

[14] V. Strassen. Gaussian elimination is not optimal. Numerische Mathematik, Volume 13, pp. 354–356, 1969.

[15] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. Journal of symbolic computation, Volume 9, pp. 251–280, 1990.

[16] V. V. Williams. Breaking the Coppersmith-Winograd barrier. 2011.
`http://www.cs.berkeley.edu/~virgi/matrixmult.pdf`.
Retrieved on 2012-04-07.

[17] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, J. Ian Munro. An Optimal Cache-Oblivious Priority Queue and its Application to Graph Algorithms. SIAM Journal on Computing, 2006.

[18] M. Mareš. Krajinou grafových algoritmů. Second edition in preparation. `http://mj.ucw.cz/vyuka/ga/`, Retrieved on 2012-03-21.

List of Figures

A. Further graphs of experiments

The measured data can be analysed in several ways. But providing too many graphs would pollute the main work. Therefore additional graphs are present in this Appendix.

A.1 Sorting

First, let us look at run time on different input data for the sorting algorithms in Figures A.1, A.2 and A.3. The random one shows only the same part as the other two (as the other were not run on as huge inputs as in Figure 6.3 in Section 6.4).

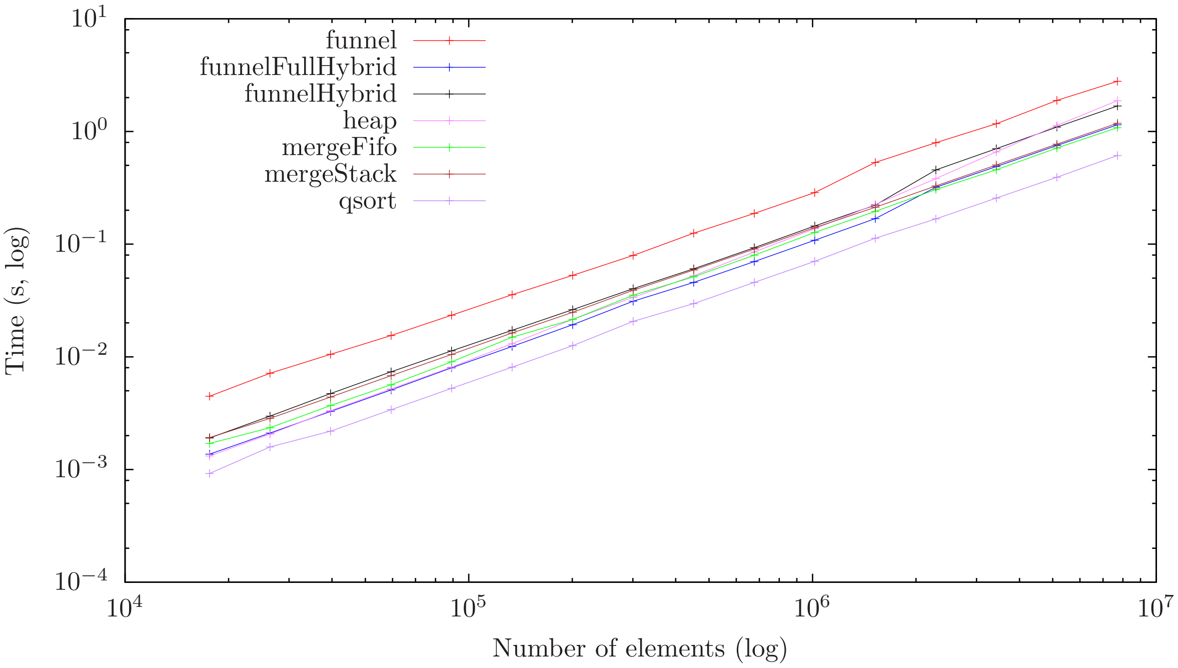

Figure A.1: Runtime of sorting on random input

As we can see, the sorted input slightly helps the quicksort algorithm. The input of all zeroes improves the performance of heap sort (since the elements are all equal and no elements bubble up and down, the algorithm effectively performing two scans of the data only).

We can also look at sorting different data types. The Figures A.4 and A.5 show run times of sorting characters (single-byte values, therefore there are only 256 possible values, which means there are many equal elements) and doubles. The results are slightly different, but the relative performance of the algorithms is similar. The only

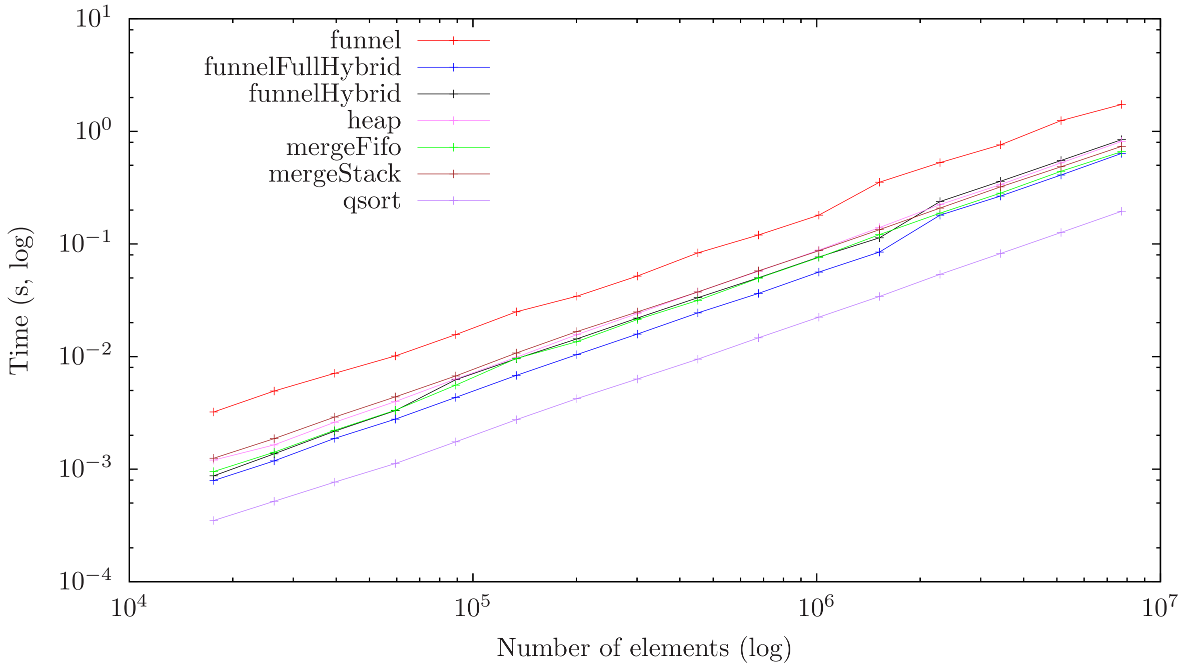

Figure A.2: Runtime of sorting of already sorted input

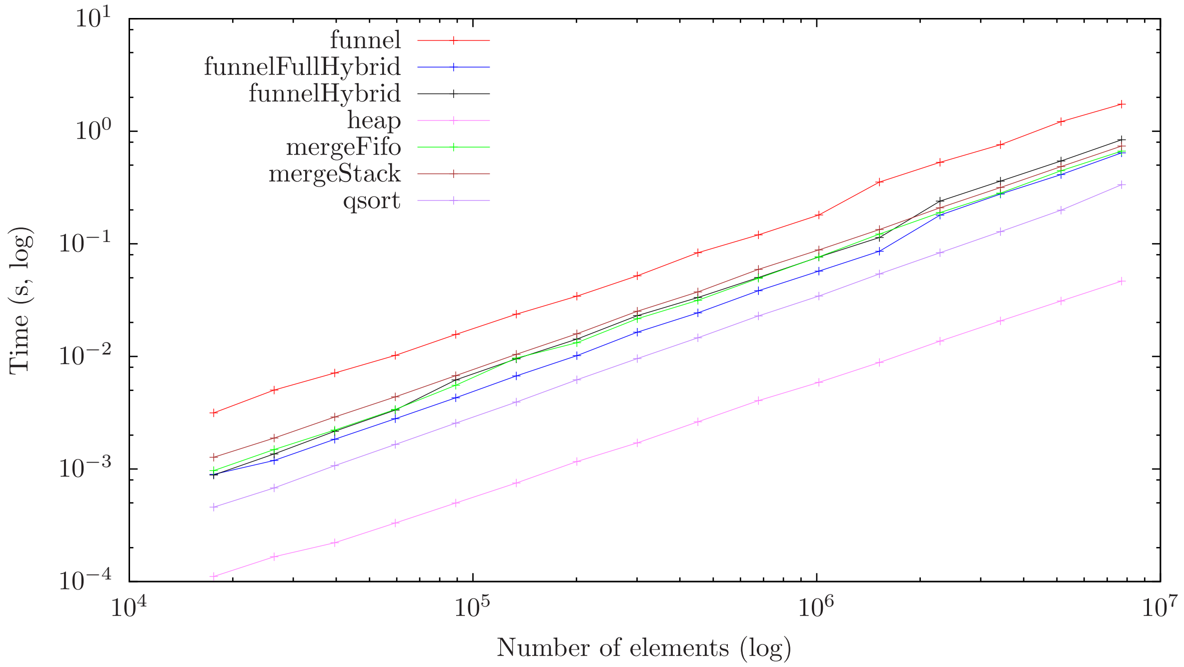

Figure A.3: Runtime of sorting of all zeroes

significant difference is the heap sort is able to take advantage of the small number of possible values in the character case. This also helps the funnelFullHybrid algorithm a little, as that one uses a heap-like structure instead of a funnel of small size.

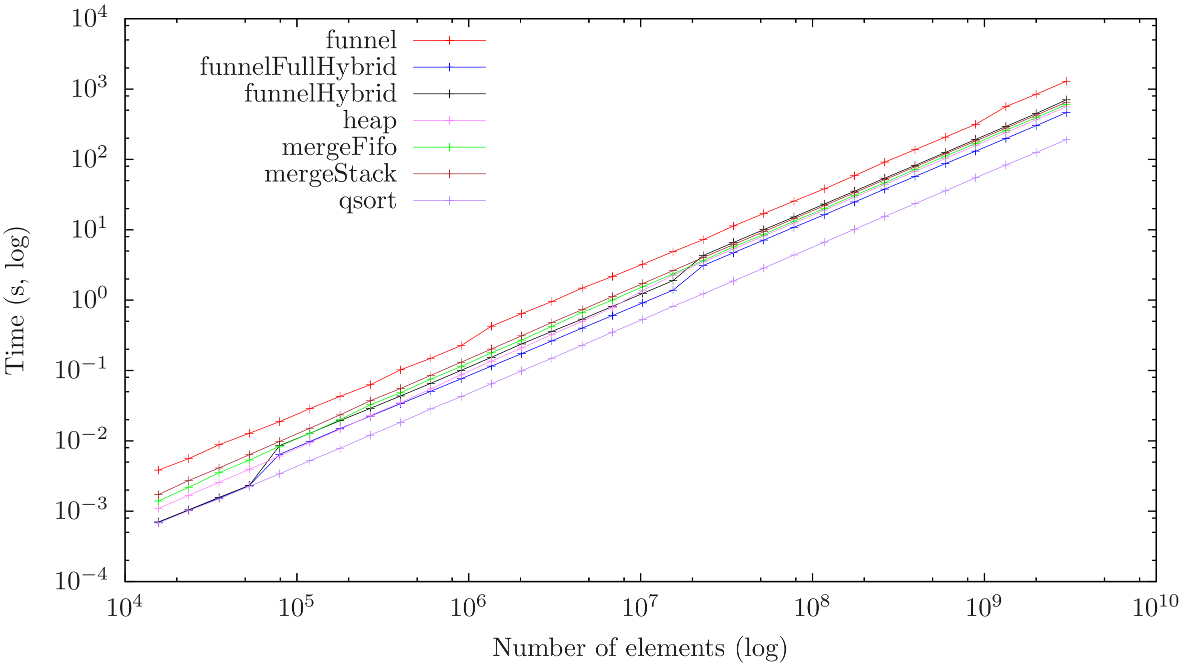

Figure A.4: Runtime of sorting characters

We want to have a look at how the algorithms would react to changes of cache properties. For example, an algorithm using all data from a page loaded into the internal memory would not be influenced much by the size of page, while one that does not will transfer more data (while possibly less pages). This is what the graph in Figure A.6 shows. As we see, only the heap sort is strongly influenced by the size of the page.

Also, when the cache grows in its size, we would like the algorithms to use it to decrease the number of cache misses. The dependency of number of cache misses on the size of cache is shown in Figure A.7 (the size of cache line is fixed to 64). As we see, both the divide and conquer algorithms (quick sort and stack-based merge sort) take benefit of the size. The queue based merge sort obviously doesn't, as it only repeatedly scans the data (it could benefit from a cache large enough to fit all the data). Surprisingly, the heap sort utilizes the cache as well, possibly because the "top" of the heap is kept in the internal memory, decreasing the number of cache misses in the lower levels. It is more complicated with the funnel sorts. It seems the funnels it uses fit completely from some size of the cache and benefit from the size no more, as the funnel inputs are effectively scanned.

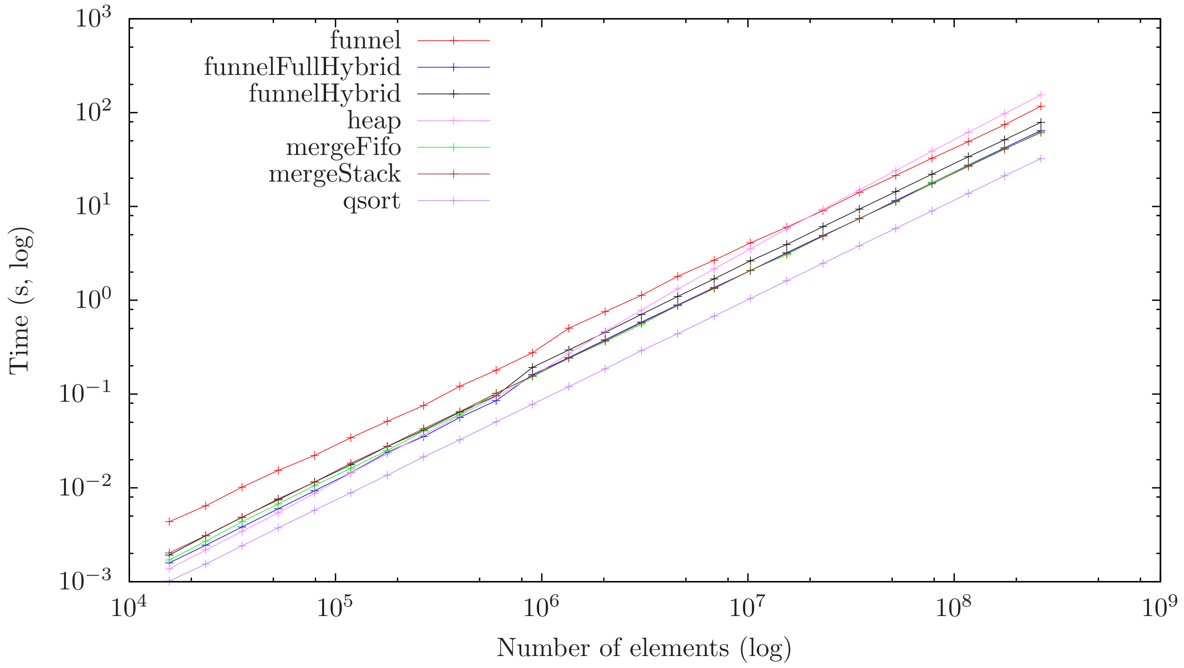

Figure A.5: Runtime of sorting doubles

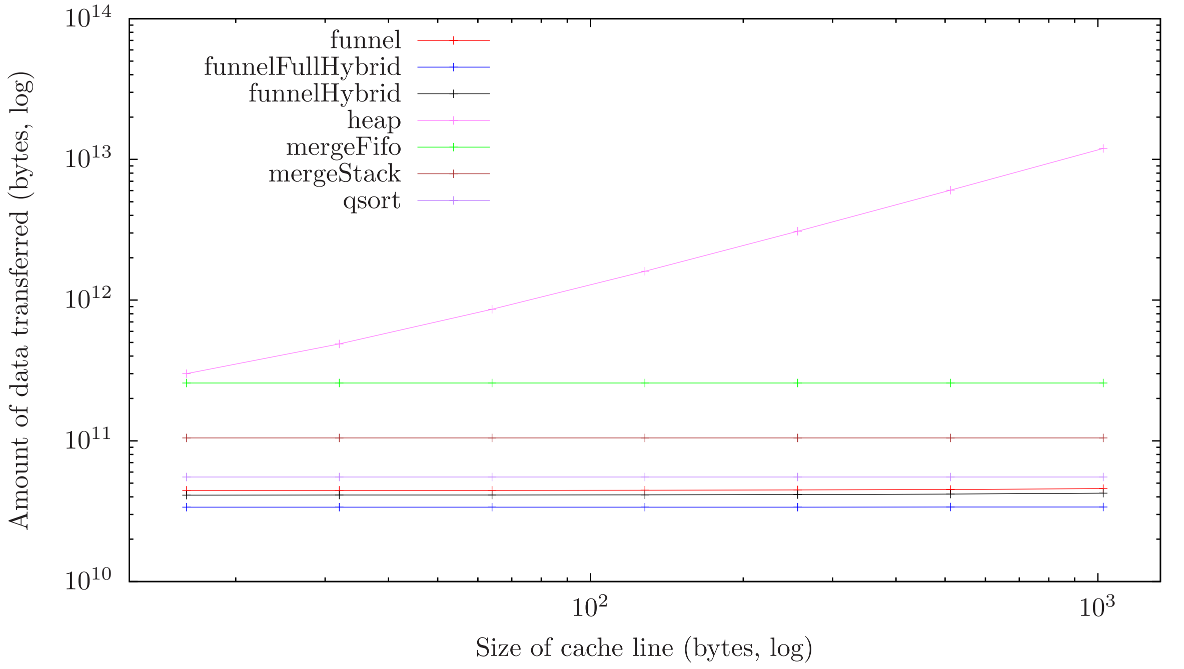

Figure A.6: Dependency of number of bytes transfered on page size

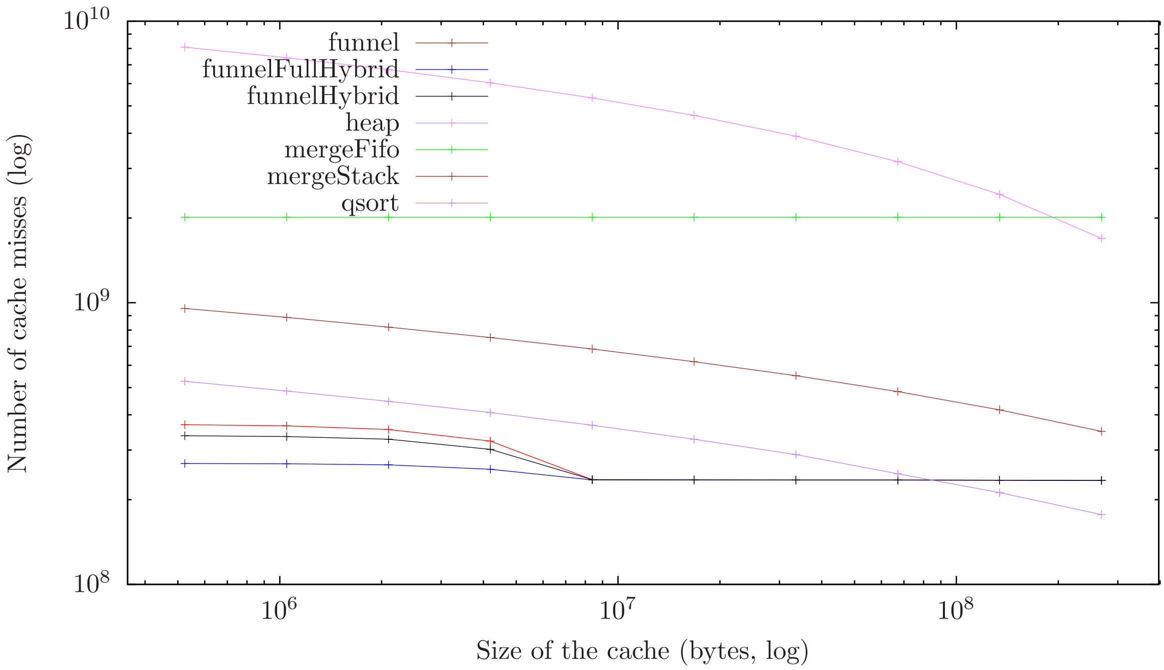

Figure A.7: Number of cache misses depending on the cache size

A.2 Matrix multiplication

There isn't much more to show about matrices. We pick only the graph of how well the algorithms take advantage of the cache size. The graph in Figure A.8 is for matrix of 4096 × 4096 elements with 16byte long cache lines.

A.3 Graph searching

First, let us show the run time and cache misses of searches on a triangulation graph in Figures A.9 and A.10. We don't show the case with tree-like input, as that one is misleading – the graph usually consists of several components of different sizes and only chance influences which component is searched or how large it is.

We again have a look at how the algorithms take benefit from the cache size (in the Figures A.11, A.12 and A.13). It seems the sorting based algorithm takes the best advantage of bigger cache. Also, the sorting based algorithm uses the data from a cache line with most efficiency of the algorithms, as shown in Figures A.15 and A.16. If the sizes of CPU caches and number of cores grow, we could expect the sorting based algorithm to became faster than the classical ones in certain situations, like planar graphs. As we see in Figure A.14, with enough edges the cache line utilization

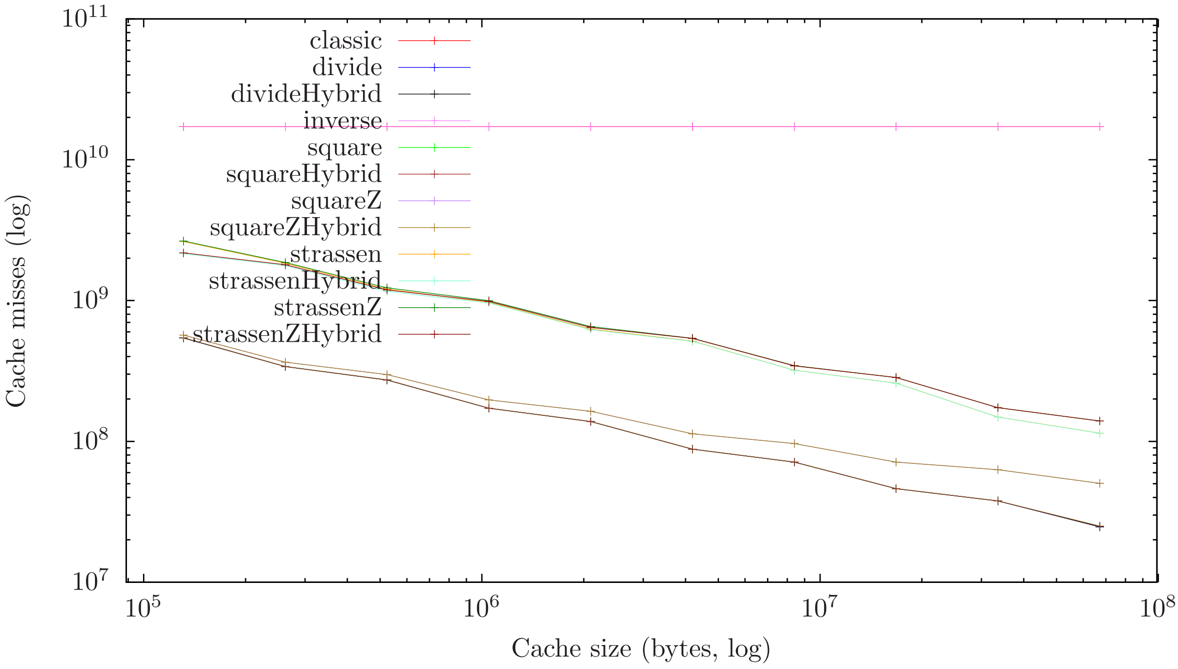

Figure A.8: Cache size utilization by matrix multiplication algorithms

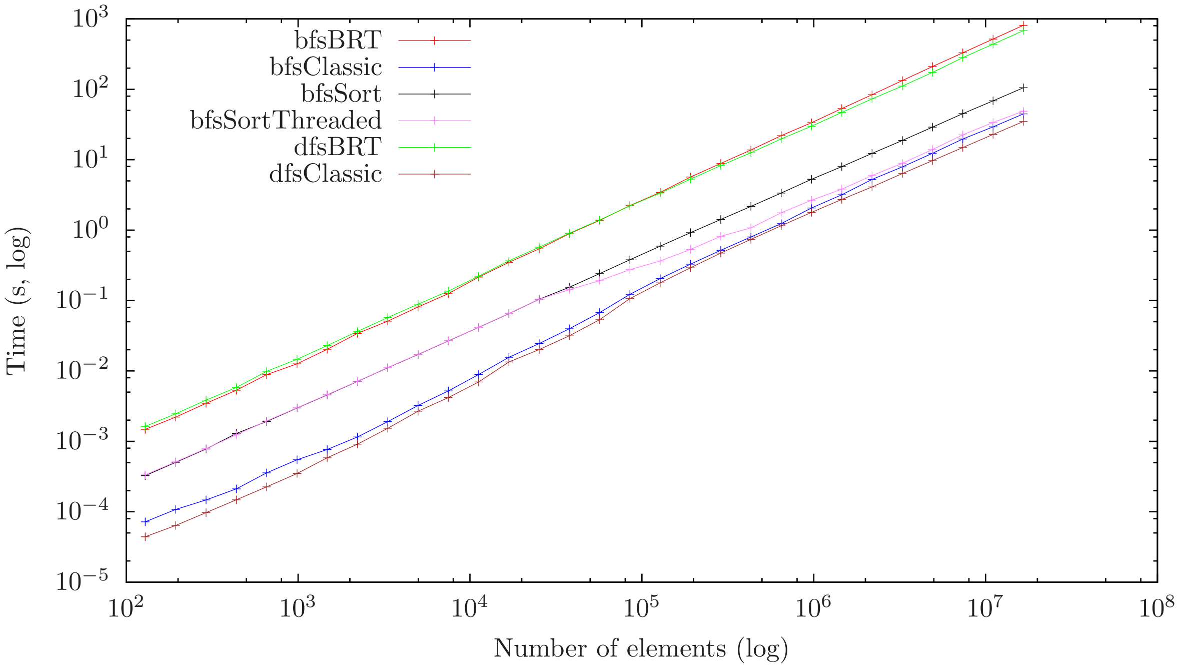

Figure A.9: Time of search on a triangulation graph (10 iterations)

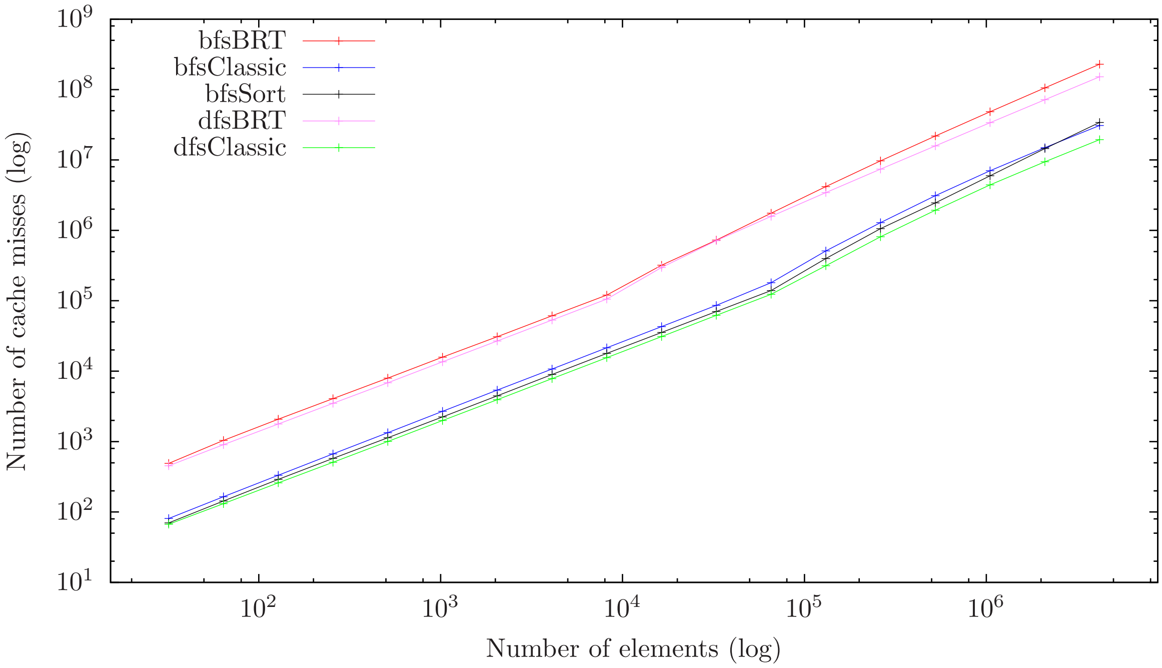

Figure A.10: Number of cache misses on a triangulation graph

is almost perfect with all the algorithms, so we can guess accessing the vertices is the problem.

A.4 Components

We show the run time on a triangulation and tree-like inputs for completeness in Figures A.17 and A.18. These, however, bring no surprise compared to the Figures 9.2 and 9.3 in Section 9.4. The peaks on classic DFS algorithm in case of the tree-like input is more likely an implementation problem or measurement inaccuracy than algorithm design issue. There's nothing surprising in the graphs of cache misses in Figures A.19, A.20 and A.21 either.

When we look at how the algorithms use a bigger cache (in Figures A.22, A.23, A.24 and A.25), we see that all the algorithms benefit from the size, but the BRT version probably the least of them. The divide and conquer algorithm has a jump at one place, which is strange, as it does not represent the vertices and it is unlikely all the edges would fit at once.

We also look at the utilization of data from a loaded cache line in Figures A.26, A.27, A.28 and A.28. We again come to the conclusion that the problem is accessing vertices (as the algorithms don't transfer more data with larger cache lines on the dense graph,

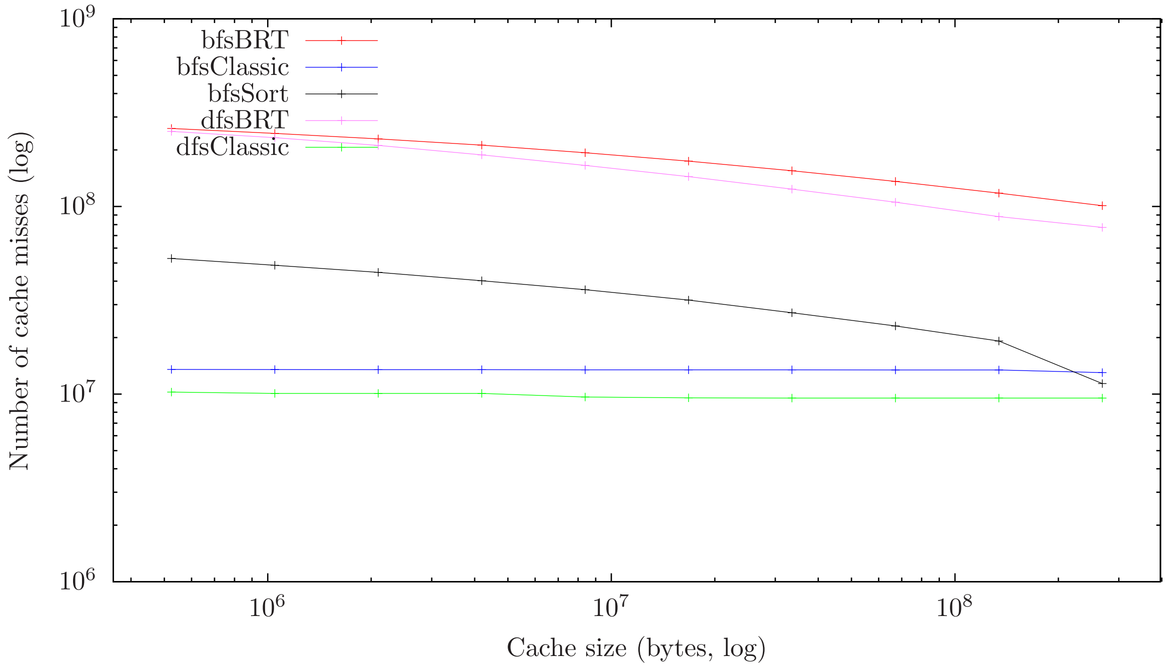

Figure A.11: Cache misses depending on the cache size on a dense graph

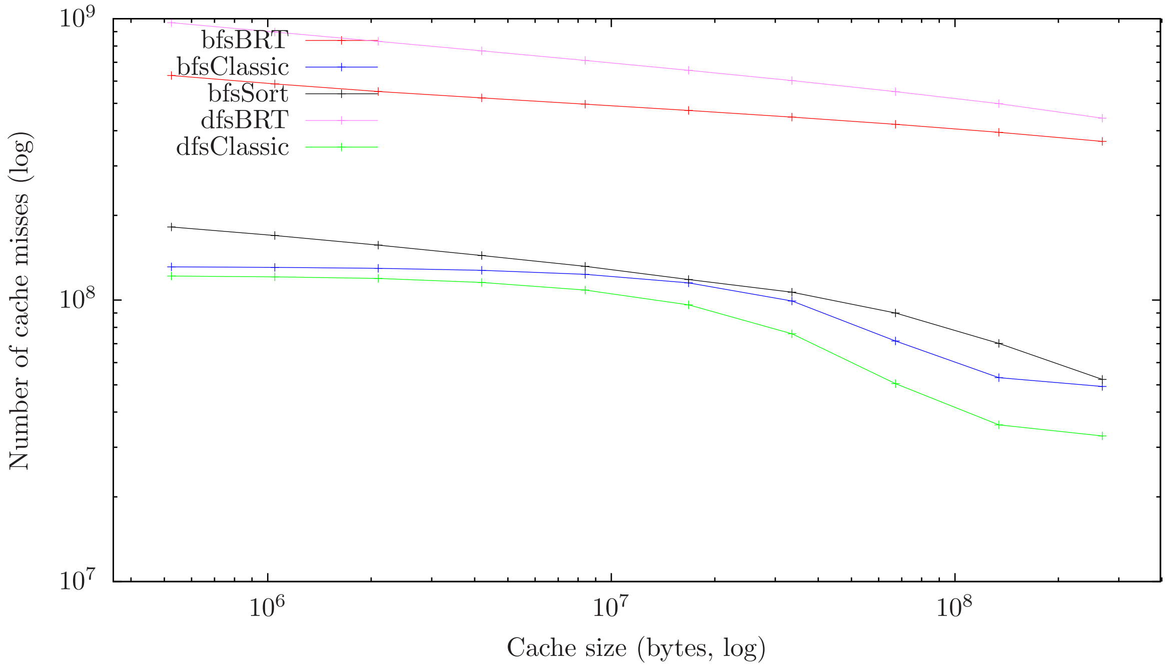

Figure A.12: Cache misses depending on the cache size on a sparse graph

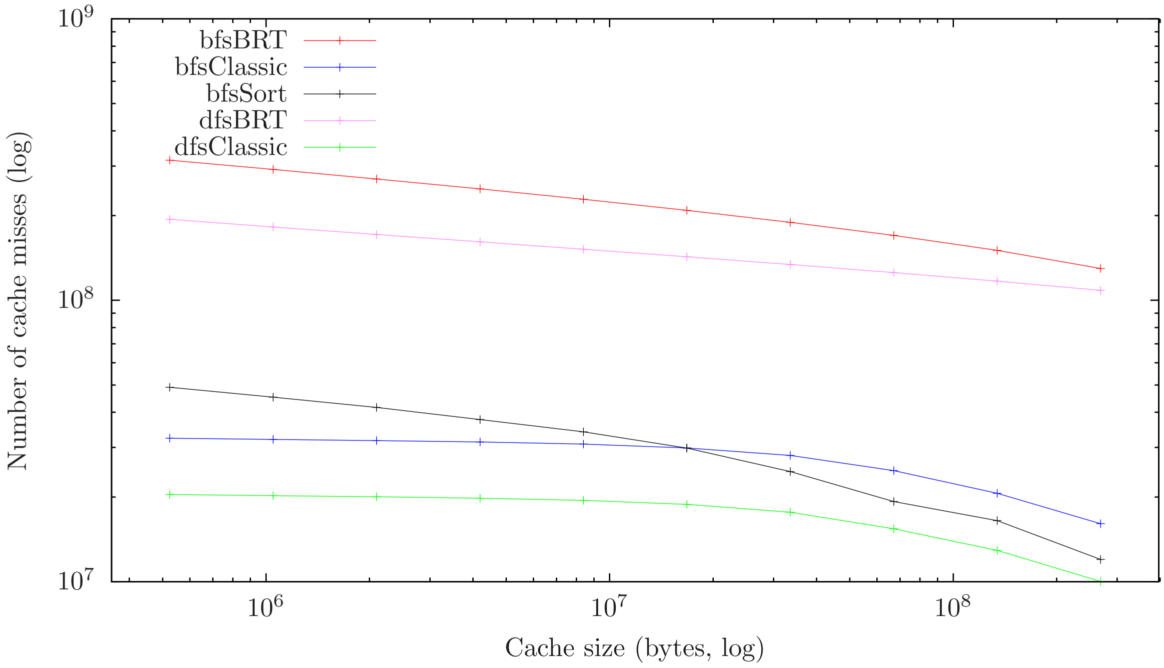

Figure A.13: Cache misses depending on the cache size on a triangulation graph

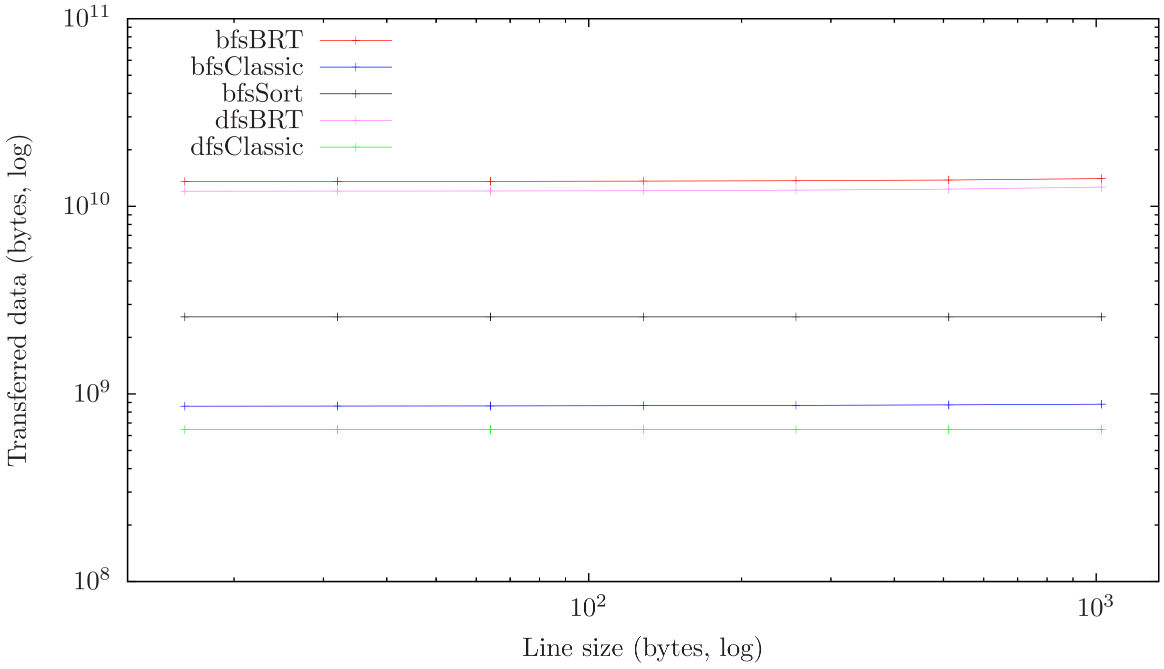

Figure A.14: Amount of data transferred on a dense graph

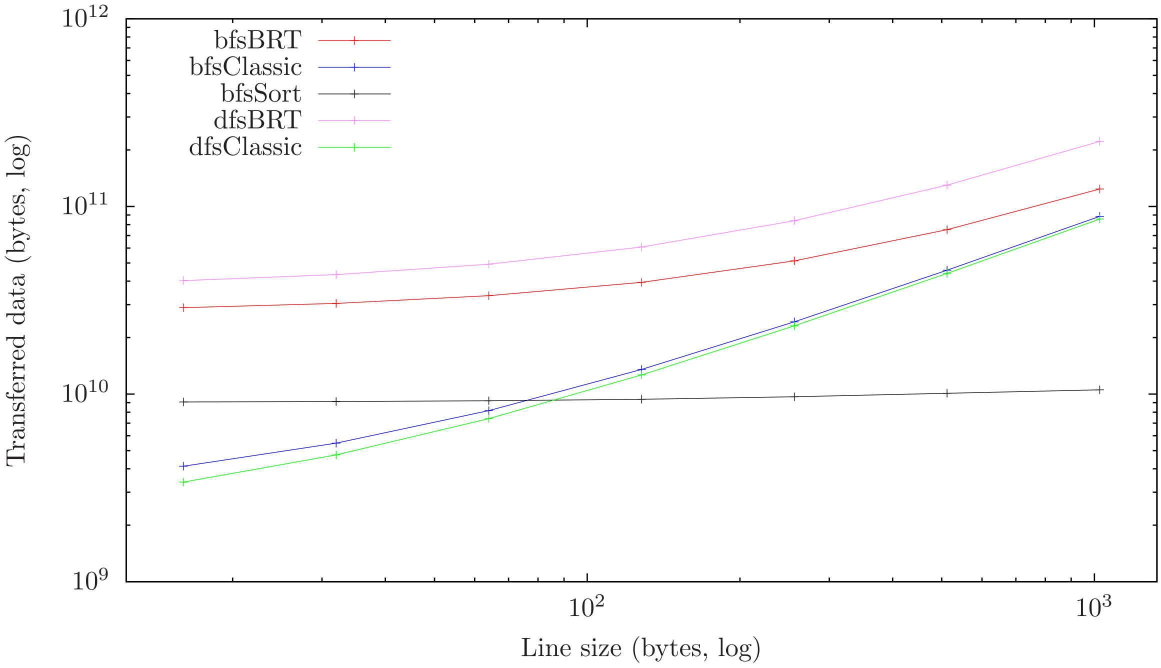

Figure A.15: Amount of data transferred on a sparse graph

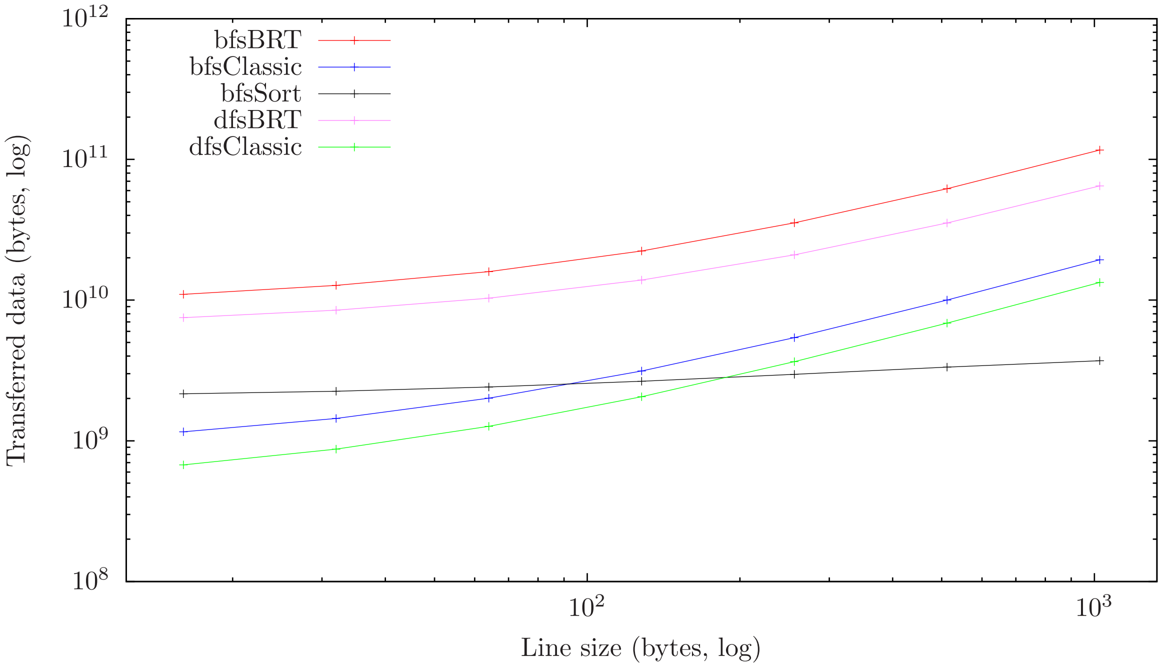

Figure A.16: Amount of data transferred on a triangulation graph

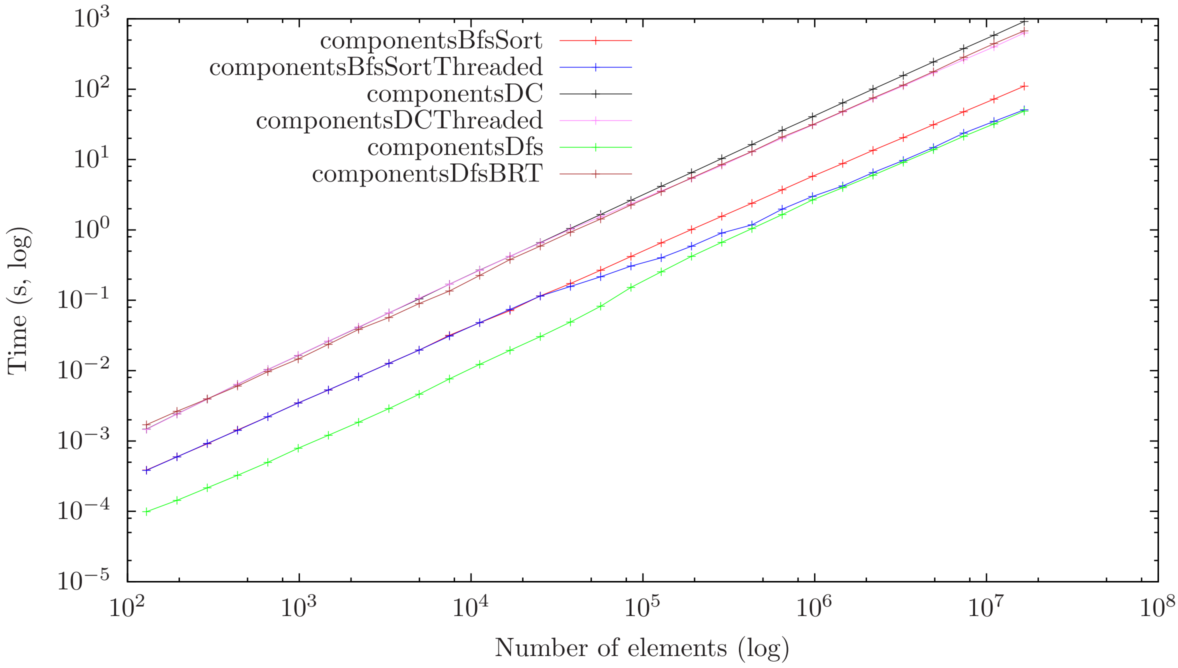

Figure A.17: Run time of components computation on a triangulation graph

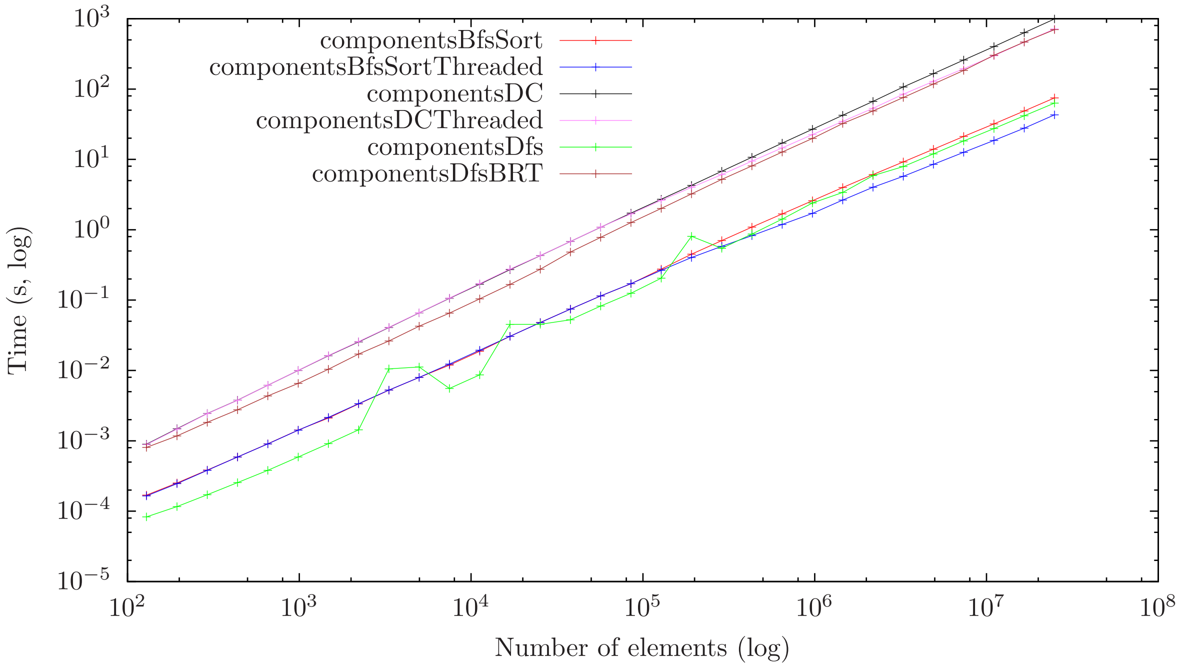

Figure A.18: Run time of components computation on a tree-like graph

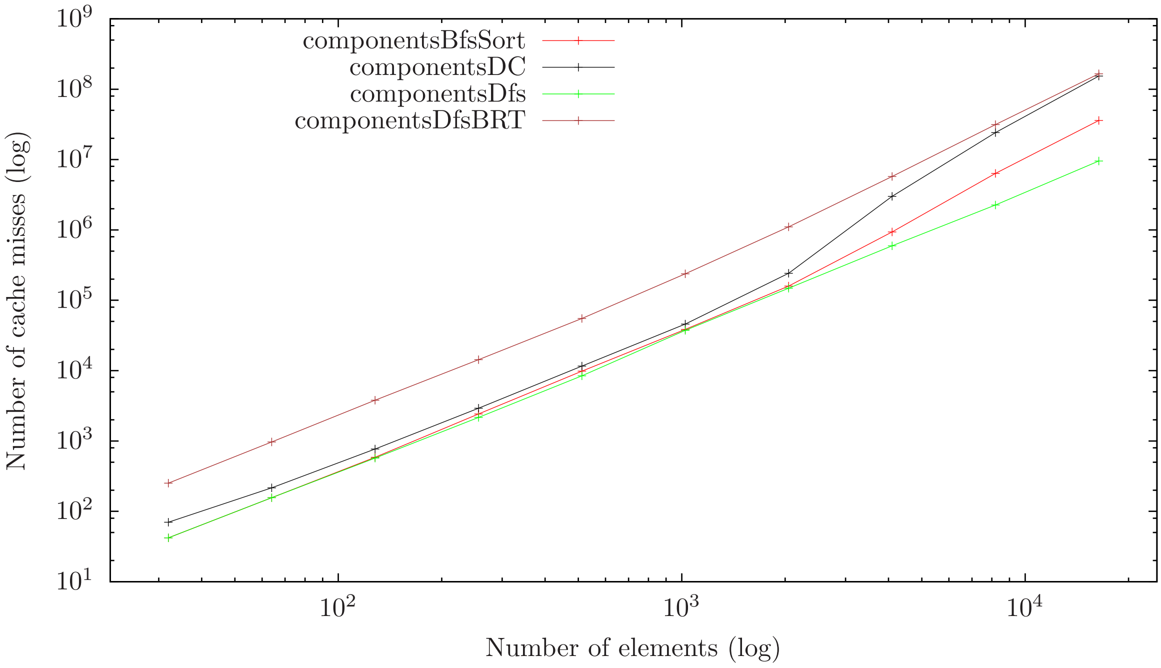

Figure A.19: Number of cache misses computing components of a dense graph

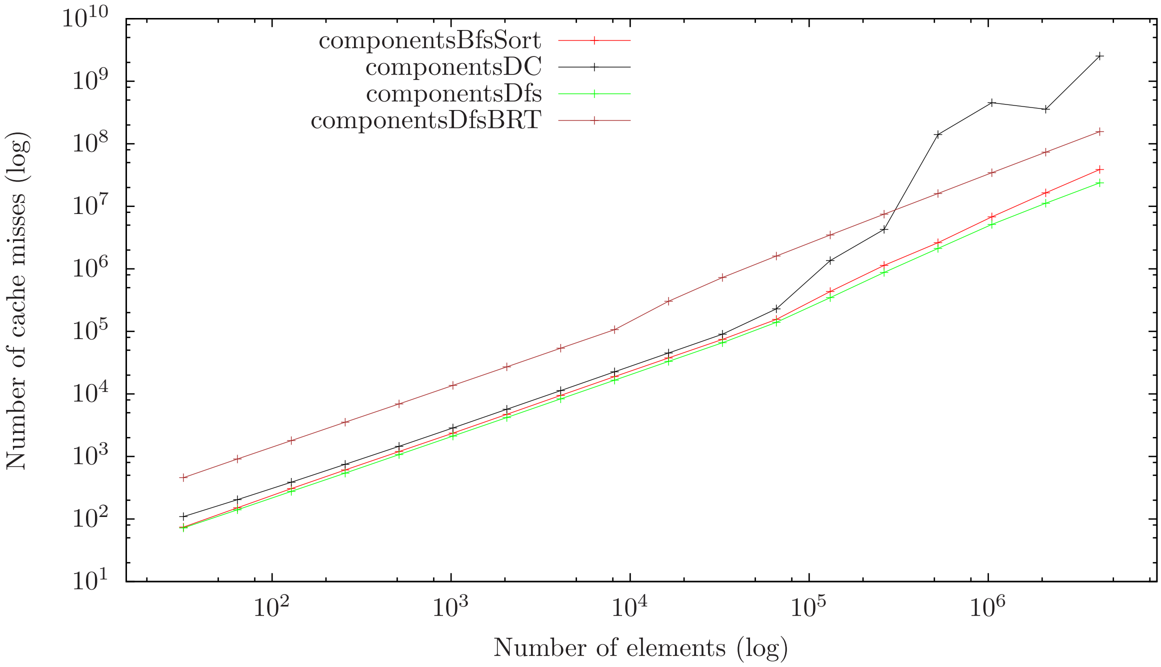

Figure A.20: Number of cache misses computing components of a triangulation graph

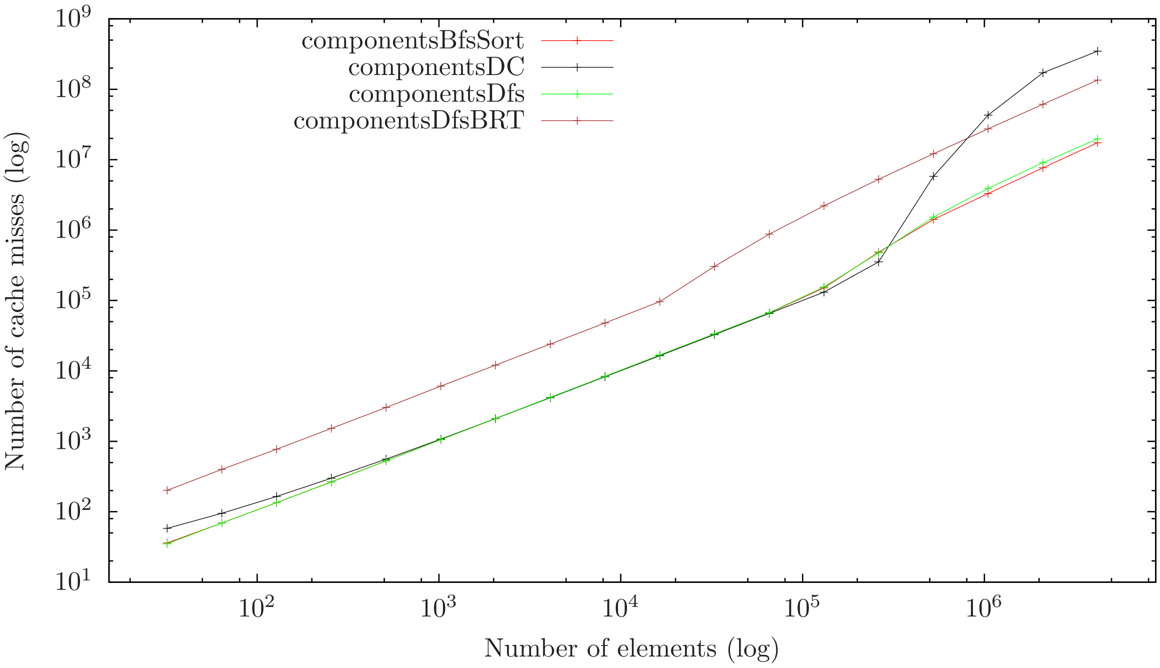

Figure A.21: Number of cache misses computing components of a tree-like graph

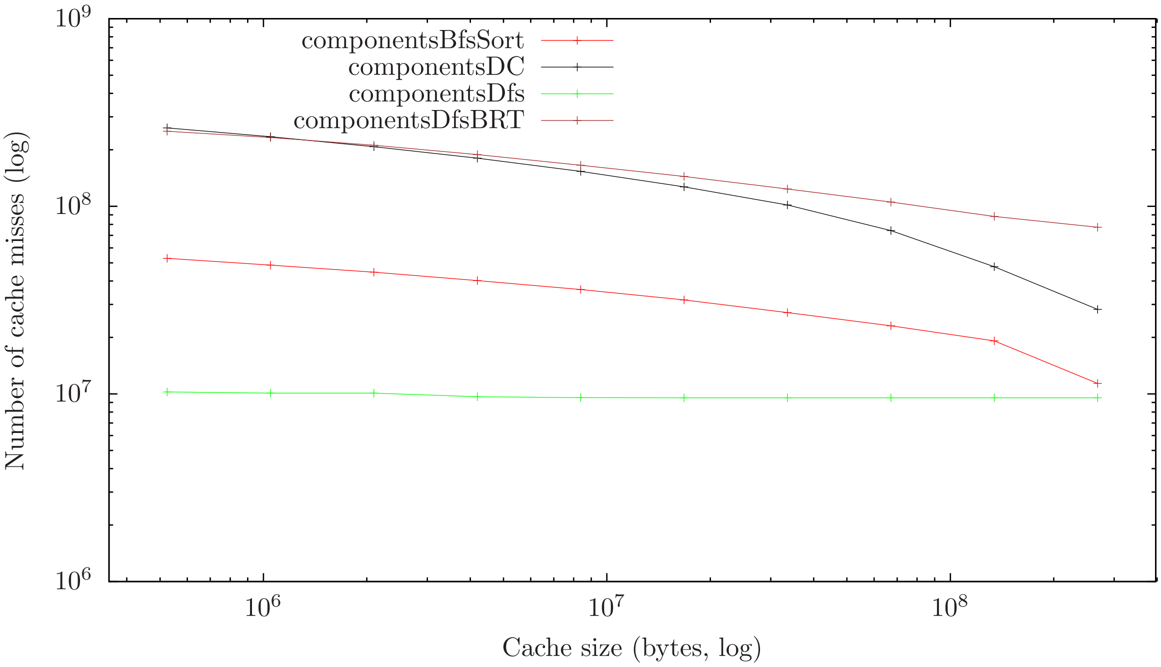

Figure A.22: Case size utilization by components of a dense graph

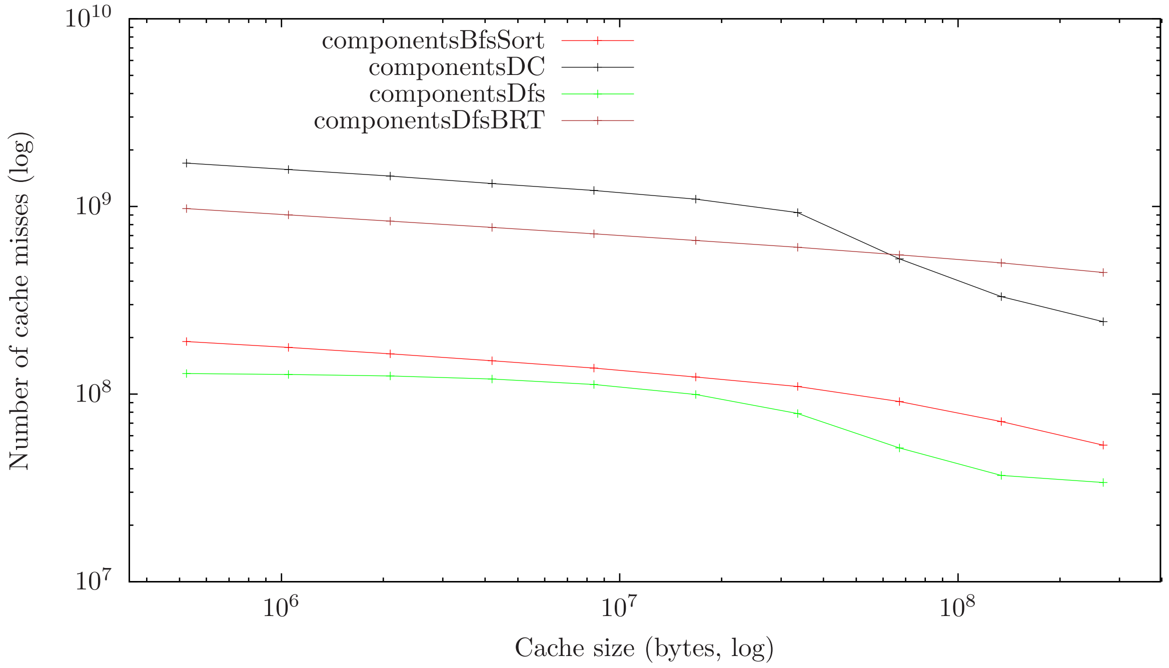

Figure A.23: Case size utilization by components of a sparse graph

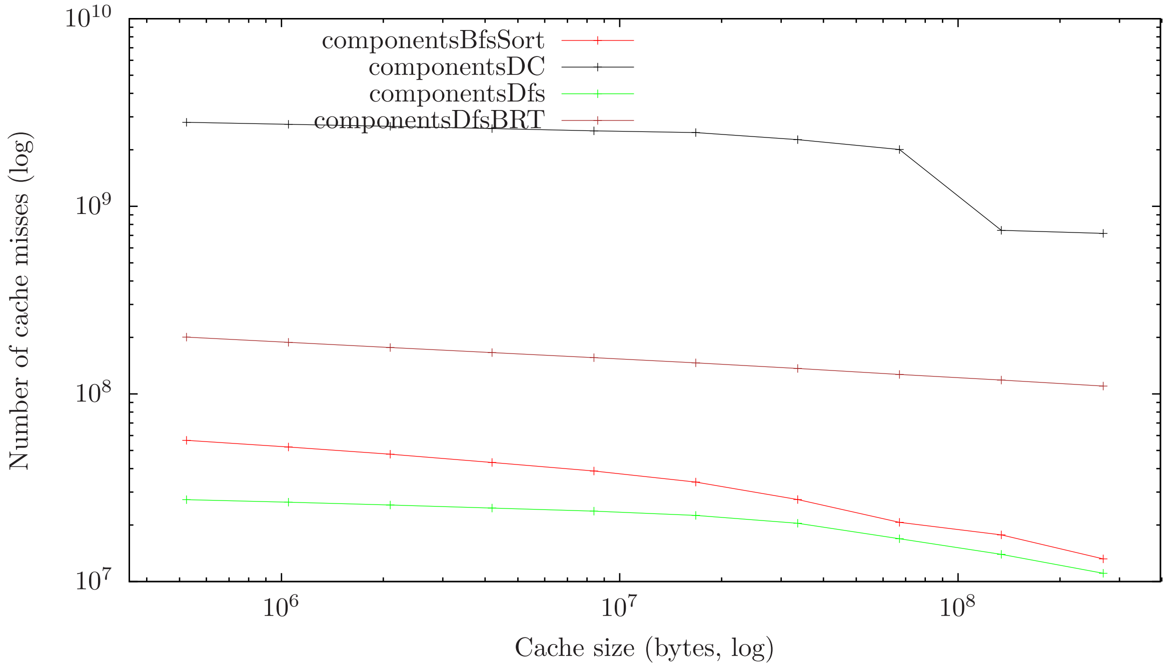

Figure A.24: Case size utilization by components of a triangulation graph

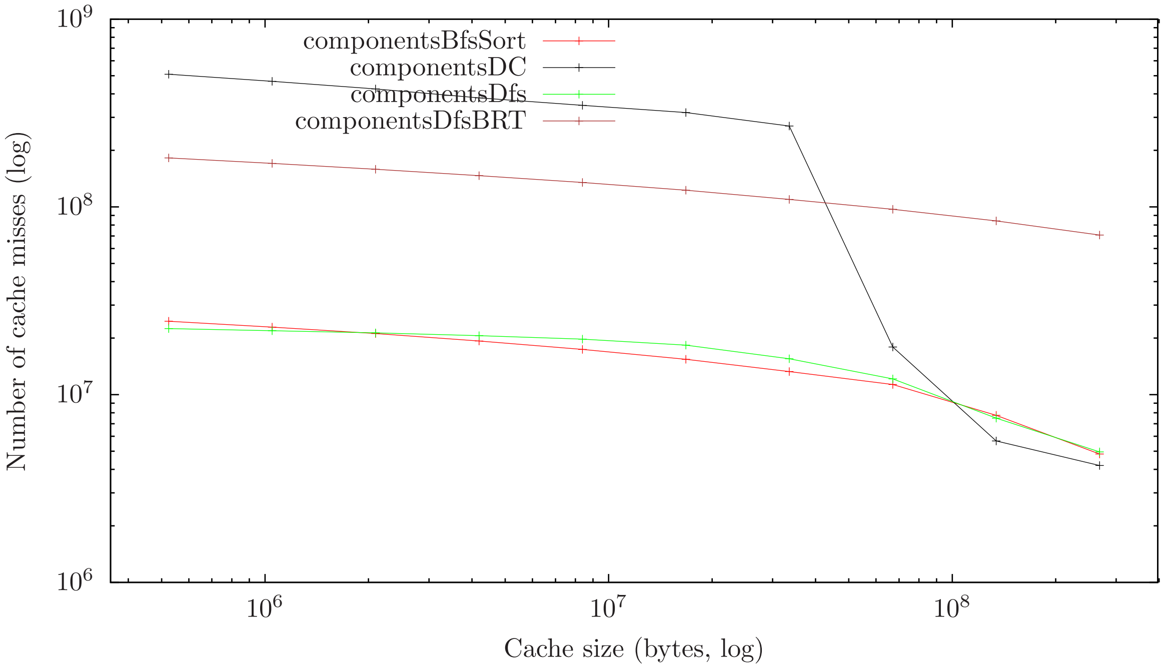

Figure A.25: Case size utilization by components of a tree-like graph

where the majority of accesses is to edges). The best utilization characteristic is clearly of the divide and conquer algorithm (possibly because it doesn't access the vertices at all), followed by the sorting based version.

A.5 Maximal matching

Like with the components algorithms, we provide the graphs in Figures A.30, A.31, A.32 and A.33 (run time and number of cache misses on sparse and tree-like graphs) for completeness, as they don't bring any surprise.

As we see on the graphs of how the algorithms benefit from bigger caches (in Figures A.34, A.35, A.36 and A.37), while the two cache-oblivious friendly algorithms benefit from the size slightly, the classical one nearly doesn't – which is expected, as it does a scan over edges.

If we look at the last group of graphs (Figures A.38, A.39, A.40 and A.41), we see that the only algorithm having problems with longer cache lines is the BRT based one, while the other algorithms use the data transfered to the internal memory almost without wasting.

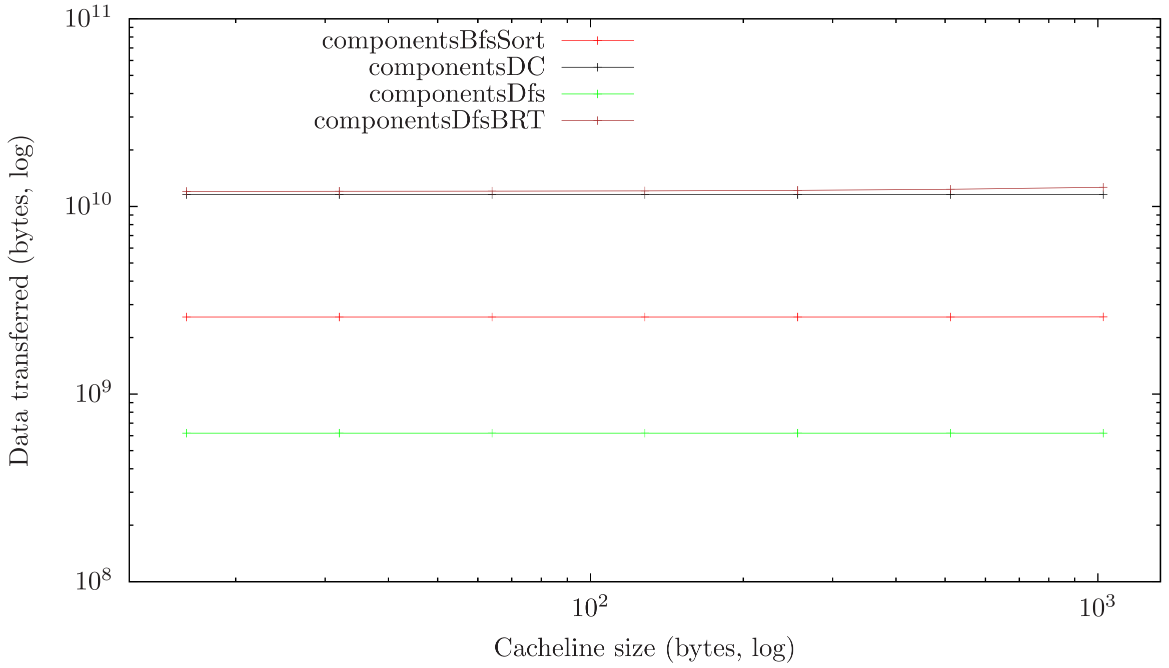

Figure A.26: Amount of data transfered for components of a dense graph

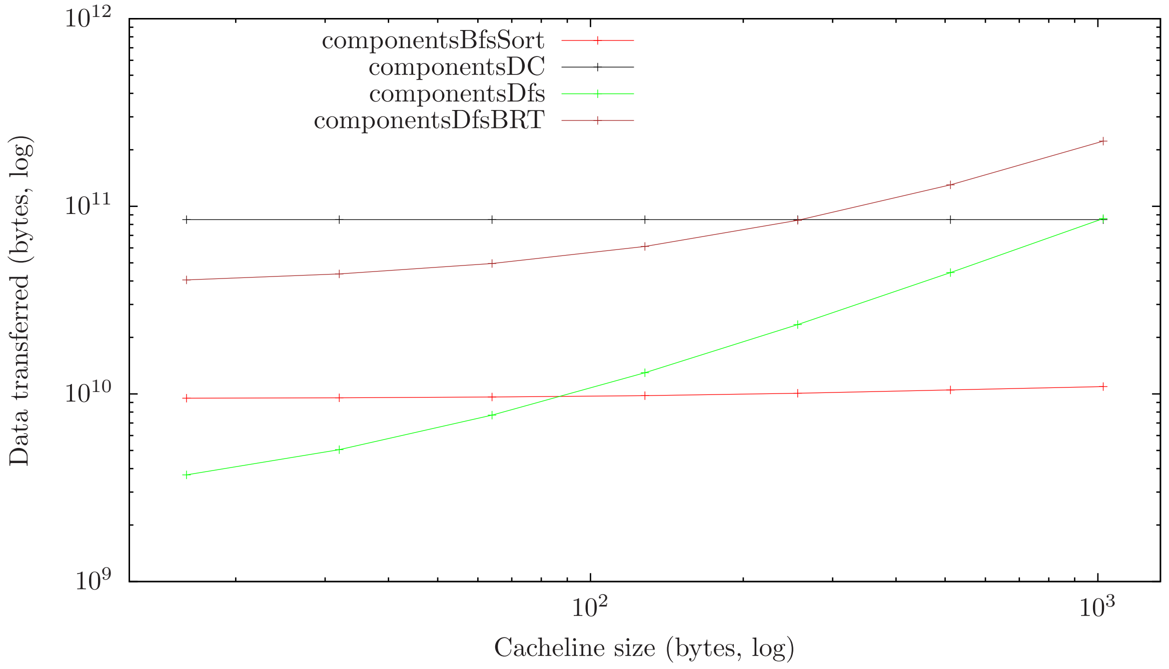

Figure A.27: Amount of data transfered for components of a sparse graph

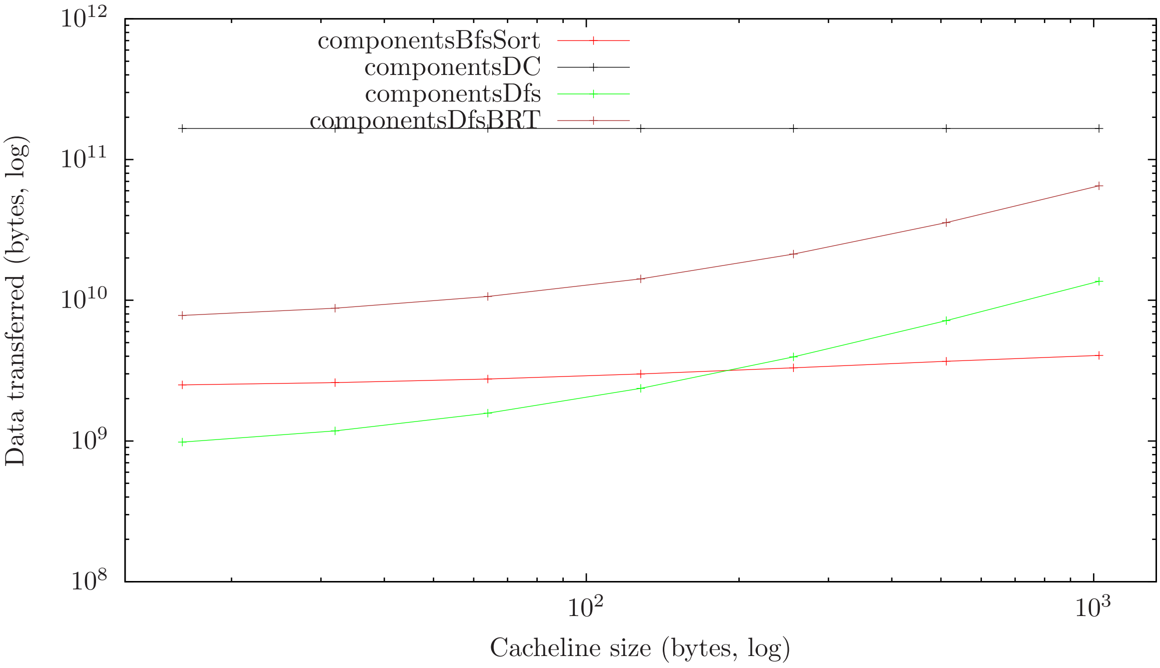

Figure A.28: Amount of data transfered for components of a triangulation graph

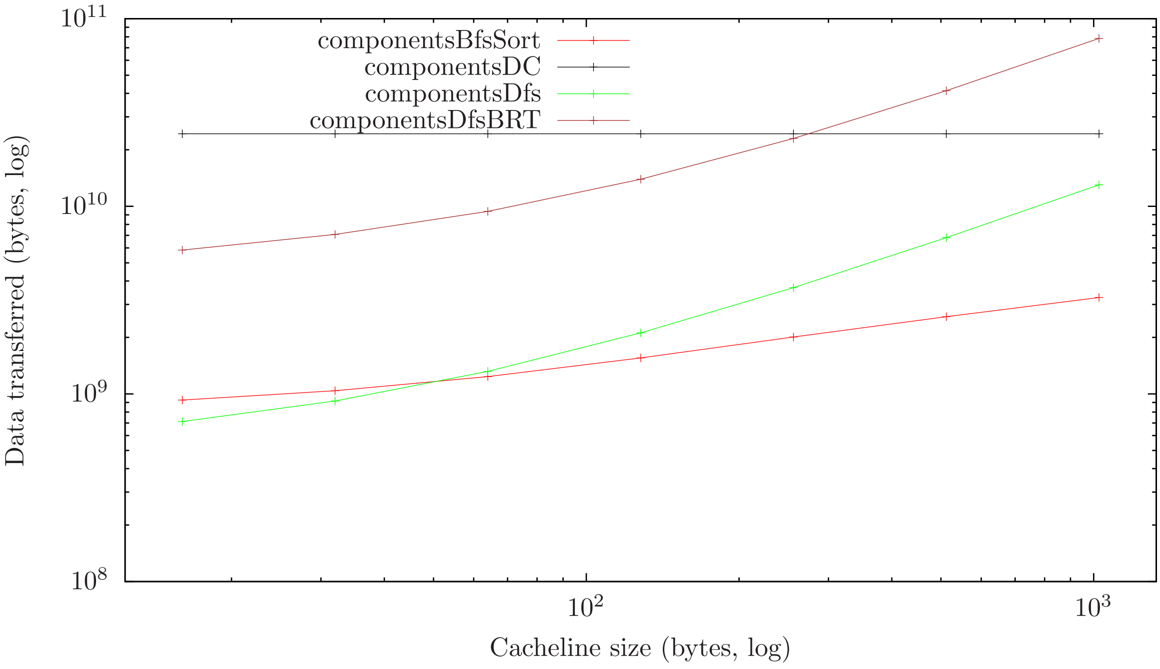

Figure A.29: Amount of data transfered for components of a tree-like graph

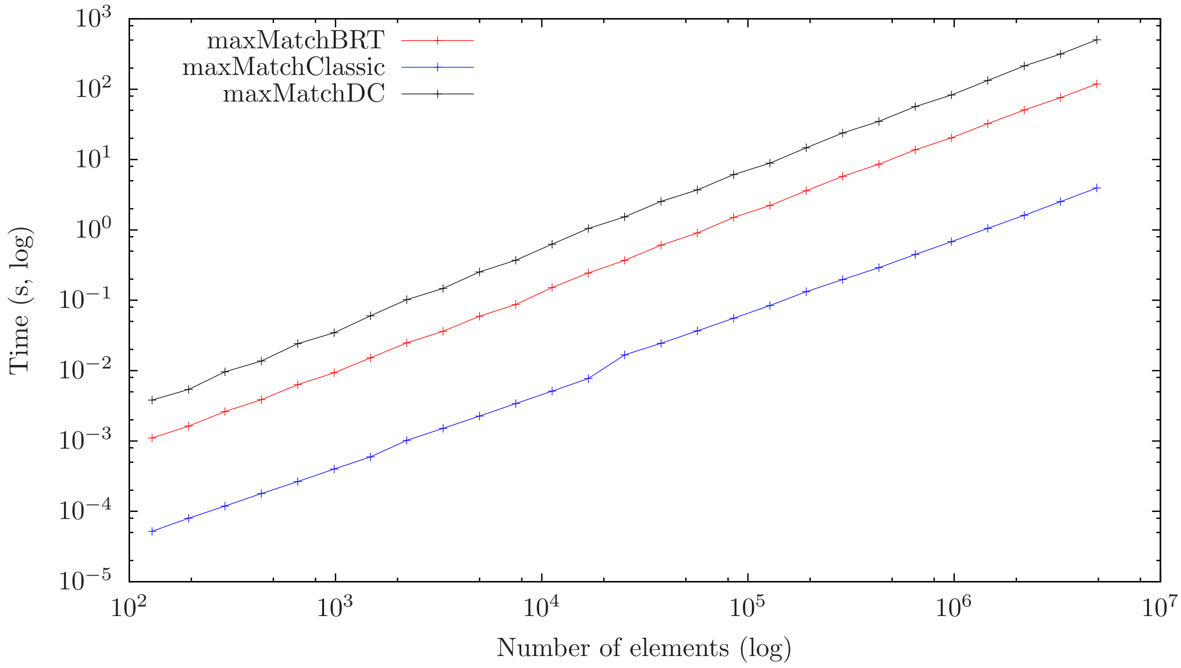

Figure A.30: Run time of maximal matching on a sparse graph

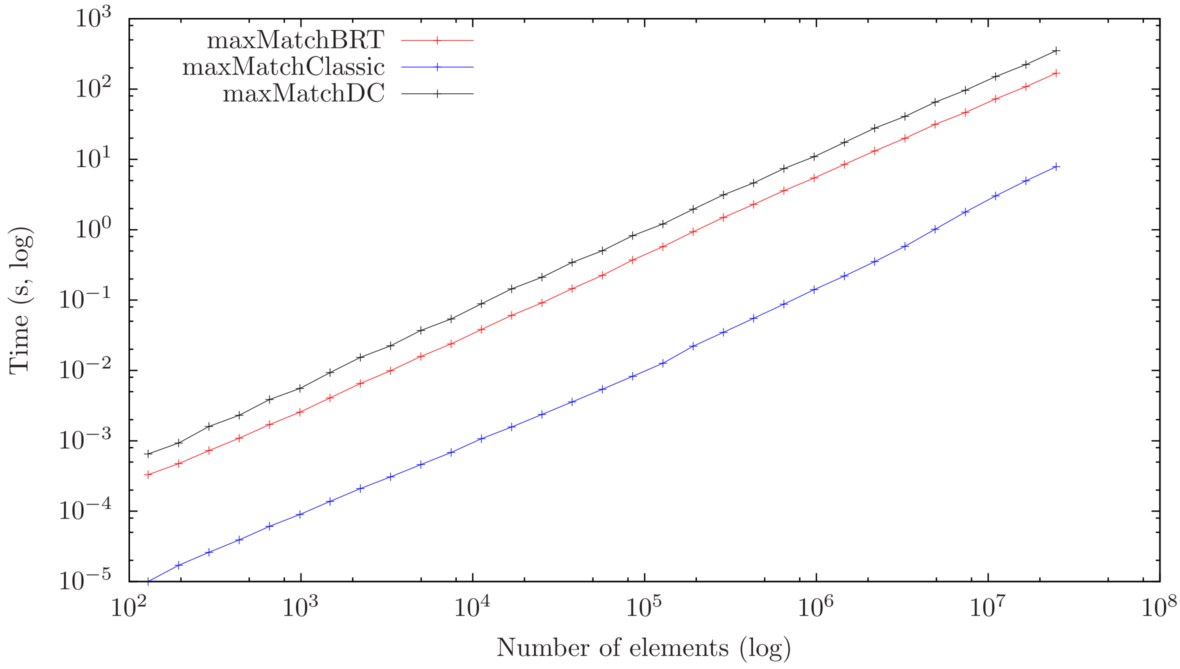

Figure A.31: Run time of maximal matching on a tree-like graph

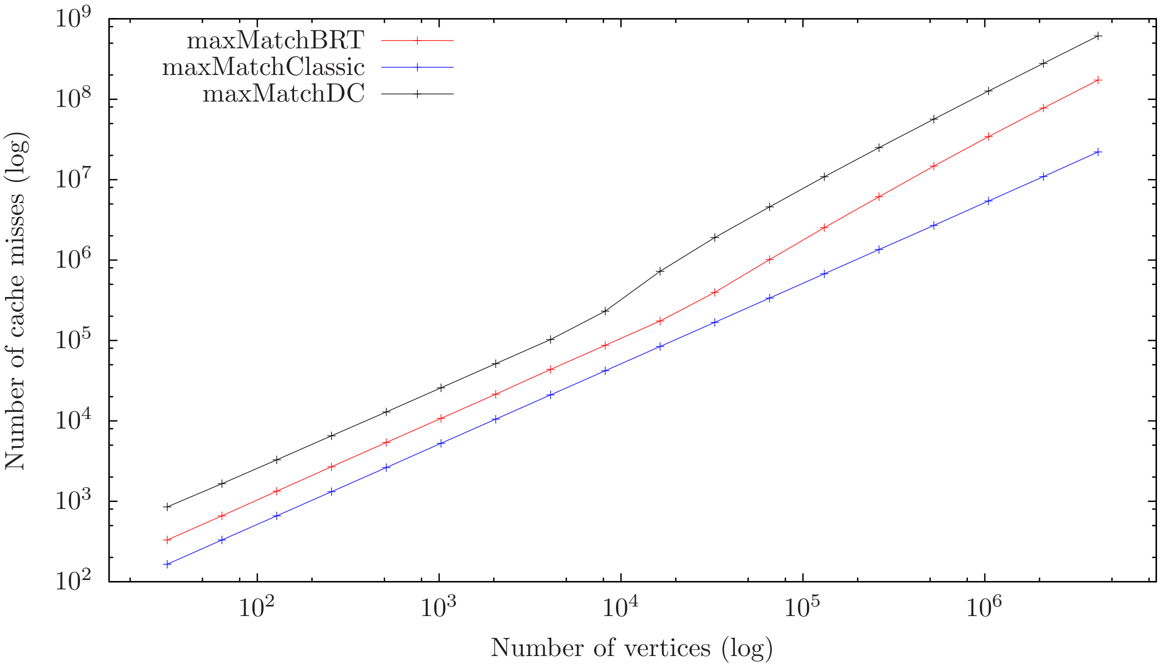

Figure A.32: Cache misses during maximal matching on a sparse graph

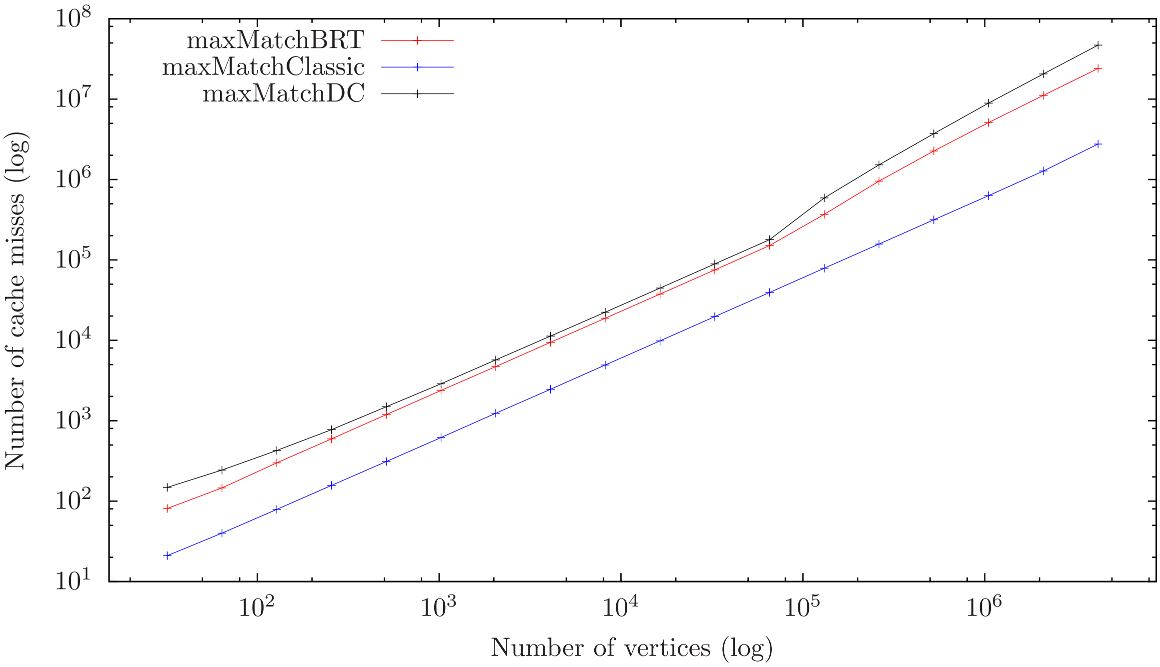

Figure A.33: Cache misses during maximal matching on a tree-like graph

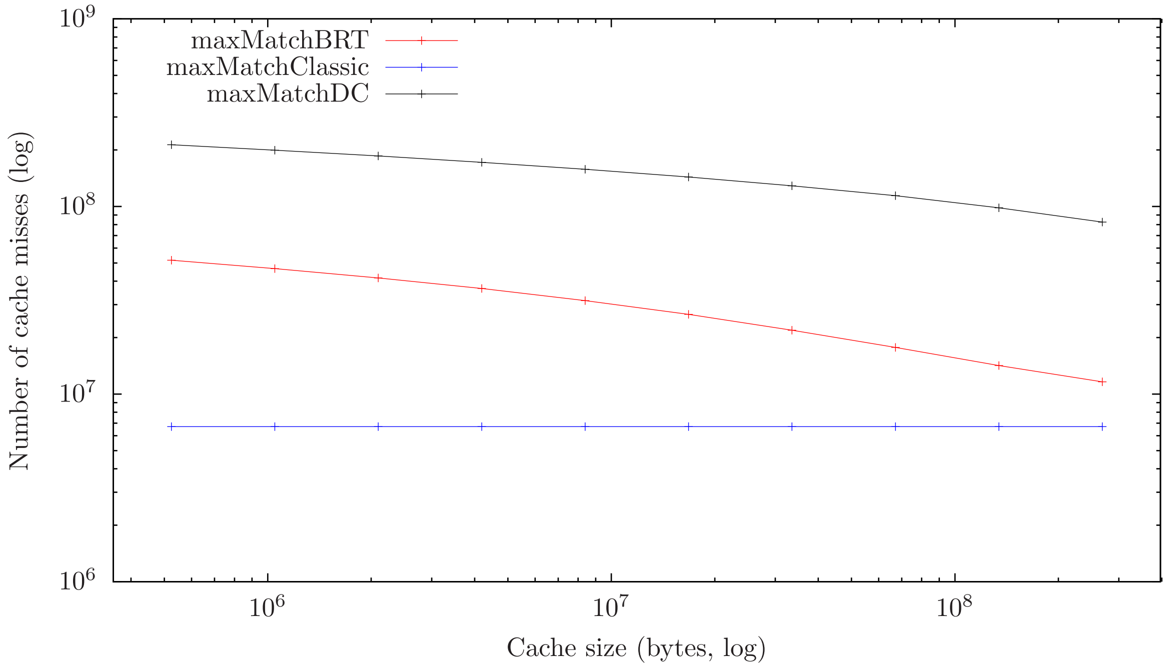

Figure A.34: Cache size utilization by maximal matching on a dense graph

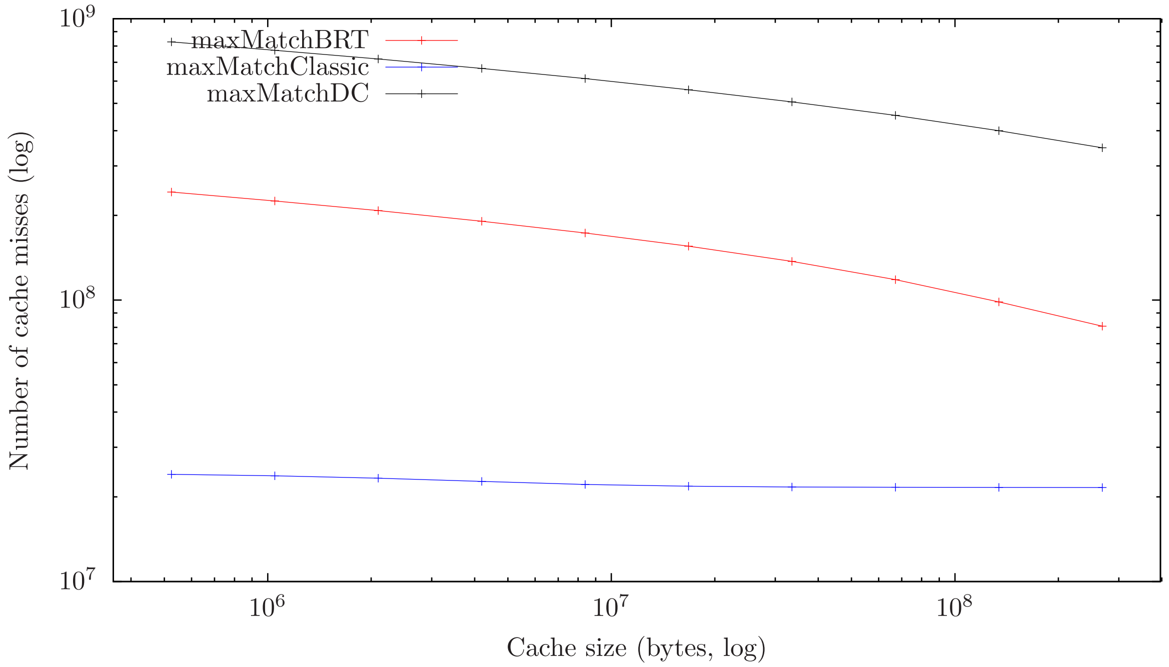

Figure A.35: Cache size utilization by maximal matching on a sparse graph

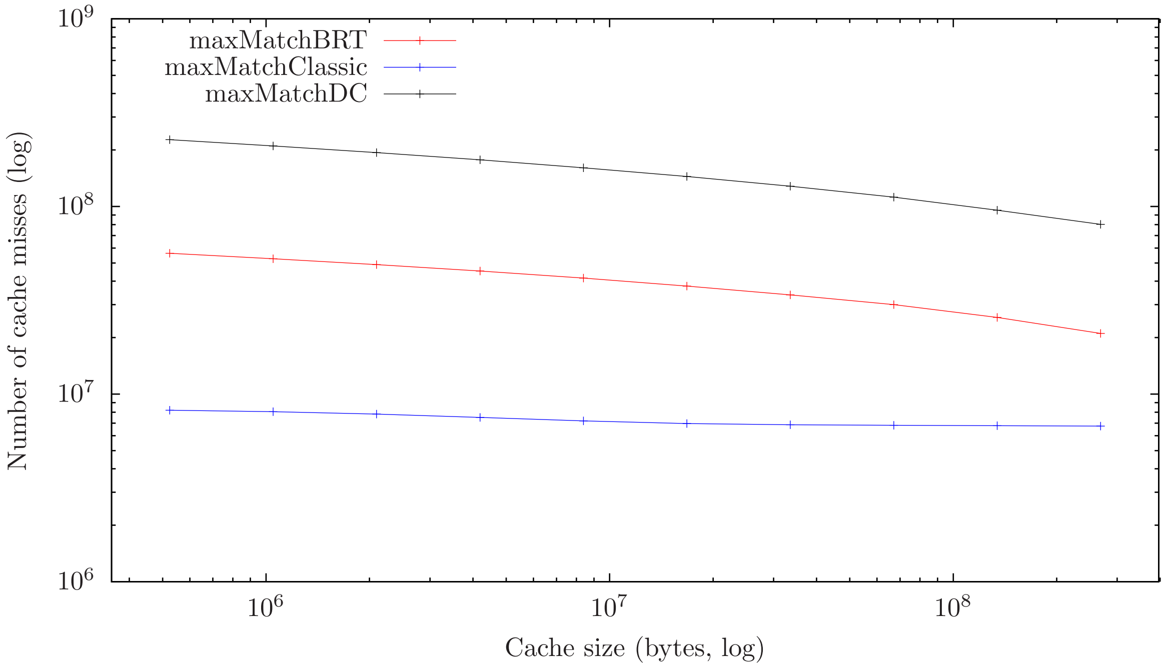

Figure A.36: Cache size utilization by maximal matching on a triangulation graph

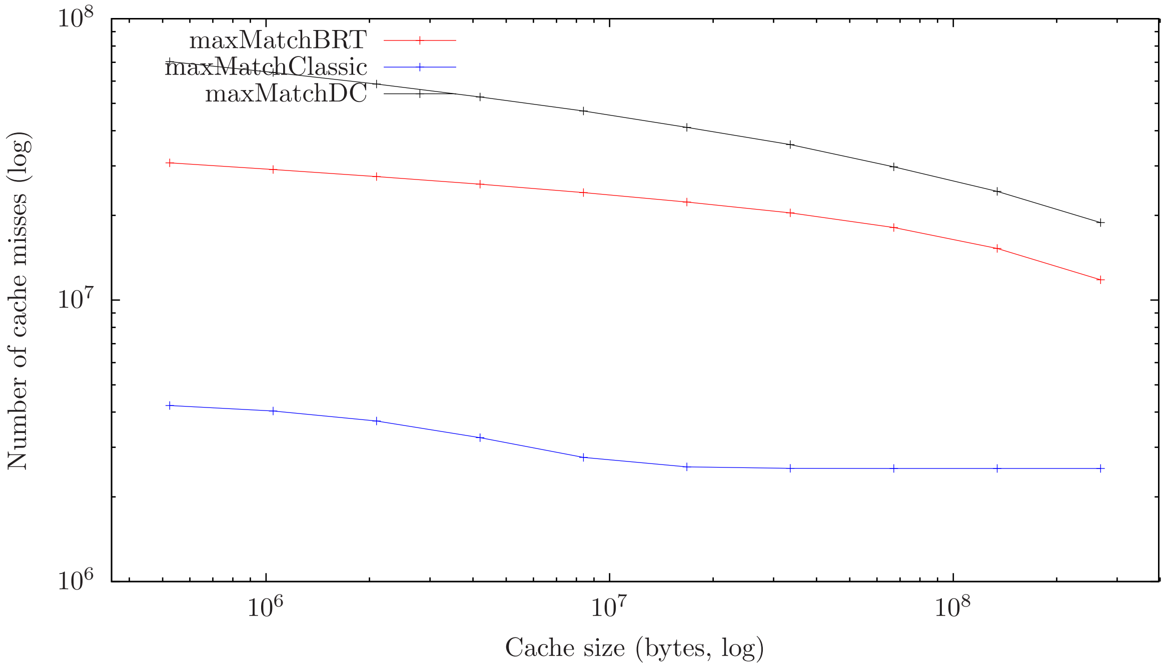

Figure A.37: Cache size utilization by maximal matching on a tree-like graph

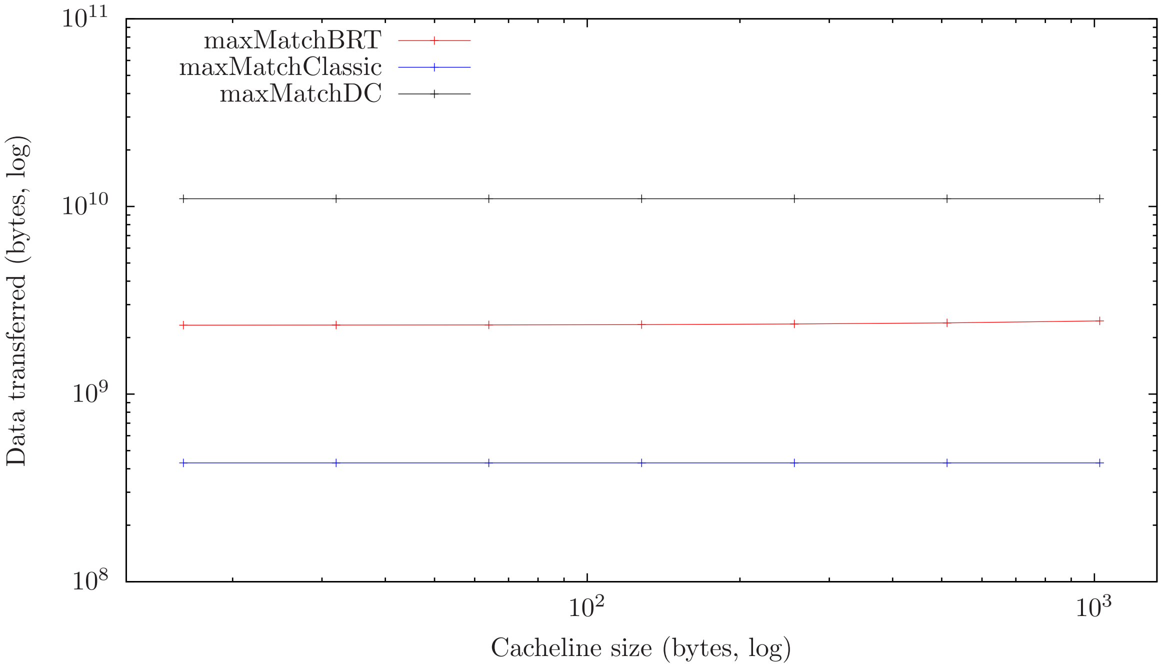

Figure A.38: Dependency on cache line size on a dense graph

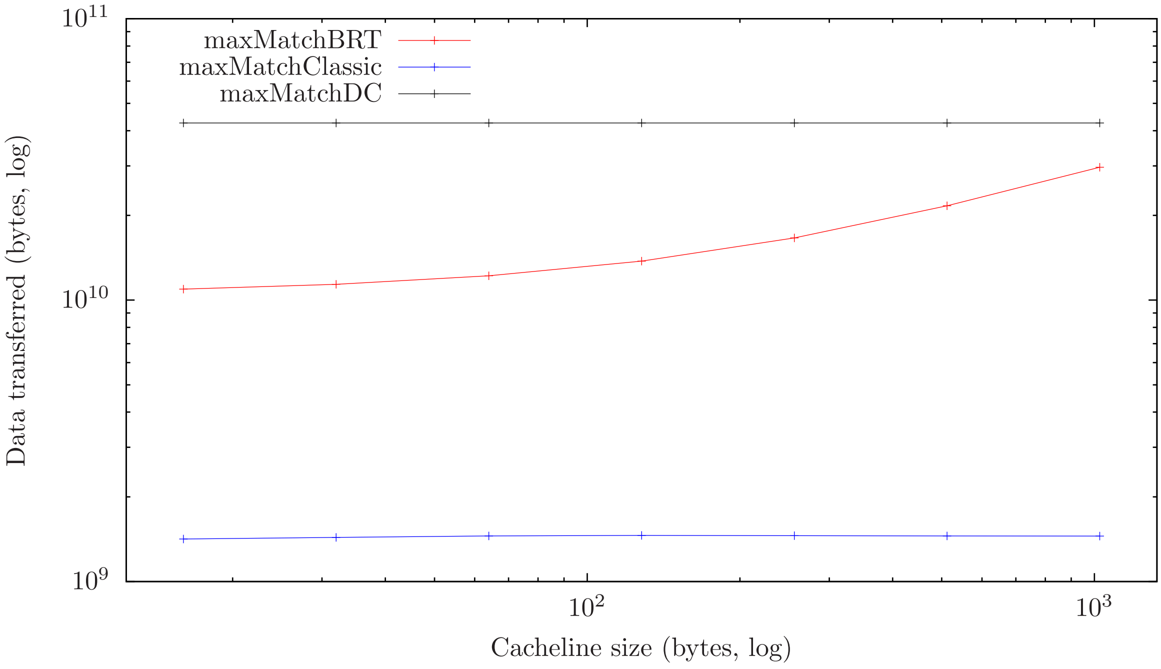

Figure A.39: Dependency on cache line size on a sparse graph

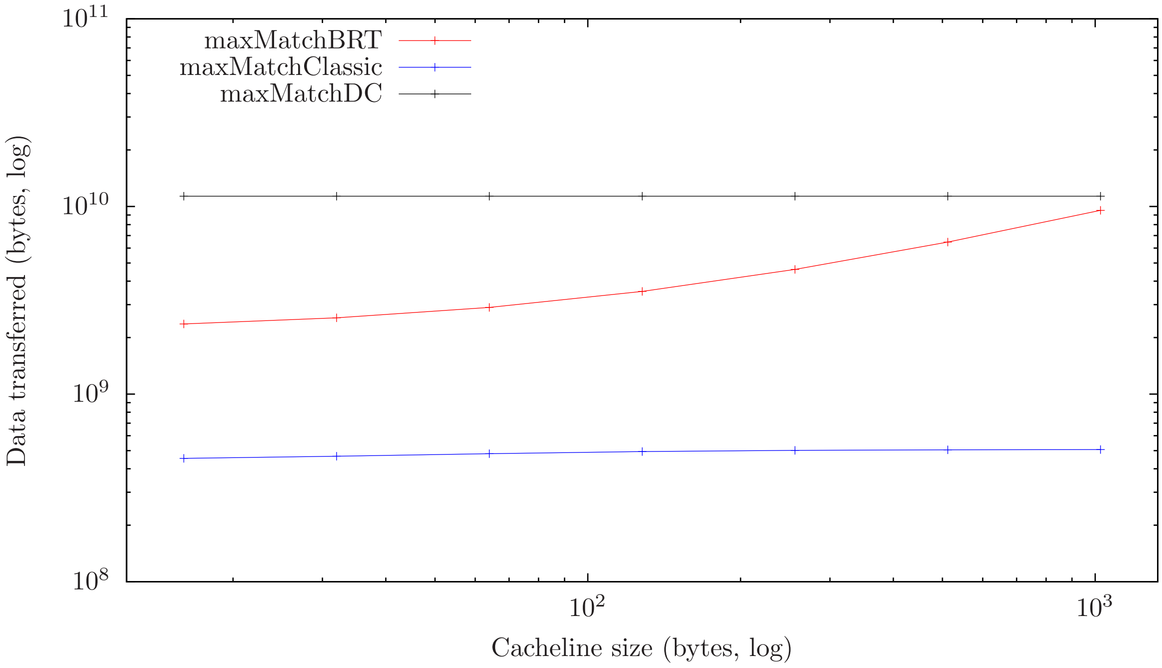

Figure A.40: Dependency on cache line size on a triangulation graph

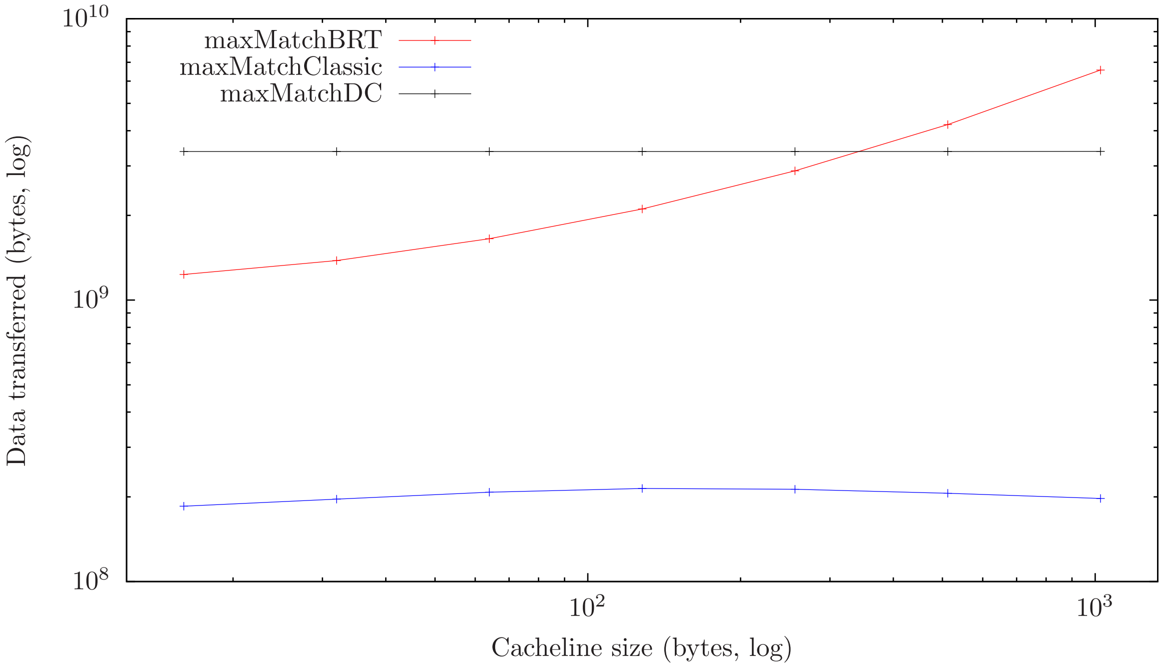

Figure A.41: Dependency on cache line size on a tree-like graph