Goethe-Universität Frankfurt

Master Thesis

Implementation of an External-Memory Diameter Approximation

Submitted to

Prof. Dr. Ulrich Meyer

Professorship for Algorithm Engineering

by

David Veith

February 09, 2012

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.
Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Frankfurt, den 09.02.2012

_____

(David Veith)

Acknowledgement

“Thank you!”:
Andreas Beckmann. He never got tired of debugging STXXL, helping me when my code was not efficient and with other questions.
Deepak Ajwani. From the first minute he always had a good advice that pushed my work into the right way. He has always been a good listener regarding my issues and problems.
And, of course, Ulrich Meyer. He gave me the chance to write this thesis. And I really enjoyed the time we worked together.

Also thanks to:
Andrei Negoescu and Volker Weichert.

Special thanks to:
Sarah Voß, David Weiß and all the others. They have been very good colleagues for several years. I have learned a lot from them.

Also a special thanks to my family, my friends and to my girlfriend Nadine Nienberg. You all supported me a lot.

A very special thank to Alexander Zech.

Contents

List of Tables

List of Figures

Abstract

Computing the diameter of a graph is a fundamental part of network analysis. Even if the data fits into main memory the best known algorithm needs $O(n^2)$ [3] with high probability to compute the exact diameter. In practice this is usually too costly. Therefore, heuristics have been developed to approximate the diameter much faster. The heuristic "double sweep lower bound" (dslb) has reasonably good results and needs only two Breadth-First Searches (BFS). Hence, dslb has a complexity of $O(n+m)$. If the data does not fit into main memory, an external-memory algorithm is needed. In this thesis the I/O model by Vitter and Shriver [4] is used. It is widely accepted and has produced suitable results in the past. The best known external-memory BFS implementation has an I/O-complexity of $\Omega(\frac{n}{\sqrt{B}} + sort(n))$ for sparse graphs [5]. But this is still very expensive compared to the I/O complexity of sorting with $O(N/B \cdot \log_{M/B}(N/B))$. While there is no improvement for the external-memory computation of BFS yet, Meyer published a different approach called "Parallel clustering growing approach" (PAR_APPROX) that is a trade-off between the I/O complexity and the approximation guarantee [6].

In this thesis different existing approaches will be evaluated. Also, PAR_APPROX will be implemented and analyzed if it is viable in practice. One main result will be that it is difficult to choose the parameter in a way that PAR_APPROX is reasonably fast for every graph class without using the semi external-memory Single Source Shortest Path (SSSP) implementation by [1]. However, the gain is small compared to external-memory BFS using this approach. Therefore, the approach PAR_APPROX_R will be developed. Furthermore, a lower bound for the expected error of PAR_APPROX_R will be proved on a carefully chosen difficult input class. With PAR_APPROX_R the desired gain will be reached.

Initial results on PAR_APPROX have been published in [7].

Chapter 1

Introduction

A graph can be used for modeling many different things from the real world, e. g. for social network research [8]. For network analysis the computation of the diameter of the network is an important part [8]. The diameter is related to the time that is needed to communicate between two arbitrary points in a network.
But not only in the networks analysis the diameter is important. Graphs are also used for modeling biological structures like viruses, RNA and DNA, chemical processes or in computer graphics for creating a scene and many other examples. The complexity of many approaches in these fields also depends on the diameter. And the diameter is also important in many other applications in computer science. The execution time of many algorithms depends on the diameter. The execution time of parallel approaches as the Parallel Shortest Path are bounded by the diameter [9, 10, p. 92].
However, the computation of the diameter is very expensive. The best known algorithm needs $O(n^2)$ operations [3], where n denotes the number of vertices in the network. As mentioned before, there already exists an algorithm, which can compute the diameter in $O(n^2)$. But this is very complex compared to other algorithms used in practice for analyzing a graph. Especially when there is a time limit as in real-time applications, this is too costly. Therefore, some heuristics were developed which will be presented in Chapter 3.

There is another aspect which makes computation difficult. Many networks are very complex and therefore large, e. g. road networks. That leads to a size of data, which does not fit into the main memory (here: DDR SDRAM) of many standard computers. The main memory is very expensive and in practice it is very difficult to upgrade a machine to an arbitrary size.
Another difficulty is that the size of graph files is growing. But the latency of I/O will not get better. The opposite is happening. The speed of the CPU is increasing much faster than the speed of the main memory – especially the ratio between the main memory and a single core of the CPU (see [11]). This is also happening to the other hardware components as to the latency between the main memory and the external-memory devices [11, 12, 13].
Data, which does not fit into the main memory has to be stored on an external-memory device. Usu-

ally this external-memory device is a hard disk (short: HDD for hard disk drive) or sometimes a solid state drive (SSD). Such devices transfer data in blocks of size B. That means, that each data transfer contains B data elements (compare to Section 2.5 or [4]). The random access to an element on the hard disk is time-consuming compared to a random access to the main memory. While a random access to an element on a hard disk takes around 8.4ms in worst case[1] or at best around 2.7ms (read) / 3.1ms (write) in worst case on very expensive hardware for enterprise[2], reading a whole block or only one element from a disk does not make a big difference. This is because of the way such a device is constructed. The read/write head has to be in the right position. The disk has to rotate until the right block is under the head. Both usually take milliseconds, while the reading of the data in the block takes a few nanoseconds. A random access to the main memory takes only around 40 to 100 nanoseconds[3]. Thus, a random access to a disk is around 100,000 times slower than a random access to the main memory. Otherwise, with one access to the disk B elements can be fetched.
Hence, it is better to use the whole information from the block for current computation than using only one specific element from the read block. In other words: The main target is to minimize the number of random accesses to the hard disk since this takes a lot of time. If B is reasonably large, and all B elements are used for following computation, this is not too bad. But there is still a slowdown, that cannot be avoided.
However, the existing different implementations of Breadth-First Search using external-memory devices (e. g. [17, 5]) have an I/O complexity of $\Omega(n/\sqrt{B})$ for sparse graphs. Reading all data in a sequence would be much faster. This kind of reading is called scanning (see Section 2.6). There is a gap of $O(\sqrt{B})$ between external-memory BFS and scanning. Until today there is no published idea how to fill this gap. One computation of an external-memory Breadth-First Search usually takes several hours or longer, depending on the size of the data[4], if the data does not fit into main memory. For some heuristics one thousand Breadth-First Searches and more are needed (compare Section 3.3). That would lead to a computation time of months or years. Because that is not feasible, other approaches are needed. A trade-off has to be found between the I/O complexity and quality of the computation.

1.1 Purpose and aim of this thesis

This thesis is an experimental study about efficient heuristics for diameter approximation. One aim is to find mechanisms, which guarantee a fast heuristic – with reasonable quality for the resulting diameter.
Another aim is to implement an approach which needs as few random accesses to the hard disk as possible. This external-memory approach should be more efficient than the existing implementations

[1] E. g. disks from the "WD VelociRaptor®" series, produced by Western Digital [14]
[2] E. g. "Savvio ®15K.3 SAS" series from Seagate [15]
[3] For example a memory module from Kingston: "2GB 1066MHz DDR3 Non-ECC CL7 DIMM Single Rank x8" with a latency of approximately 60ns [16]
[4] In this thesis data sets with a size between about 8 and 40 Gigabyte will be considered.

for Breadth-First Search. One trade-off will be analyzed in greater detail in Chapter 4. This trade-off is based on the ideas in: “On Trade-Offs in External-Memory Diameter-Approximation” [6].
Also an improvement of the ideas in [6] will be developed in Section 4.2. In [6] it was mentioned that a recursive approach can also reduce the I/O complexity a lot. The problem is, that after the first shrinking of the graph, the edges are not unweighted any more. These weights make it difficult to find a reasonable bound for the expected error of the recursive approach. That will be discussed in Section 4.5.
In the end of this thesis a reasonable trade-off between I/O complexity and quality of the resulting diameter should be found.

1.2 Structure of this thesis

In the second chapter background information will be presented which is needed to understand the following chapters. A commonly used model for computation with accesses to disk will be introduced. Also, basic algorithms, which are related to the basic ideas, will be introduced.
In the third chapter related work will be introduced. This will be heuristics for a lower and an upper bound for the diameter based on Breadth-First Search and others. Also, an existing approach for using hard disks, based on Minimum Spanning Tree (see Section 2.10) will be introduced.
In chapter four the concept behind the new approach will be introduced. Also for an implementation with low I/O complexity the different possibilities will be discussed and rated. At the end of this chapter a recursive idea of this approach will be developed. Also the quality of this new approach will be discussed in more detail.
In chapter five the experiments and their results will be presented. Many results were predicted by theory, others were explored during the experiments. The explored results will be discussed in more detail.
In the sixth chapter the conclusion will be presented. A perspective of the results will be discussed and also possible work for the future.

Chapter 2

Basics

2.1 Graphs

A graph $G = (V, E)$ is a pair of sets [18, 19, p.529]. The set V contains the vertices v and the set E contains the edges e. $E \subseteq \{\{u, v\} | u, v \in V\}$, or in other notation: $E \subseteq V \times V$.
The notation n denotes the cardinality of the set $|V|$ and m the cardinality of the set $|E|$. The cardinality m is less than or equal[1] to $O(n^2)$ unless duplicates are allowed. A graph without any duplicate edges is called a simple graph. In this thesis the focus will be on simple undirected sparse graphs. If a considered graph has different properties, it will be mentioned.
In graphs with undirected edges an edge $\{u, v\}$ can be used in both directions: from u to v and vice versa. An edge with notation (u, v) instead of $\{u, v\}$ shows that the edge is directed, what means that v can be reached by u but not vice versa. Only, if also $(v, u) \in E$ then the connection between u and v is bidirectional.

2.2 Diameter of a graph

To define the diameter, the concept of a path in a graph G has to be defined first. "A walk in a graph G is a finite sequence of vertices $v_0, v_1, ..., v_n$ and edges $e_1, e_2, ..., e_n$ of G. The endpoints of an edge e_i are v_{i-1} and v_i for each i." (adapted from [20]). A path is a sequence of vertices $v \in V$ like a walk but each vertex in this sequence appears only once.
The distance between a pair of vertices $v_i, v_j \in V$ $(i \neq j)$ is the shortest path from v_i to v_j in G. The diameter of G is defined as the longest path among all shortest paths.
In this thesis it is assumed that G is connected. If G is not connected the largest connected component will be selected to calculate the diameter of G (compare [21]). To calculate the diameter of G for each vertex in G the distance to all other vertices has to be calculated. The best known expected

[1] for example a complete graph

running time for this problem is $O(n^2)$ with high probability [3]. In worst case[2] $\Theta(n^3)$ is possible, if $m = \Omega(n^2)$. Calculating the exact diameter is possible in reasonable time if needed but it is very expensive. Some heuristics are known to estimate the diameter with less running time. A heuristic with a single BFS will be introduced in this chapter. Other heuristics will be introduced in Chapter 3.

2.3 Breadth-First Search

Breadth-First Search (short: BFS) is a fundamental algorithm for the traversal of graphs. BFS is an uninformed search method which means that a priori there is no information, except the graph itself, that can be used to traverse the graph. Beginning in a start vertex $s \in V$ BFS visit each reachable vertex $v \in V$ level wise. Vertices in the same level are combined to a so-called "BFS-Level". In implementations for main memory BFS usually uses a standard Queue (or Array + Pointers with similar meaning) as data structure for searching.
The complexity of BFS is $O(|V| + |E|)$. Each reachable vertex $v \in V$ will be inserted only once into the queue and each edge will be considered around two times.

Algorithm 1 Pseudo code of Breadth-First Search according to [19, p. 536]

```
mark all vertices as unvisited
select a start vertex s ∈ G
mark vertex s as visited
distance[s] = 0
Queue Q = Queue();
Q.enqueue(s);
while !Q.isEmpty() do
  u = Q.dequeue();
  for all v adjacent to u do
    if v has not been visited before then
      mark v as visited
      distance[v]=distance[u]+1
      Q.enqueue(v);
    end if
  end for
end while
```

2.4 BFS as a method to approximate the diameter

The height of a BFS tree T is limited through the diameter D of the graph G. Furthermore, there is a bound for the diameter which can be deduced by the height of a BFS tree: $D_{BFS} \leq D \leq 2 \cdot D_{BFS}$.

Proof. $D_{BFS} \leq D$: If the diameter D is smaller than the height of the BFS tree, there must be a shorter

[2] In worst case in each iteration every single vertex and edge have to be regarded at least once. The number of iterations is the number of vertices.

way from the source s of the BFS tree to the vertex v with the longest distance to s in the original graph. But then the output of BFS was not correct because BFS calculates the distances for each vertex $v \in V$ which can be reached by the start vertex s.
$D \leq 2 \cdot D_{BFS}$: Let u and v be two arbitrary vertices of the graph G. Then the length of the path between these two vertices is bounded by the length of the path from u to s and from s to v. The distance of u and v to s is bounded by the height of the BFS tree D_{BFS}. Therefore, the distance of any two vertices and therewith the diameter in G is bounded by $2 \cdot D_{BFS}$. □

2.5 External-memory model

Vitter and Shriver defined a model for external-memory which is widely accepted in the community [4]. Their model defines a simple memory hierarchy. There is an internal-memory device with size M (main memory). This can be accessed by the CPU through its controller / cache very fast within one computation step (idealized for this model).
Then there is the external-memory device (e. g. hard disk) which has no direct limit but it must be sufficient to store the input of size N and intermediate results during the algorithm execution. It is assumed that $N \gg M$. The communication between the internal and the external memory, respectively main memory and hard disk, does not transport only one element per communication step but B elements. B is the block size, defined by the number of elements which can be transferred in each step. On real world devices, the block size of a device is not defined by elements directly. Hard disks have default sizes for their internal structures, but consecutive physical blocks from the hard disk or similar structures can be summarized into one logical block. The number of disks is denoted with D.
The main goal of this scheme is to model the I/O complexity ignoring the internal-memory computation. The I/O complexity is comparable for different algorithms in this model. Thus, this model can be used to find an approach with reduced I/O complexity. But this cannot be done for each cost. If the internal computation begins to dominate the complexity, there is no gain in the terms of running time. Also usually the external-memory devices (and operating systems) do not allow full control of the I/O operations. For optimization of an algorithm M and B have to be known in many cases. Nevertheless, the results since the development of this model showed that it has as practical value.

2.6 Scanning in external-memory

As shortly mentioned in the introduction in Chapter 1, the time to access data on an external-memory device like a hard disk is very slow compared to an access to the main memory. Depending on the device the access time can typically be between around $10,000$[3] and $100,000$ times slower [22].
A direct access from the CPU to a hard disk has a slowdown factor of around $1,000,000$. But caches

[3] This access time is reachable with modern hardware devices as Solid State Disks which did not exist when the model was developed.

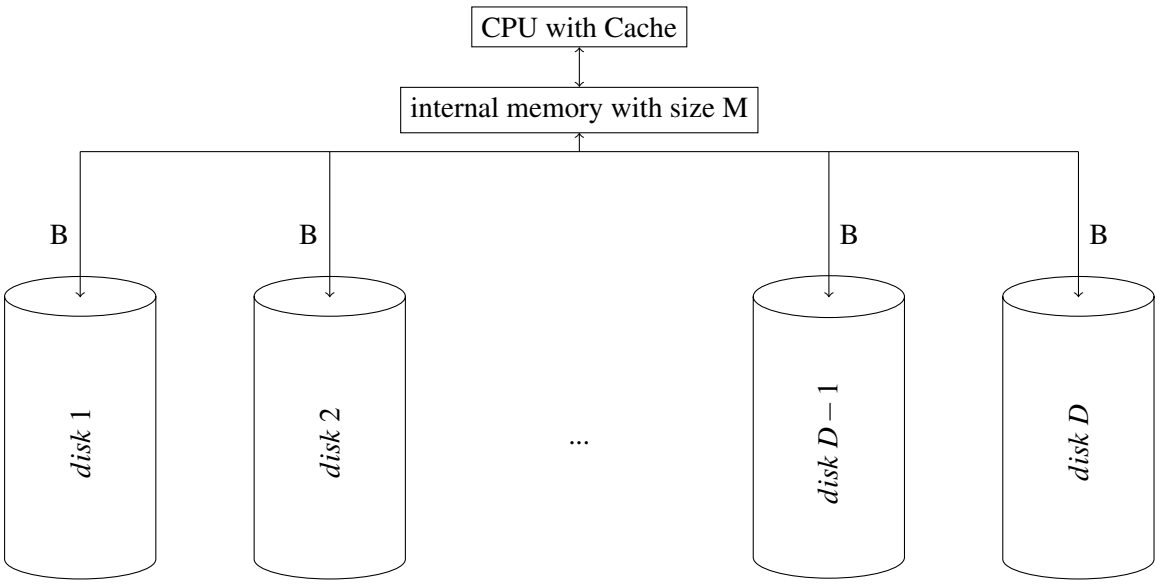

Figure 2.1: A picture of the EM-model after description of [4] with D disks. Each disk can send/receive B items in each I/O step to/from the internal memory.

are very small and thus this thesis concentrates on the gap between main memory as internal memory and hard disks as external memory.

In order to accelerate the transfer of data from the external-memory device it is assumed that the data is organized in blocks of size B on the device, where B is the number of elements which can be saved in a block.
Considering a stripe of data on a disk of length N the time to read in the data is the time to access each block plus reading the block itself. Typically reading a block is very fast. The access time dominates the process. Hence, scan has the I/O complexity $O(N/B)$ (short: $scan(N)$) with one disk and $O(N/(B \cdot D))$ with D disks. Scanning a file is the fastest operation which can be done in external-memory, if all elements have to be read at least once in given order.

2.7 Sorting and permutation of data in external-memory

External-memory sorting is based on the idea of merge sort. To improve the running time the following adaptions have been realized: runs have the size of initially $\Theta(M)$ elements instead of one.[4] The size of a run increases during the recursion. Also, not only two runs are merged in each step but k runs.[5] This leads to a I/O complexity of $O(N/B \cdot \log_{M/B}(N/B))$ (short: $sort(N)$) with one disk and $O(N/(B \cdot D) \cdot \log_{M/B}(N/B))$ with D disks [4].

[4]Each run of size $\Theta(M)$ will be sorted with an internal-memory sorter to create sorted runs.

[5]Not the whole run is kept in internal memory but a fraction of its blocks. Following blocks of a run will be loaded on the fly, when the data from the loaded blocks is consumed, until the whole run has been read.

The combination of scanning and sorting can be used to solve the problem of permuting data[6]:

Algorithm 2 Pseudo code for a typical access problem in external-memory

```
ARRAY[1 to N] X,Y,Z //N >> M
initialize X[i]=i, Y[i] and Z[i]=i for each i ∈ 1,...,N
create a permutation of Z and assign it to Z
for (i=1;i<=N;i++) do
  X[i]=Y[Z[i]]
end for
```

Algorithm 2 has an I/O complexity of (N) in worst case, because the element Y[Z[i]] and Y[Z[i+1]] can be in different not consecutively following blocks with a high probability depending on which permutation was created. Assuming that only one block can be hold in internal-memory for simplicity, for each random access $O(1)$ I/O has to be done in worst case. This is a loss of the factor B. The solution is not to assign the values of Y[Z[i]] directly to X[i] but to build tuples and do a few scanning and sorting operations. After that each value Y[Z[i]] will have received its right position in the array X.

Algorithm 3 Solution for the access problem in external-memory

```
ARRAY[1 to N] X,Y,Z //N >> M
initialize X[i]=i, Y[i] and Z[i]=i for each i ∈ 1,...,N
create a permutation of Z and assign it to Z
SCAN Z and build a sequence of tuples T with T[i] = (Z[i],i) //(value, position)
SORT T by value
SCAN Y and T in parallel and do
for (i=1;i<=N;i++) do
  T[i].value=Y[i]
end for
SORT T by position
SCAN T and X in parallel and do
for (i=1;i<=N;i++) do
  X[i]=T[i].value
end for
```

By using sorting as access pattern in Algorithm 3, the best case I/O complexity is not $O(N/B)$ as in Algorithm 2, but the worst case complexity shrunk down to sorting complexity, what is the best reachable I/O complexity for a computation based on comparison [23]. The same procedure was used to adapt the semi external-memory SSSP implementation (compare Section 4.4).

2.8 STXXL

STXXL is a software library developed in C++ by Dementiev, Kettner, Sanders [24] and later maintained and extended by many researchers.[7] STXXL is thought as an implementation of the C++ STL

[6] The example is based on lecture notes of "Efficient Algorithms" held in summer term 2009 by Meyer.

[7] The current (2011) main developers of the STXXL are: Beckmann and Singler.

(Standard Template Library) for huge data sets. The library works on various operating systems as Linux, Microsoft Windows, Mac OS X and FreeBSD.
The STXXL supports sorting, stack, queue, dequeue, vector, search trees and priority queues based on a sequence heap. Matrix operations were not supported in the past [24].
As a special feature, STXXL also supports pipelined algorithms. TPIE and LEDA-SM, two comparable external-memory libraries did not support that feature when the previous mentioned article about STXXL was published.[8] "The original purpose of pipelining in the STXXL was to save I/Os by passing between algorithmic components without writing them to disk in between." [25]. In [25] the idea of pipelining in STXXL was improved to create a task parallelism. In some special cases the library ensures automatically that the specific code segment will be parallelized. For the other operations in the pipeline the programmer is able to define the parallelism with inserting asynchronous nodes [25].

2.9 External-memory BFS

Internal-memory BFS has two parts where it randomly accesses data: Extract all neighbors of u if u was the last element from Q and check if these neighbors have been visited before (compare Section 2.3). The problem of determining if a neighbor has been visited before can be solved by considering the neighbors of vertices in current level $t-1$ along with the vertices in level $t-2$.
The following has been proved in [17]: $L(t) = \{N(L(t-1)) - duplicates\} \setminus \{L(t-1) \cup L(t-2)\}$, where $N(L(t))$ is the multiset of neighbors of vertices in level t.
This fact was used by Munagala and Ranade in [17] to implement a BFS algorithm for external-memory. In this thesis, this algorithm will be referred to as MR-BFS. The I/O complexity of MR-BFS is $O(min\{n+sort(n+m), sort(n+m)+L\cdot scan(n+m)\})$, where L denotes the number of BFS levels. For dense graphs or graphs with short diameter this can be acceptable but it is very slow for sparse graphs with large diameter.
Mehlhorn and Meyer [5] improved the I/O complexity of BFS computation for sparse graphs by addressing the problem of extracting neighbors without spending $\Omega(1)$ I/O for each access. Their BFS algorithm reorganizes the graph data on the disk for this purpose. In this thesis their approach will be called MM-BFS according to the notation used in other publications (e. g. [26, 27]). The adjacency lists are organized in clusters such that if one list of a cluster is required for BFS, the others are also required in a short time. If adjacency lists are used while they are in the main memory, they will not be reloaded. Otherwise, they are stored in an efficiently data structure (hotpool). This reduces the I/O-complexity.
However, scanning the hotpool produces extra costs. That leads to the trade-off that the I/O-complexity is not dominated by $O(n)$ any more but by $O(n/\sqrt{B})$. The resulting total complexity is $O(n/\sqrt{B} + sort(n))$ for sparse graphs, with $m = O(n)$.
BFS can sometimes be computed faster if the diameter is known. This was validated with experiments

[8] Even today this feature is not supported. In the library TPIE are some features for streaming, which were developed for streaming algorithms and not for pipelining as a method to save I/Os in many cases with a generic structure.

in [27]. But in many heuristics one or more BFS computations are used to approximate the diameter. But in the external-memory case, even this is not viable. The diameter is needed to choose a better access pattern for BFS computation. Hence, approaches with better I/O complexity are needed to approximate the diameter.

2.10 Minimum Spanning Tree

A spanning tree T is a subgraph of a connected graph G with the properties that T has $|V|-1$ edges $e \subseteq E$ and T is connected. The set of vertices of T is equal to the set of vertices from G. A minimum spanning tree (short: MST) has the additional property that the sum of edge weights is minimum.
Early algorithms for spanning tree computation in internal memory were developed by Borůvka (1926) [28], Jarník (1930) [29] / Prim (1957)[9] [30] and Kruskal (1956) [31]. An algorithm for minimum spanning tree computation with complexity of $O(n+m)$ has been developed in [32] for a specific computation model. For the usual used computation model for internal memory the bound of $\Theta(n\log(n)+m)$ can be reached with Prim's algorithm.
Minimum spanning trees or in general spanning trees are used to create a tree out of a given graph G for solving problems which are easy to solve on trees but difficult on general graphs. In many external-memory approaches (MM-BFS, semi-external SSSP,...) a spanning tree is used for the preprocessing steps to organize data in a way that it can be accessed much faster in the following computation step, for example an Euler-Tour technique.
The randomized implementation for minimum spanning tree in external-memory has an I/O complexity of $O(sort(n))$ [33].
MST is used in some approaches for finding the diameter of a graph.[10] In general the resulting MST can have an arbitrary depth. Also the MST of a graph G is not necessarily unique. Figure 2.2 shows an example.

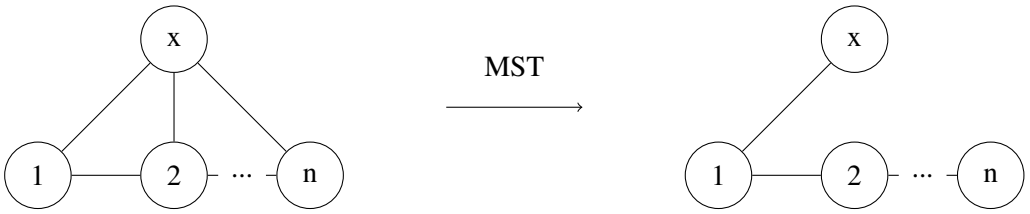

Figure 2.2: The left part of this figure shows a list graph with an extra vertex x. This extra vertex is connected to all n vertices in the list. One possible MST of this graph has one connection from x to the fist vertex of the list and then all edges from the list. The actual diameter of this graph is 2. The diameter of this possible MST is $n-1+1=n$.

For the fully connected graph with n vertices, Rényi and Szekeres showed that the expected diameter of a random spanning tree is $O(\sqrt{n})$ while the actual diameter is 1 [34]. Nevertheless in Section 3.4 a

[9] Prim rediscovered the algorithm from Jarník
[10] For example for upper bound computation. See Section 3.2

heuristic will be presented which is based on work by Brudaru in [35]. Her work is based on the idea that a MST can be improved iteratively with some local rules.

2.11 Euler-Tour

The Euler-Tour algorithm is based on the Euler circuits or Eulerian trail. Leonhard Euler used this technique first to solve the "Seven Bridges of Königsberg" problem in 1736 [36]. The statement of this problem was to use each of the seven bridges once and finally reach the starting region again. Euler transformed it into a graph problem: All regions of Königsberg which are connected via a bridge are vertices and the bridges are edges. For the "Seven Bridges of Königsberg" problem Euler showed, that it is not possible to find a path without a detour through a visited bridge with his new invented Euler circuits.
To find such a circle or tour can be difficult in general[11] but it is comparatively easy for trees.
Trees with edges in both directions for every vertex always have an Euler-Tour. This is based on the fact that all vertices have an equal in- and out-degree and this is one requirement that needs to be fulfilled for the existence of an Euler-Tour (or even Euler circuits).
In the external-memory model, an Euler-Tour for trees can be computed with $\Theta(sort(N))$ I/Os [38]. The algorithm builds a list of edges which represents the tour through the tree. The task is to find the successor of an edge as a list element. This can be done by using a cyclic order and list ranking. Details are explained in [39, p. 108–118] for RAM model algorithms and in [38] for external-memory approaches.

2.12 SSSP

Single source shortest path is the problem of finding the shortest path from a single start vertex s to all other reachable vertices $v \in V$. The edges $e \in E$ are weighted with edge weights $c(e) \in \mathbb{R}^+$.[12] The weights are the main difference from the BFS, that can compute the distance of all vertices $v \in V$ to s for unweighted graphs. Reaching a vertex v from the source s via a second possible path of more vertices may lead to a smaller sum of weights on the path than e. g. a direct connection from s to v. See Figure 2.3 as example.

[11] In [37] it is mentioned that computing an Euler-Tour on a tree is equivalent to computing DFS on the same tree. For general graphs finding an Euler-Tour is linear in the number of edges with the algorithm from Carl Hierholzer.

[12] Negative edge weights are not allowed. For negative edge weights the correctness of the invariant of the algorithm is not guaranteed. For such a case an algorithm like Bellman-Ford would be an option. But this is not a topic of this thesis.

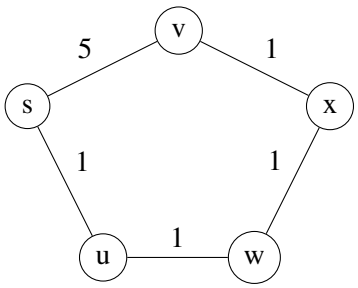

Figure 2.3: The direct path from s to v has not the shortest path length. SSSP finds the path from s to v with $p = \{s,u,w,x,v\}$.

If for some reason, the distances in a weighted graph G have to be calculated with BFS, this is possible if G is transformed to an unweighted graph G', where an edge $\{u,v,c(e)\}$ is replaced by a path with length c(e) with dummy vertices in the path between u and v, which are ignored from the final output. See Figure 2.4 for an example. For a $c \in \mathbb{N}$ it is not too difficult to transform G into G' but very costly. The complexity of the computation is now depending on the edge weights. The complexity of BFS could grow up to $\Theta(w \cdot (n+m))$, where w denotes the heaviest edge weight in G. For small w this might be acceptable but for growing w, BFS is not a feasible alternative.

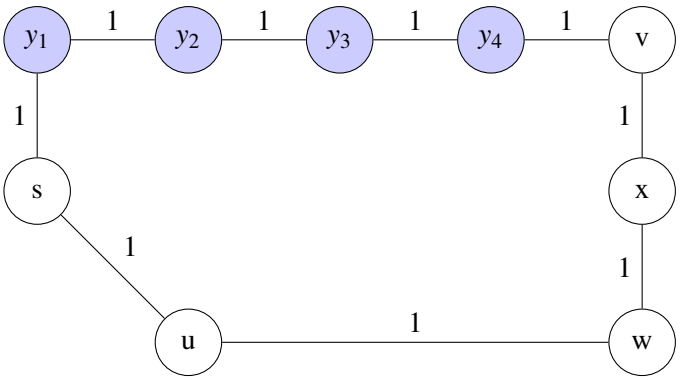

Figure 2.4: Transformed graph G from Figure 2.3 into a graph G' on which BFS can be used. Vertices $y_1,...,y_4$ are inserted to replace the edge $\{s,v\}$ with weight 5.

In 1959 Dijkstra published the underlying ideas for his solution of the SSSP problem [40]. Starting from a single vertex s as source, the distance from s to all other reachable vertices is computed iteratively. In each iteration the vertex v with the shortest tentative distance to all vertices in the set S with final distance is added to the set S. After v has final distance, the distances to its non-final neighbors is updated. Because of the fact that the distance of a vertex with final distance will not be changed again, v iteration steps are needed.
Since the first publication, solutions for solving SSSP have developed a lot.[13] With a priority queue (e. g. a heap) as data structure, SSSP can be solved with a complexity of $O(n \cdot \log(n+m))$. It is not

[13] Dijkstra used a linked list as data structure.

possible to achieve a better complexity by comparison based algorithms [41, p. 180–193]. In the past few years some algorithms have been developed which can solve SSSP in $O(n)$ in average case [42].

Algorithm 4 Basic algorithm for Single Source Shortest Path

```
distance[1...n] = ∞
select a start vertex s
distance[s] = 0
PriorityQueue PQ;
for all neighbors u of s do
  //tentative distance is the weight of the edge between s and u
  distance[u] = c(s,u)
  //vertex u is inserted into the PQ with its tentative distance as priority
  PQ.insert(u);
end for

while !PQ.empty() do
  //get the next element
  vertex u = PQ.del_min()
  for all neighbors v of u do
    if (distance[v] > distance[u] + c(u,v)) then
      distance[v] = c(u,v)
      PQ.insert(v)
    end if
  end for
end while
```

2.13 Semi external-memory SSSP

In "Design and Implementation of a Practical I/O-efficient Shortest Paths Algorithm " Meyer and Osipov published a semi external-memory implementation of the SSSP algorithm with I/O complexity of $O(\sqrt{\frac{n\ m\ K}{B}} + MST(n,m))$.[14] [1]. In the previous work of Meyer and Zeh [43] some basic ideas of the implementation by [1] were developed. The main problem of [43] is that the constants of their approach are too big that it can be executed in realistic time in practice with known development tools and algorithms. Hence, Meyer and Osipov changed the side conditions in a way that their implementation would have better constants but their implementation is not fully independent from the size of the main memory.

- For each vertex there is at least one bit memory free in internal memory. This bit is a flag if the vertex has reached final distance.
- The clustering of the input graph is done ignoring the edge weights. The consequence is that clusters cannot be organized as efficiently as they would have been if the edge weights were

[14] This I/O bound depends on the interval $[1,K]$ of the edge weights. $MST(n,m)$ is the I/O-complexity of a minimum spanning tree computation. It can be $sort(m)$ randomized or $sort(m) \cdot \log(\log(nB/m))$ deterministically.

considered.

With these two assumptions Meyer's and Osipov's approach reaches reasonable I/O complexity in many cases. In [1] the edge weights for the experiments were generated randomly from an interval $[0,k]$. For an input graph with given edge weights this implementation needs to be adapted.
Figure 2.5 shows the main stages of the semi-external SSSP implementation.

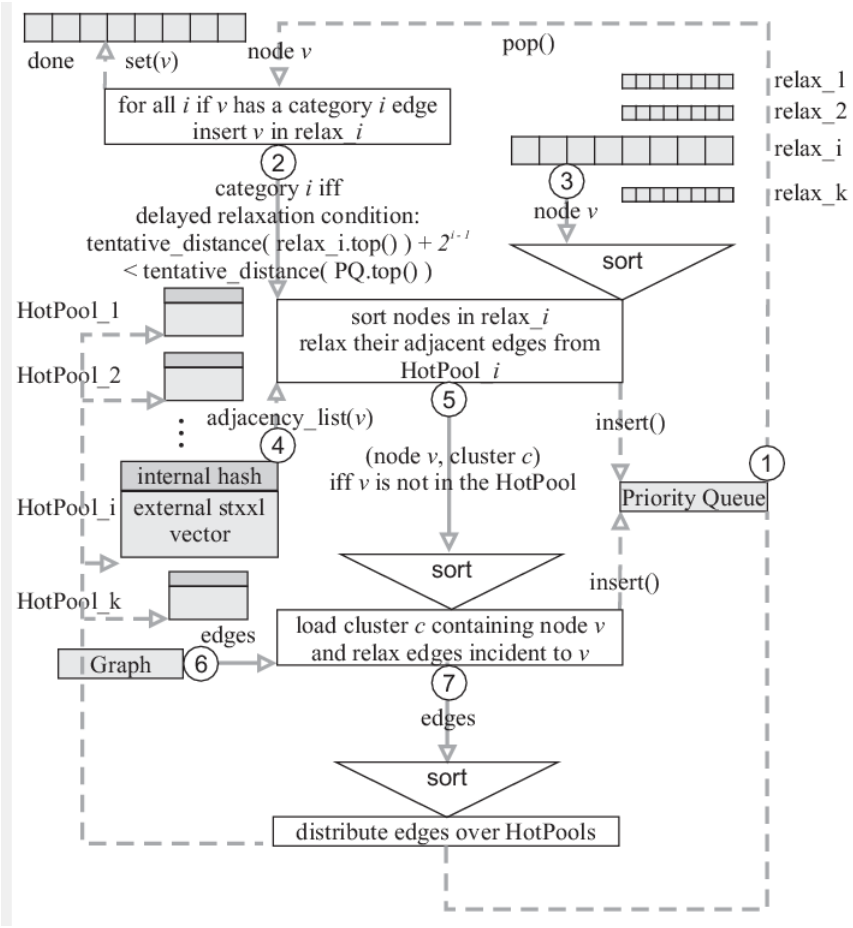

Figure 2.5: Diagram of the semi-external SSSP implementation [1]. The implementation uses k hotpools to keep some elements with edge weights from interval 1 to k in internal memory. The priority queue also considers k intervals for relaxing tentative edge weights.

2.14 Probability theory

For the proof of a lower bound in Section 4.5 some basics from probability theory are needed.
A probability $\mathrm{P}(E)$ for an event E has a value between 0 and 1, so $0 \leq P(E) \leq 1$. $\mathrm{P}(E) = 0$ means that the event E will never happen. If $\mathrm{P}(E) = 1$, then E will always happen. The subset E of the sample space Ω contains elements from the sample space and only from this space.[15] If $E = \Omega$ then $\mathrm{P}(E) = 1$

[15] It is possible to have many sample spaces. But in this thesis it is assumed that each probability is based on a single sample space.

and if $E \cap \Omega = \emptyset$ then $\mathrm{P}(E) = 0$.
It is allowed to have many events E_i, where the events can be pairwise disjointed. Disjointed means that the intersection of two different events E_i and E_j is empty. For dependent events the probability of an event E_i can depend on the fact that another event E_j happened or not. This is the *conditional probability* $\mathrm{P}(A|B) = \frac{P(A \cap B)}{P(B)}$. If two events A and B are independent it holds that $\mathrm{P}(A|B) = P(A)$.

Two important parts of the probability theory for this thesis are the *Bernoulli process* respectively the *Bernoulli distribution* and the *Chernoff bound*.
The idea behind the Bernoulli distribution $B(n;p;k)$ is to determine the probability of occurrence of k hits for an experiment within n trials. In one experiment, the probability for such a hit is p. The probability for a miss is $q = 1 - p$. Different experiments are independent of each other. The formula that describes this distribution is $B(n;p;k) = \binom{n}{k} \cdot p^k \cdot (q)^{n-k}$ with $\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$ and $n \geq 0$. $n! = n \cdot (n-1) \cdot ... \cdot 2 \cdot 1$ is a notation for factorial. A special case is $0! = 1$.
The expectation $\mathrm{E}[X] = n \cdot p$ of the Bernoulli distribution with the random variable X is a value that tells how many hits are expected in mean.[16] For the expectation of two Bernoulli processes, $\mathrm{E}[X+Y]$ =$\mathrm{E}[X]$+$\mathrm{E}[Y]$, holds the so-called linearity of expectation.

The Chernoff bound [44, 45] describes the probability for a sequence of independent Bernoulli processes $X_1, X_2, ..., X_n$ to be a multiplicative factor $(1+\delta)$ away from the expectation μ. The probability for a hit is $\mathrm{P}[X_i = 1] = p$ for each X_i $(i = 1, ..., n)$.The probability for a miss is $\mathrm{P}[X_i = 0] = 1 - p = q$. The expectation value is $\mu = p \cdot n$ which describes the number of expected hits $(X_i = 1)$.

Chernoff bound

Let $X_1, X_2, ..., X_n$ be a sequence of n independent Bernoulli processes (or random variables).
$\mathrm{P}[X_i = 1] = p$, $\mathrm{P}[X_i = 0] = q$ and $\mu = p \cdot n = E[\sum_{i=1}^{n} X_i]$.
Then it holds for all $\delta > 0$: $\mathrm{P}[\sum_{i=1}^{n} X_i \geq (1+\delta) \cdot \mu] \leq e^{-\frac{min\{\delta, \delta^2\}}{3} \cdot \mu}$.
Furthermore it holds for all $\delta \in [0,1]$ the following can be occurred: $\mathrm{P}[\sum_{i=1}^{n} X_i \leq (1-\delta) \cdot \mu] \leq e^{-\frac{\delta^2}{2} \cdot \mu}$.

The Chernoff bound is often used in computer science to analyze randomized algorithms. The X_i are random variables which can happen with a probability p $(X_i = 1)$ and do not happen otherwise $(X_i = 0)$. This will also be the case in the proof in Section 4.5.

[16] It is also possible to construct distributions where the expectation will never show up. In this thesis this is not explained in detail because it will be of no interest in the proof in Section 4.5.

2.15 Summary

Graphs are a useful abstraction to model many problems. Because of the diversity of the graph model many applications and algorithms have been developed for this model in the past: BFS, SSSP, MST, Euler-Tour and others. In many cases it is possible to use more than one algorithm on a given graph to compute the information which should be extracted in a few more computation steps. One example will be the approach in this thesis which will be based on BFS and SSSP.
Due to the fact that an access to the disk is very costly, the external-memory model has been developed by Vitter and Shriver, which is widely accepted. Two important basic algorithms are used as access pattern for external-memory algorithms: scanning with I/O complexity $O(n/B)$ and sorting with I/O complexity $O(n/B \cdot \log_{M/B}(n/B))$, where B is the block size and M the size of main memory. Many of the presented algorithms have been implemented in this model in a efficient way. But for other popular algorithms like DFS there are no implementations that are faster than I/O-complexity of $\Theta(n)$ for spare graphs [46].

Chapter 3

State of the Art

In this chapter various techniques will be described to approximate the diameter of a graph. As basic concepts double sweep lower bound and other simple BFS based heuristics will be described briefly for lower bound. Then a heuristic for an upper bound by Crescenzi et al. [2] will be introduced. Finally, an existing external-memory diameter approximation will be presented. The approach presented in Chapter 4 will have to compete with the existing external-memory heuristic (see Chapter 5).

3.1 Double sweep lower bound

As described in Section 2.4, with a single BFS the diameter can be approximated in the interval $D_{BFS} \leq D \leq 2 \cdot D_{BFS}$. The diameter D_{BFS} of the BFS tree depends on the randomly chosen start vertex s. To improve the diameter of the BFS tree a second BFS is executed with a new start vertex s'. s' is a vertex at farthest distance from s. This heuristic is called double sweep lower bound, or short **dslb** [47, 21].
Crescenzi et al. [2] showed that the error of the resulting diameter of the second BFS can also be bad, but not as bad as the trivial bound calculated with one BFS. Their results show that for many practical cases the dslb is, in fact, very tight. There are many existing graph classes for which the dslb is equal to the exact diameter. This is the case for lists, grids and many other regular structures.
A worst case graph in [2] is constructed in the following way (see Figure 3.1):

- Grid with k rows and $1 + \frac{3}{2}k$ columns. All vertex pairs with Euclidean distance at most $\sqrt{2}$ are connected.
- p additional vertices $x_1, x_2, ..., x_p$ with $p \gg k$. These vertices are connected to all neighbors of the middle point of the upper row.
- One additional vertex y which is connected to the middle point of the lower row.

With a high probability one vertex out of the set $x_1, x_2, ..., x_p$ is chosen as start vertex for large p. y will be the vertex with highest distance to any x_i. The height of the BFS tree of y will be $k+1$, but the diameter is in fact $\frac{3}{2}k$.

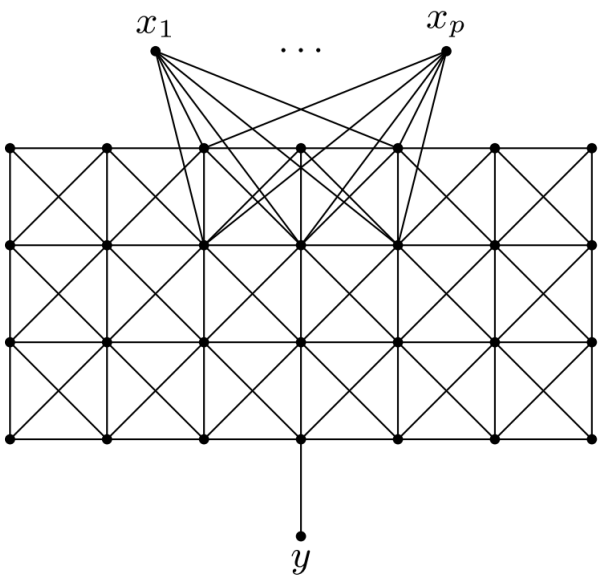

Figure 3.1: One possible input for which the bound of dslb is not tight [2].

3.2 Heuristics for computing an upper bound

In this section three heuristics to find an upper bound for the diameter of a graph G are presented. These three heuristics are described in [21].

Tree upper bound: As noticed before, the diameter of a spanning tree T can be a very bad approximation of the actual diameter of a graph G. Nevertheless, the diameter of a spanning tree T can not be smaller than the diameter of G. A spanning tree can be computed very fast in internal-memory as well as in external-memory. [21] claim that this bound is always better than the corresponding trivial upper bound by computing a single BFS and taking the upper bound as two times the height of the BFS tree.

Highest degree tree upper bound: Vertices with a high degree are likely to produce BFS trees with a small diameter. The vertices are sorted in descending order and the vertex with highest degree is chosen as source for a BFS. By using more and more high degree vertices as start vertex for a BFS the upper bound gets more tight. "This is motivated by the fact that real-world complex networks are known to have some vertices with very high degree[...]" [21].
The quality of this approach depends on the number of iterations. The results of the experiments from [21, Table 1] show that *highest degree tree upper bound* needs less iterations for tight bounds compared to *tree upper bound*, e. g. 34 iterations versus 1572 for a web graph.

Random tree upper bound: Similar to *highest degree tree upper bound*. Iteratively BFS from different sources are computed. Sources are not chosen by a special criterion but randomly. In many cases

this method needs more iterations than using the degree as selection criterion.

3.3 Fringe: improved upper bound heuristic

Crescenzi et al. developed a new heuristic to compute an upper bound of the diameter, which was implemented as an internal-memory approach in [2]. In the beginning some definitions are required:

- $G = (V, E)$ is an unweighted and undirected graph. Also, it is assumed that G is connected. Otherwise the largest component of G is considered.
- Let $ecc(u) = max_{v \neq u} distance(u, v)$ be the eccentricity of $u \in V$. If u is the source of a BFS tree, $ecc(u)$ is the height of the tree.
- T_u is the BFS tree rooted at u and $2 \cdot ecc(u) \geq diameter(G)$.
- The fringe of u, denoted as $F(u)$, is the set of vertices $v \in V$ such that $distance(u, v) = ecc(u)$.
- $B(u) = max_{z \in F(u)} ecc(z)$
- The upper bound is defined as $U(u)$, computed by the fringe algorithm.
- $$U(u) = \begin{cases} 2 \cdot ecc(u) - 1 \; if \; |F(u)| > 1 \; and \; B(u) = 2 \cdot ecc(u) - 1 \\ 2 \cdot ecc(u) - 2 \; if \; |F(u)| > 1 \; and \; B(u) < 2 \cdot ecc(u) - 1 \\ diameter(T_u) \quad otherwise \end{cases}$$
- Let r be a random source of a BFS. Furthermore let a be a source for a second BFS with maximum distance to r. Finally, let b be is a vertex with farthest distance to a.

With these components the Fringe algorithm works the following way:
1. Let r, a, and b be the vertices identified by double sweep method (using two BFS).
2. Find the vertex u that is halfway along the path connecting a and b inside the BFS-tree T_a.
3. Compute the BFS tree T_u and its eccentricity $ecc(u)$.
4. If $|F(u)| > 1$, find the BFS trees $T_z \in F(u)$, and compute $B(u)$:
   – If $B(u) = 2 \cdot ecc(u) - 1$, return $2 \cdot ecc(u) - 1$.
   – If $B(u) < 2 \cdot ecc(u) - 1$, return $2 \cdot ecc(u) - 2$.
5. Return the diameter $\text{diam}(T_u)$.

Crescenzi et al. noticed that in practice $|F(u)|$ is reasonable small. One iteration of the fringe algorithm needs $|F(u)| + 3$ times BFS [2, Theorem 1]. In internal-memory this might be acceptable but for disk-based algorithms this is not feasible. Nevertheless, in many cases the fringe algorithm can help to find the exact diameter with less iterations BFS than brute force with a BFS for each vertex. During their experiments Crescenzi et al. discovered that in many cases the lower bound (dslb) is equal to

their upper bound (Fringe), which can be only be the case if the lower bound is tight (compare Section 3.1).
The only disadvantage is that Fringe probably needs many Breadth-First Searches if $|F(u)|$ is large and sometimes Fringe does not find a tight upper bound. In fact there have been cases, in which about one thousand iterations were needed to find a tight bound equal to the dslb. However, the results of Fringe are usually better compared to the results of the heuristics in Section 3.2.

In this thesis the Fringe algorithm will be applied to compute an upper bound on graphs used for testing.

3.4 External-memory spanning tree heuristics

In her thesis, Brudaru developed [35] heuristics based on the idea of computing a spanning tree first and then refining it. The target was to develop an external-memory approach with less than $O(n/\sqrt{B})$ I/Os that BFS would need for a single computing step. Computing a minimum spanning tree can be done with an I/O complexity of $O(sort(n))$ for sparse graphs $(m = O(n))$. The problem is that the diameter of the MST can be arbitrarily bad compared to the actual diameter (compare Section 2.10). However, a MST can be used as starting point for the approximation of a BFS tree for the chosen source with less I/Os.
There are two main possibilities to refine the tree iteratively:

- lazy refinement of the tree: search for better local connections and update the information in the end of each iteration.
- Greedy refinement of the tree: search for better local connections and update the information during the iteration.

The greedy refinement is implemented as follows: the height $h_{i+1}(u)$ of a vertex u in the iteration $i+1$ is initialized with the height $h_i(u)$ from the previous. Then, the height is updated by scanning the edges of the original graph and the heights from the previous iteration in ascending order to update the subgraph downwards and then in descending order the update the graph upwards. To send the information of an update to another vertex, the time-forward-processing strategy [48] and the external-memory priority queue from STXXL is used.
A possible way to improve this approach for diameter approximation, is to use a technique similar to dslb by finding a source with farthest distance to the randomly chosen source of the MST with an Euler-Tour in sorting complexity to improve the height of the tree and therefore the approximation ratio.

Chapter 4

Parallel clustering growing approach

In the previous chapter, some approaches like double sweep lower bound or the Fringe algorithm by [2] based on BFS have been introduced. As mentioned before, these approaches are not viable for large data sets, because in the worst case they would lead to an I/O complexity of $\Omega(\frac{n}{\sqrt{B}})$ for sparse graphs. The heuristic in Section 3.4, based on spanning tree computation, also has high constants. Furthermore, it is unclear how many iterations of refinement are needed for an input graph G.
Therefore, this chapter will introduce a new heuristic. This heuristic is based on the ideas in [6]. In this paper two approaches are introduced, which will be presented in the following section. The second approach will be used later for implementation.

4.1 Theory

4.1.1 Euler-Tour based approach

In this subsection, the approach in [6] based on the preprocessing of MM-BFS will be presented. The preprocessing is used to shrink the graph before a BFS is executed on the shrunken graph.

The implementation of MM-BFS (refer to Section 2.9) uses the following steps as preprocessing:

- Compute a MST.
- Build an Euler-Tour on the MST.
- Split the vertices into sets of length k (chunk size) on the Euler-Tour output and remove duplicates.
- Use this information to create files for the hotpool.

These steps can also be used for shrinking the graph and doing a BFS on the shrunken graph after that. Instead of building files for the hotpool, all vertices in the same set can be shrunken to one vertex. The number of vertices is reduced from n to $\Theta(n/k)$. On sparse graphs with $m = \Theta(n)$ and $k = \Theta(B)$, an

I/O complexity of sorting can be reached if the randomized MST algorithm is used.
Afterwards BFS is computed on the condensed graph. The approximated diameter is twice the height of the BFS tree. Generally, an I/O complexity of $O\left(\sqrt{\frac{\frac{n}{k}\cdot(\frac{n}{k}+m)}{B}}+sort(\frac{n}{k}+m)+ST(\frac{n}{k},m)\right)$ can be obtained for the BFS computation. $ST(\frac{n}{k},m)$ is the I/O complexity of the spanning tree.
The weakness of this approach is the expected multiplicative error of this approach. It is $\Theta(k)$ in the worst case. This is because BFS only computes distances on unweighted graphs. Each chunk could have a length of $\Theta(k)$. This is reduced to an edge with weight 1. The sum of the diameter determining path length would be smaller by a factor $\Theta(k)$. Hence, the parallel clustering growing approach has been developed to shrink the input to a weighted condensed graph for a better approximation guarantee.

4.1.2 Parallel clustering growing approach

This approach uses a parallel BFS computation to shrink the graph G into a graph G'.[1] The shrunken graph G' has weighted edges. After condensing the graph, a SSSP is computed to determine the distances between the vertices.

- Choose master vertices uniformly and independently with probability $1/k$.
- Build an Euler-Tour of graph G and choose each k-th vertex being a master (if the vertex has not been selected as a random master yet).
- Run a local BFS from each master vertex "in parallel". This can be done by $O(k)$ scanning and sorting steps. The vertices of the level $t-1$ are stored in a sorted vector. The edges will be scanned and for each unexplored vertex v being a neighbor of a vertex u in the level $t-1$, the vertex v will be marked as a member to the cluster of u. Note that the I/O complexity depends on the size of k.
- When all vertices have been reached by a master vertex through the local BFS, compute the weight of the edges between clusters. The clusters are represented by their masters and therefore the weight of an edge is the shortest path between the two master vertices.
- Compute the distances between the master vertices with SSSP.

This approach has an expected multiplicative error of $O(\sqrt{k})$ instead of $O(k)$. The ideas of the proof in [6] will be discussed at the end of this section. The random master vertices are chosen to guarantee this bound. The deterministic chosen masters guarantee that there is no vertex left with a distance greater than k to a master vertex.
The parallel BFS step to construct G' with deterministically and randomly selected master vertices has an I/O complexity of $O(k\cdot scan(n+m)+sort(n+m)+ST(n,m))$.[2] The spanning tree is needed

[1] In [6] G' is named G''_k
[2] Compare lemma 2 in [6]

to determine the set of deterministic master vertices.
The parallel cluster growing itself has an I/O complexity of $O(k \cdot scan(n+m) + sort(n+m))$. Each non-master vertex has distance of at most $k-1$ to a master vertex. Thus, the adjacency lists have to be scanned $k-1$ times. During the scanning of the adjacency lists, $O(sort(n+m))$ I/Os are needed in total for fringe vertices and to sort neighbor vertices. The Euler-Tour is hidden in the sorting term as an additional constant.

After producing the weighted graph G' with a few more sorting and scanning steps the final distances between the master vertices can be computed. Now G' can be used as input for SSSP with which the diameter can be computed. The I/O complexity over all including the SSSP is $O(n \cdot \sqrt{\frac{log(k)}{(k \cdot B)}} + k \cdot scan(n) + sort(n))$ for sparse graphs. For this bound, the I/O complexity of the SSSP by Meyer and Zeh is assumed [43].
In Section 4.2, its predicted implications will be discussed. Also, the choice of a start vertex s for the SSSP will be described.

Sketch of the proof of the expected approximation bound $O(\sqrt{k})$:

The proof in [6] was split into two parts, the case that diameter D_G is smaller or equal $2 \cdot \sqrt{k}$ and the case that it is greater than $2 \cdot \sqrt{k}$.
For the case $D_G \leq 2 \cdot \sqrt{k}$, the argument is that each shortest path $P = w_0, ..., w_{D_G}$ in G has at most $2 \cdot \sqrt{k}$ edges. Each vertex W_i could be matched to a different cluster in G'. Even if each master is as far away from the respective vertex on P as possible, it can be at most a distance of $2\sqrt{k}$ as that is the bound on diameter in G. Hence, the diameter in G' is bounded from above by $4 \cdot \sqrt{k} \cdot D_G$.

For the case $D_G > 2 \cdot \sqrt{k}$ the argument is as follows. Now, the path P is split into sub-paths P'_i. Each sub-path has between $\sqrt{k}$ and $2 \cdot \sqrt{k}$ edges. In [6] each sub-path is considered separately. With the argument that each sub-path will be reached by the nearest master in at least $t \leq k$ steps, the argument is that the length of a detour for a sub-path is bounded by $O(k)$ and the linear expectation of the sum over the expected detours of all sub-paths, the ratio of the diameter in G and G' is bounded by $O(\sqrt{k})$ with high probability. Hence, the diameter can be increased by a factor of $O(\sqrt{k})$ with high probability.

4.2 Implementation details

In transforming this algorithm into a practical implementation, the following issues needed to be solved:

1. Which master vertex should be the start vertex s for the SSSP?

2. Is one execution of SSSP with one start vertex s enough? Is a set of start vertices $v \in V$ needed?

Due to the fact that no information about the input graph or the condensed graph is available, a randomly selected start vertex s is used first.
The answer to the second question is implied by the answer to the first one. The height of the SSSP tree with the random source s as root is at least half of the diameter of the condensed graph. Nevertheless, a deeper (SSSP) tree can often be found by using the double sweep lower bound technique as for BFS (compare Section 3.1). Hence, a second SSSP with a start vertex s' is executed. s' is a vertex in V with largest distance to the randomly chosen start vertex s. This is affordable because the SSSP is executed on a significant smaller graph.

Note that the resulting largest path is not the final result. An error correcting variable d has to be added twice. The value of d is the largest distance of any vertex $v \in V$ to its master vertex. If all vertices are selected as master vertices, the value of $d = 0$. Otherwise d is between 1 and k, if the deterministic master vertices ensures the largest distance of at most k for each vertex $v \in V$ to a master vertex. Without the correcting factor d the result would be smaller than the height of the BFS tree with s' as root in the original graph G.
The Figures 4.1 and 4.2 give a visual proof sketch for the necessity of adding d twice to the result of the condensed graph G'.

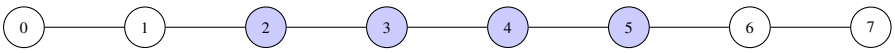

Figure 4.1: The blue colored vertices should be master vertices. The value of d is 2 in this case.

Figure 4.2: The condensed graph has a diameter of 3.

4.3 Internal-memory implementation

4.3.1 Aims

Before the implementation of an external-memory version, an internal-memory prototype has been developed. This prototype had the task to check if the ideas in [6] are practically viable. Working with hard disks makes testing very difficult and slow. Therefore the prototype does not use the STXXL and is not designed for arbitrary large data sets but to work on the HPC cluster at Goethe Universität. The acronym HPC stands for "High Performance Computing". The prototype is written in C++[3].

4.3.2 The implementation details

The prototype works with the same input type as the Fringe algorithm presented in [2]. Their application accepts two types of inputs: nde-text-file and nde-binary.[4] The first entry in a nde file is the number of vertices n. Then follows for each vertex $v_0, v_1, ..., v_{n-1}$ a pair of values (v_i, degree of v_i). The degree is the number of connections from the vertex to other vertices. After that the rest of the file stores pairs of vertices u and v with $u, v \in V$ as the edges. A graph stored in a nde-file does not guarantee that the graph is connected.
To work with other test data than the graphs collected in [2], a converter converting graph data, generated by the external-memory BFS implementation presented in [27], into the above format has been written.

After reading the input the prototype tests the connectivity of the input graph. In the case that the graph is not connected the largest component is selected (compare Section 2.2). Afterwards the master vertices are selected on the regarded input data G uniformly and independently with the probability $\mu = w/n$, where w denotes the number of desired master vertices.
In the implementation, no deterministic master vertices are selected. Therefore, the maximum distance between master vertices is not bounded. There are two reasons for this decision. Firstly, the probability that the maximum distance between randomly chosen master vertices differs much from the distance ensured by the deterministic masters is not very large. Secondly, it is of interest if this preprocessing step can be saved in practice. Also, the size of the condensed graph is smaller. This is also important for the external-memory implementation.

The selected master vertices are used as starting points for a parallel clustering of the graph G. This is implemented similar to BFS. Each master vertex is marked and queued in a queue Q. All other vertices are unmarked. After that each master vertex is dequeued again in one loop step. All neighbors v of the dequeued master vertex, which have not been marked before, are marked as a member of the

[3] Compiled with g++-4.4 in C++0x mode. More details in Section 5.2.

[4] The graphs in the nde-files are collected by the authors of [2] and can be found on http://diameter.algoritmica.org/networks

cluster of the related master vertex. After that the newly marked vertices are stored in the queue. If two vertices can add the same vertex v to different clusters, the vertex with smaller index marks v as a member of its cluster.[5] After all master vertices have been dequeued, the vertices with distance 1 are dequeued and so on until the queue is empty which means that all vertices have been marked.

Algorithm 5 Parallel clustering code in C++ of the prototype

```
//visit neighbors and build cluster
while (!Q.empty()) do
  //ClusterNode contains the label of the vertex and its references to the adjacency list
  ClusterNode tmpNode = Q.front();
  Q.pop();
  if (tmpNode.begin <= tmpNode.end) then
    unsigned int new_master = whoIsMaster [tmpNode.node];
    unsigned int neighbor_distance = Master_distance[tmpNode.node];
    //consider neighbors of element: if anyone has no cluster => add to cluster of element
    for (unsigned long i=tmpNode.begin; i<=tmpNode.end; i++) do
      unsigned tmp_target = inputEdges[i].target;
      if (Master_distance[tmp_target] == 0 && isMaster[tmp_target] == false) then
        //neighbor gets the same master and distance + 1
        whoIsMaster[inputEdges[i].target] = new_master;
        Master_distance[inputEdges[i].target] = neighbor_distance+1;
        Q.push(inputNodes[inputEdges[i].target]);
      end if
    end for
  end if
end while
```

After each vertex $v \in V$ belongs to a cluster $C(v)$, the distances between the connected clusters are computed. Each neighbor u of v is checked if it belongs to a different cluster $C(u)$. If this is the case the distance between $C(u)$ and $C(v)$ is the distance from u to its master $d(u)$ plus the distance from v to its master $d(v)$ plus one for the edge between u and v. Hence, the distance between $C(u)$ and $C(v)$ is $d(C(u),C(v)) = d(u)+1+d(v)$. After the distances between the clusters are computed, duplicates are removed. Only the connection between two clusters with shortest distance is kept. The related C++ code to the distance calculation is listed in Algorithm 6.

[5] In [6] it was mentioned that an arbitrary master wins if there is a conflict.

Algorithm 6 Calculate the distances between all connected clusters. During the calculation the largest distance of a vertex to its master is stored in max_distance.

```
//go through all vertices V
for (unsigned long i=0; i<inputNodes.size(); i++) do
  //take the biggest distance of the distances between a vertex and its master
  max_distance = std::max(max_distance,Master_distance[i]);
  //consider neighbors of v
  for (unsigned long j=inputNodes[i].begin;j<=inputNodes[i].end;j++) do
    unsigned int neighbor = inputEdges[j].target;
    //if master nodes differ
    if (whoIsMaster[i]!=whoIsMaster[neighbor]) then
      //insert tuple (Master_smaller, Master_bigger, Dist) into vector
      ClusterEdge tmpM_Edge;
      if (whoIsMaster[i]<whoIsMaster[neighbor]) then
        tmpM_Edge.source = whoIsMaster[i];
        tmpM_Edge.target = whoIsMaster[neighbor];
      else
        tmpM_Edge.source = whoIsMaster[neighbor];
        tmpM_Edge.target = whoIsMaster[i];
      end if
      tmpM_Edge.weight = Master_distance[i]+1+Master_distance[neighbor];
      masterDistances.push_back(tmpM_Edge);
    end if
  end for
end for
```

After the clustering and the calculation of the distance the cardinality of the number of vertices is not n but approximately the desired number of master vertices $w = \Theta(n/k)$. The labels of the remaining vertices are mapped into the new interval (1 to $\Theta(w)$) with two scanning and sorting steps. For a more efficient use of memory the edges in Algorithm 6 were only stored in one direction. The other direction to construct an undirected graph is added before doing the SSSP computation, which means that the edge list has to be sorted again. The last step, before the SSSP can be computed, is to build an address list for the adjacency lists for each master vertex. The adjacency list of each vertex v starts in the position where an entry $(v,\star)$ occurs first and ends where $(v,\star)$ occurs last in the sorted edge list.

In a final step the SSSP algorithm is executed with a random start vertex s and then with a second

start vertex s' with farthest distance to s.[6] Because of the fact that the priority queue of the STL (Standard Template Library) in C++ does not support the operation *decrease key* each new shorter tentative distance is inserted into the queue. If the *delete min* operation returns a distance value $d(v)$ that is taller than the current saved tentative (or final) distance for a vertex v, $d(v)$ will be ignored.

To verify the distances, computed by the SSSP algorithm, a method has been written to check the correctness of the output with the following rules (similar to [49]):

1. The source s has distance $d(s) = 0$.
2. $\forall e = \{u,v\} \in E : |d(u) - d(v)| \leq c(e)$. With $c()$ the edge weight is denoted.
3. $\forall u \in V$ with $u \neq s$: There is at least one edge $\{u,v\}$ with v adjacent to u so that the distance $d(u)$ of u is equal to the edge weight $c(\{u,v\})$ plus the distance $d(v)$ of v. This only holds if the graph G is connected. Vertices which are not connected to the subgraph including s will have an infinite distance to s.

The distance $d(s)$ of s is zero by assumption.
For the second rule: if difference of the distance of two adjacent vertices u and v to s is greater than the edge weight $c(\{u,v\})$, the computed final distance of $d(u)$ or $d(v)$ is not as small as possible.

Proof. Without loss of generality let $d(u) < d(v)$. Assume the distances $d(u)$ and $d(v)$ are final and that the distance $d(v)$ of v is greater than $d(u) + c(\{u,v\})$. Then the path from s over u to v with the edge $\{u,v\}$ would result in a smaller path length. That is a contradiction to the assumption that the distances of u and v are final. Therefore, the final distances of u and v cannot differ by more than the edge weight $c(\{u,v\})$. □

While the second rule checks that there is no obvious option to shrink the length of a path by using an edge to a neighbor, the third rule ensures that each vertex $u \in V$ has a neighbor v on the path from s to u with distance $d(u) = d(v) + c(\{u,v\})$. If the shortest path from s to u is the edge $\{s,u\}$ this rule is correct, too. It holds that $d(s) = 0$ and then $d(u) = c(\{u,s\})$. The third rule ensures the existence of a vertex $v \in V$ with $\{u,v\} \in E$ so that the distance of u is the distance of v to the source s (if v is not the source itself) plus the edge weight $c(\{u,v\})$ of the edge $\{u,v\}$.

Although it is very simple to see that these rules are necessary, here is a proof to show that its also sufficient.

Proof. Set $d(s) = 0$. Compute the distances $d(v)$ for all vertices $v \in V$ with $v \neq s$. Also for each vertex v, save its shortest path from s to v. These paths can be saved in a SSSP tree S_T.
Construct S_T in the following way:

[6] Compare description in Section 4.2.

Each vertex v has an entry $(parent\_id, distance)$ in an array *parent*. The entry parent[s] is the pair $(s,0)$ that marks s being the root of S_T. For every other vertex, create an entry in the parent array $(null, \infty)$. During the execution of the SSSP algorithm replace the entry of a vertex v in the parent array with the pair $(u, d(v))$, when the tentative distance $d(v)$ of v is decreased by vertex u with the edge $\{u,v\} \in E$.
After the construction of S_T, it holds for each vertex v that the corresponding entry in *parent* is the pair $(u, d(v))$ where u is the vertex before v on the path from s to v and $d(v)$ is the shortest possible path length from s to v.
The path from s to v can be constructed backwards now via $p_i = parent[v]$ until an i where p_i is $parent[p_i]$ which means that $p_i = s$. The distance between each consecutive pair of vertices (p_i, p_{i+1}) on the path backwards from v to s is $c(\{p_i, p_{i+1}\}) = |d(p_{i+1}) - d(p_i)|$.
That means that there is at least one vertex $u \in V$ such that $d(v) = d(u) + c(\{u,v\})$. This is ensured by rule three.

As a final step it has to be checked if there is no other possible parent p' for each vertex $v \neq s$ which has shorter distance to s. By construction the distance of v to its parent should be the smallest possible. It has to be verified that with testing each other possible parent p'. Only vertices u that are adjacent to v ($\{u,v\} \in E$) could be such a parent p' of v. If there exists such a parent p' the SSSP tree S_T is not correct. If there exists no p' for any vertex v the SSSP tree is correct.
The argument of correctness can be easily proved by induction. If the children of s in S_T have no better parent and if the grandchildren of s have no better parent etc., S_T represents an optimal solution. But this is the case when there exists no parent p' for any vertex v. The second rule ensures that.

The result of the SSSP algorithm as a set of distances is equivalent to the result of the SSSP tree S_T.[7] With the presented set of rules for S_T it is possible to verify if S_T is correct. Therefore these rules are sufficient to verify that the output of an arbitrary SSSP algorithm is correct as long as the weights are not negative. □

4.4 External-memory implementation

For the development of the external-memory application the structure of the prototype was taken as a pattern. For reading the input data existing methods of the BFS implementation by [27] were used. The structure of the input file is a binary edge list. That means that two consecutive entries of x bytes are defined as an edge. The value of x depends on the corresponding type for the vertices. For example if the node type is an integer, then $x = 4$ which means that 8 consecutive bytes are an edge.
The graph is stored in an instance of the graph class which was also used in the implementation by [27]. The edges are stored in ascending order in an adjacency list, the STXXL vector `edgeVec`

[7] Construction: $\forall v \in V$: create a pair $(u, d(v))$ with u being an adjacent vertex with $d(v) = d(u) + c(\{u,v\})$

`*_adj_vec`. To access its adjacency list each vertex needs to know where its edges are stored in this vector. Hence, there is a second STXXL vector `nodeVec *_node_vec` that stores the position where the adjacency list of the corresponding vertex starts. The end of the adjacency list of vertex v_i is the start of the adjacency list of vertex v_{i+1}. For $i = n-1$ the end of its adjacency list is marked by a dummy vertex at the end of the vector.
Thereafter the clustering method is called. In this method the graph will be condensed.

The method for shrinking the graph was implemented by Ajwani et al. earlier. It is based on the preprocessing method for external-memory implementation of BFS [27]. It works as follows: Each vertex is a master vertex with probability $\mu = w/n$, where w is the number of desired master vertices and n the total number of vertices. Each master vertex is assigned now to its own cluster. This is saved in an extra information – a label for the cluster. Thereafter the remaining vertices will be scanned until each vertex, not being a master, has a cluster label. Every time a cluster label is assigned to a vertex, an edge (cluster label source, cluster label destination, distance) is added to the output vector. No deterministic master vertices are selected for the same reasons as explained in Subsection 4.3.2.

The output of the clustering is not sorted, contains duplicates with different weights and edges are not necessarily saved in both directions. Therefore, two scanning and two sorting steps are needed to sort the data, remove duplicates and then ensure that only the smallest edge weight is used. The sorted vector with edges is the edge list for the SSSP.
After creating an input for SSSP there are two choices: use internal memory SSSP like in the prototype or the semi external-memory SSSP by [1]. The SSSP implementation of the prototype has the advantage that it is fast compared to the semi external-memory SSSP. But it is unclear if the condensed graph is small enough to fit into main memory. Therefore, both possibilities have been implemented. If the condensed graph fits into main memory, the internal-memory SSSP is used. Otherwise, the semi external-memory SSSP is used.

If the data does not fit into main memory, it is saved into two files for executing the semi external-memory SSSP. One contains the edges without the weights and the other file contains only the edge weights. The file with the edges will be load by the external-memory BFS code [27] and only the first phase will be executed. In this phase each vertex is assigned to a cluster. The cluster label is saved into a new output file with the edge information.

As a last step the semi external-memory SSSP is executed with the output file from the first phase of external-memory BFS and the edge weight file. The semi external-memory SSSP has been adapted to work with the second file. In the original version the edge weights were generated randomly in an interval $[1,k]$. Now each edge weight has to be assigned to its corresponding edge again. This is done with a few scanning and sorting steps.

The edges are stored in the order of the clusters. To assign the weights to their corresponding edges a temporary vector is created. Each entry of this vector looks as follows: (v_i, v_j, weight, position). First the edges are read. They will be copied with their position in the clustered order. Then the temporary vector is sorted by the first two entries V_i and v_j. The weights are assigned with a scan. After that the temporary vector is sorted by the position. As a last step, the temporary vector is scanned again and the weights are inserted into the vector for weights. The idea is equivalent to the idea of Algorithm 3 in Section 2.7.

The semi external-memory SSSP is executed two times like the internal-memory SSSP with two different sources s and s' with $d(s')$ maximal to s.

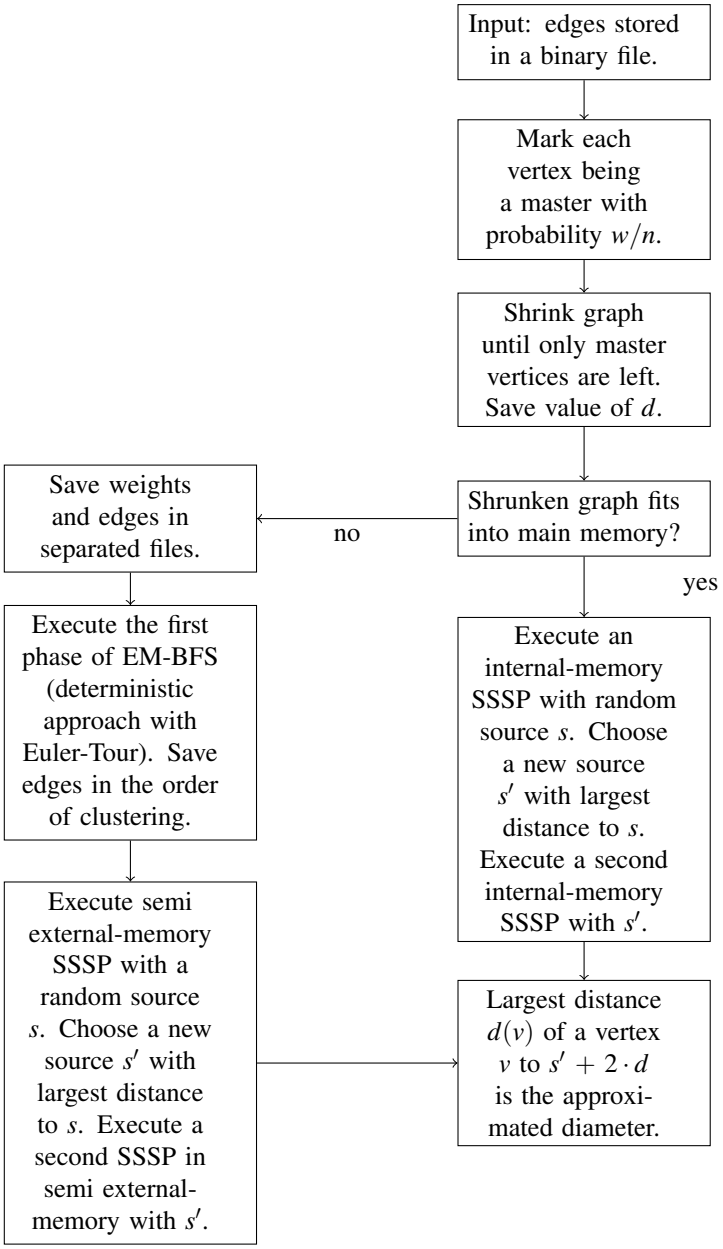

Figure 4.3: Flowchart of the external-memory implementation.

4.5 Recursive approach

Switching between the internal-memory SSSP or the semi external-memory SSSP version is not satisfying because the semi external-memory SSSP increases the I/O complexity much. If it would be possible to shrink the condensed graph again at reasonable cost, the new approach could be applied recursively. Unfortunately, the condensed graph has weighted edges. Considering the edge weights for clustering would lead to a similar problem as using BFS for weighted graphs instead of SSSP (see Section 2.12). The benefit of the recursive approach is that the sum of the I/O complexity of all recursion steps is smaller than the I/O compliexty of one huge clustering step. The I/O complexity of a clustering step is related to the size of d. $\Theta(d)$ sorting and scanning steps have to be done. During the clustering, the data is shrunken. Therefore, the next clustering step will be computed faster and so forth. The I/O complexity can be reduced to $\Theta(\log(B) \cdot (scan(n) + sort(n)))$ with recursion.
A way to construct a recursive approach, is to ignore the edge weights while shrinking the graph again. Select about w' master vertices, where w' denotes the desired number of master vertices for the next step, and apply the clustering like before. The distance of a new cluster member v reached by u is not $d(v) = d(u) + 1$ any more but $d(v) = d(u) + c(\{u,v\})$. The same rule is applied for the distance between the clusters: $d(\{C(u),C(v)\}) = d(u) + c(\{u,v\}) + d(v)$.

As a proof of concept, the prototype of the internal-memory implementation was adapted first. After the first results were promising, the external-memory BFS preprocessing was adapted so that it can shrink a graph with arbitrary edge weights. Thereafter, the external-memory approach was improved to a 2-step approach. To this end, a method for building a new graph instance out of the clustering output was written. After the clustering is done, the two scanning and sorting steps are computed to organize the output of the clustering again as after the first clustering step (compare 4.4). The rest of the code is similar to the single step external-memory approach.

While the implementation of the recursive approach itself was not that difficult, identifying a non-trivial upper bound for the expected error for this approach was difficult. It is easy to see that in worst case the expected error for this approach is $O(k)$ for the second step. The expected error of the first step stays $O(\sqrt{k})$ like before.

Before the proof of a lower bound for the expected error of the recursive approach is discussed in more detail, first a proof of an upper bound for the expected error of a weighted graph as input for the first phase is presented.

Proof. Assume a graph G being a linked list of the length k. All edges have the weight 1, except those edges at the beginning and at the end of the list, which have weight k. Now add to each vertex of the list except at the beginning and the end of the list a new vertex and an edge with an edge weight k (see Figure 4.4). The diameter of G is now $2 \cdot k + k - 2 = 3 \cdot k - 2$.

If such a graph is shrunk with around a half of the vertices being selected as a master vertex, the diameter of G will be $O(k^2)$ with high probability. This is because a constant fraction of the added vertices will be selected as master vertices with high probability and then the distance to other clusters could be $\Theta(k)$ with high probability. The expected multiplicative error is $\Theta(k)$ instead of $\Theta(\sqrt{k})$ for this example. Figure 4.5 visualizes such a case. □

Figure 4.4: A weighted graph as an input where the clustering approach produces an error of $\Theta(k)$.

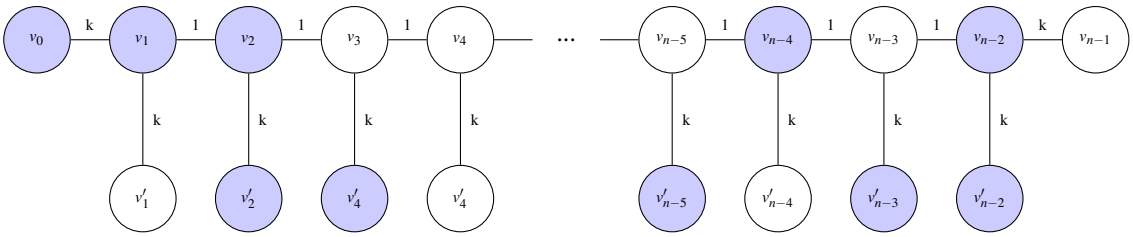

Figure 4.5: The colored vertices are selected as master vertices. A fraction of the vertices v'_i connected to the list are selected as master vertices. If the related vertex v_i in the list is not selected as a master vertex, the master vertex at v'_i blows up the diameter by the edge weight k.

For a lower bound of the worst case scenario for the recursive approach the fact that the expected error for the first iteration is bounded by $O(\sqrt{k})$ has to be regarded. Therefore a graph structure has to be found that is bad in the first iteration and also in the second. The best non-trivial lower bound for two consecutive clustering steps which was found so far is $O(k^{4/3-\varepsilon})$ with high probability.

Theorem: A lower bound for two consecutive clustering steps is $O(k^{4/3-\varepsilon})$ with high probability.

Proof. The graph construction for the lower bound is based on a few subgraphs which will be reused several times.

- **x-fan**: a vertex v is connected to x vertices, so that $\{v, v_i\} \in E$ with $0 \leq i \leq x-1$. These x vertices are only connected to v. See Figure 4.6.

- **x-doublefan**: similar to the x-fan. But now each vertex v_i with $0 \leq i \leq x-1$ is also connected to another vertex w_i and each w_i is connected to a set of vertices z_{ij} with $0 \leq j \leq x-1$. See Figure 4.7.

- **side chain block**: the vertex u in the middle of the horizontal line is connected to a list of $k^{1/3}$ vertices v_i to the right and to the left. Each v_i has x children (x-fan). Vertex u is also connected to another list of length $k^{5/3-\varepsilon}$. The last vertex of the latter list is the root v' of an x-doublefan. See Figure 4.9.
- **basic block**: The basic block is constructed via connecting $k^{1/3}$ side chain blocks. The first side chain block is also connected to a list of $k^{2/3}$ x-fans to the left. The end of this list is connected to an x-doublefan. The last side chain block on the right is similar. See Figure 4.10.
- $\Omega(x)$**-fan**: The $\Omega(x)$-fan is the x-doublefan after it was shrunken in the first phase. Although the structure of this subgraph is randomly determined, it will be similar to an x-fan with high probability. See Figure 4.8 for a possible shape of the $\Omega(x)$-fan.

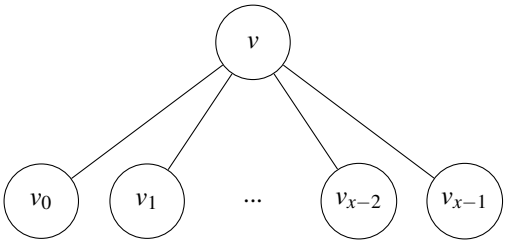

Figure 4.6: The x-fan.

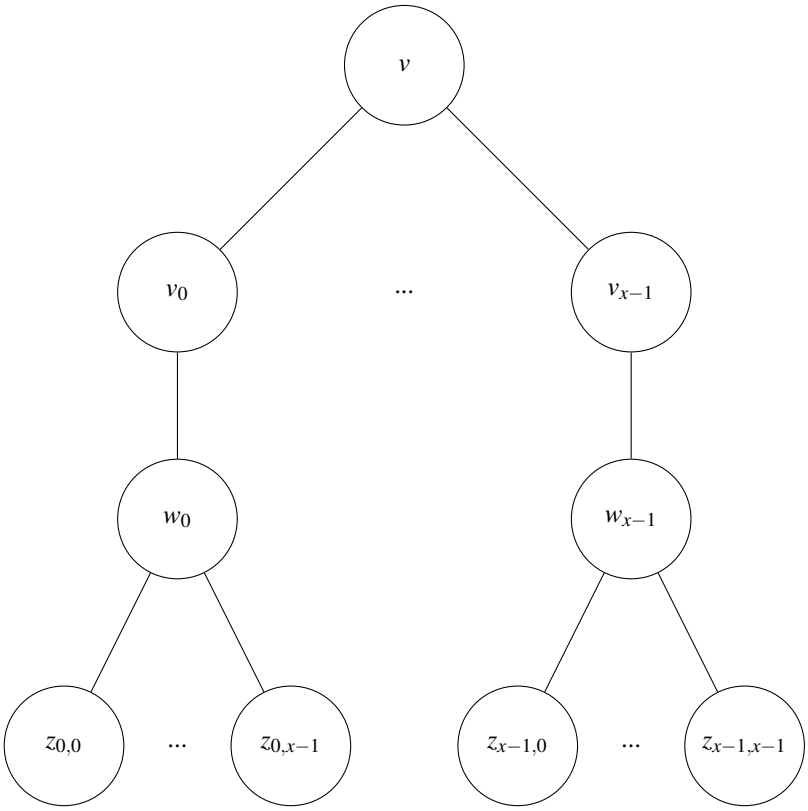

Figure 4.7: The x-doublefan.

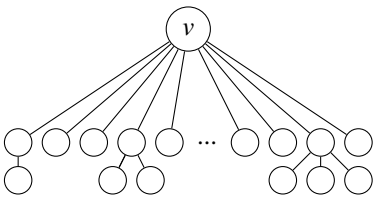

Figure 4.8: A possible shape for the $\Omega(x)$-fan. The shape of the $\Omega(x)$-fan is not fixed.

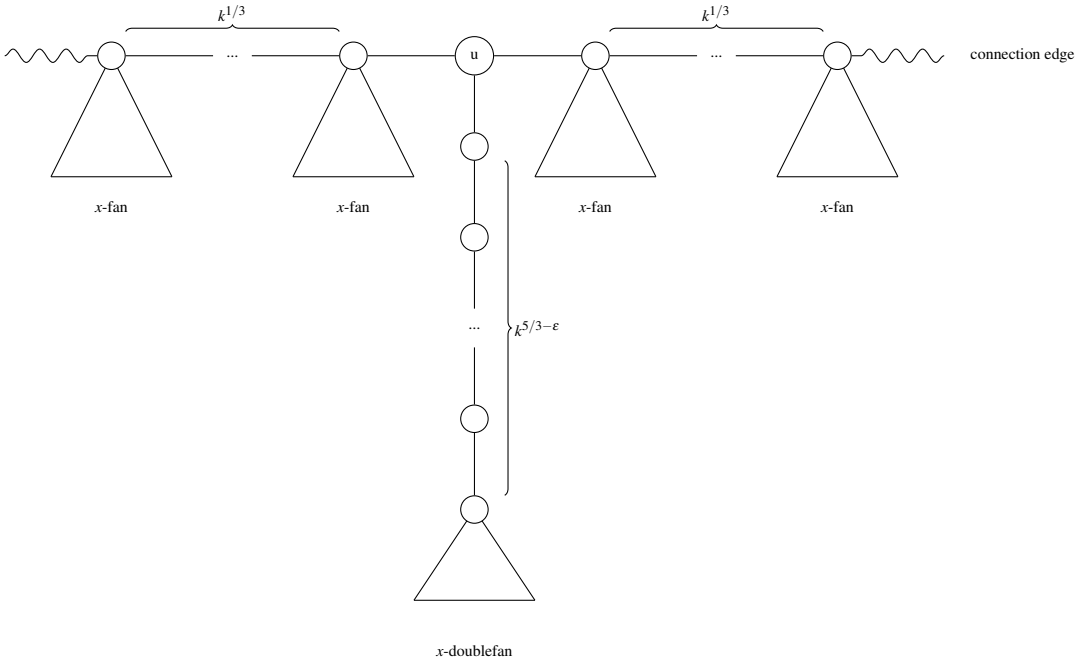

Figure 4.9: A side chain block.

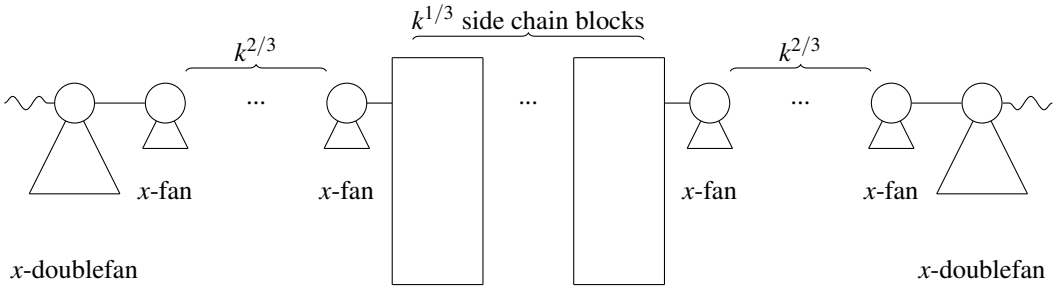

Figure 4.10: A basic block.

The input graph G is obtained by connecting y basic blocks for global diameter $O(y \cdot k^{2/3} + k^{5/3-\varepsilon})$. For sufficiently large y the first term dominates. Let x be $\Omega(k \cdot \log(n))$. Choose master vertices with probability $1/k$ for the first round. Note that an x-doublefan will turn into an $\Omega(x)$-fan with probability

$p_{xd} \geq 1 - P(\text{“root of the } x\text{-doublefan is selected as a master”}) = 1 - \frac{1}{k} = \frac{k-1}{k}$. For sufficiently large k the probability that the root of an x-doublefan is selected as a master is almost zero.
After the first shrinking with high probability each basic block will have the following shape:

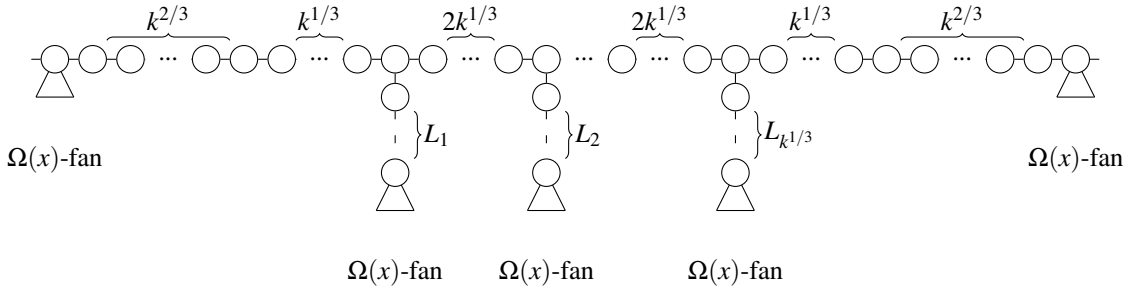

Figure 4.11: The shape of the graph after the first iteration.

Each chain L_j $(j = 1,2,...,k^{1/3})$ has an accumulated weight of $k^{5/3-\varepsilon}$. The average number of vertices for any L_j is $\mu = k^{5/3-\varepsilon} \cdot 1/k = k^{2/3-\varepsilon}$. By Chernoff bounds, the actual number of vertices on a chain L_j is outside the range $k^{2/3-\varepsilon} \pm k^{1/3}$ with a probability of at most $2 \cdot e^{-k^{1-\varepsilon}/3}$ with $\varepsilon < 1$.

Lemma: The length of any chain L_j $(j = 1,...,k^{1/3})$ is outside the range $k^{2/3-\varepsilon} \pm k^{1/3}$ with probability $2 \cdot e^{-k^{1-\varepsilon}/3}$.

Proof. The random variables X_i will denote the vertices in a chain. The variance is denoted by δ.

Case $\delta = +k^{1/3}$:

It holds that $\mathrm{P}\left[\sum_{i=1}^{n} X_i \geq (1+\delta)\cdot\mu\right] \leq e^{-\frac{min\{\delta,\delta^2\}}{3}\cdot\mu}$ (compare Section 2.14).

$$\mathrm{P}\left[\sum_{i=1}^{k^{5/3-\varepsilon}} X_i \geq (1+k^{1/3})\cdot(k^{5/3-\varepsilon}\cdot\tfrac{1}{k})\right] \leq e^{-\frac{min\{k^{1/3},k^{2/3}\}}{3}\cdot(k^{5/3-\varepsilon}\cdot\frac{1}{k})}$$

$$\mathrm{P}\left[\sum_{i=1}^{k^{5/3-\varepsilon}} X_i \geq (1+k^{1/3})\cdot k^{2/3-\varepsilon}\right] \leq e^{-\frac{k^{1/3}}{3}\cdot k^{2/3-\varepsilon}} = e^{-\frac{k^{1-\varepsilon}}{3}}$$

Case $\delta = -k^{1/3}$:

It holds that $\mathrm{P}\left[\sum_{i=1}^{n} X_i \leq (1-\delta)\cdot\mu\right] \leq e^{-\frac{\delta^2}{2}\cdot\mu}$.

$$\mathrm{P}\left[\sum_{i=1}^{k^{5/3-\varepsilon}} X_i \leq (1-k^{1/3})\cdot(k^{5/3-\varepsilon}\cdot\tfrac{1}{k})\right] \leq e^{-\frac{k^{2/3}}{2}\cdot(k^{5/3-\varepsilon}\cdot\frac{1}{k})}$$

$$\mathrm{P}\left[\sum_{i=1}^{k^{5/3-\varepsilon}} X_i \leq (1-k^{1/3})\cdot k^{2/3-\varepsilon}\right] \leq e^{-\frac{k^{2/3}}{2}\cdot k^{2/3-\varepsilon}} = e^{-\frac{k^{4/3-\varepsilon}}{2}} \leq e^{-\frac{k^{1-\varepsilon}}{3}} \qquad \square$$

Therefore with a probability $p \geq 1-k^{1/3}\cdot 2e^{-k^{1-\varepsilon}/3} - O(\frac{1}{k^{\varepsilon}}) \geq c$ for sufficiently large k and a constant c. The term $O(\frac{1}{k^{\varepsilon}})$ stands for the probability

$$p_{ch} = 1 - \underbrace{(1-\frac{1}{k})^{k^{1-\varepsilon}}}_{\text{No master appears in a chain}} \geq 1-(1-\frac{1}{k}\cdot k^{1-\varepsilon}) = 1-(1-k^{-\varepsilon}) = \frac{1}{k^{\varepsilon}}$$

that a master appears on any of the chains L_j, too. These conditions will hold simultaneously and masters for the second phase only appear in the fans with a high probability.
But then clusters for L_j grow towards the horizontal baseline and only meet there in the buffer spaces of $2k^{1/3}$ vertices between them. See Figure 4.12 for an example.

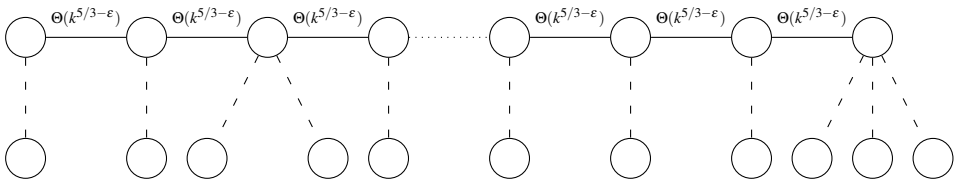

Figure 4.12: A possible shape for the graph after the second clustering. The largest path is determined by the horizontal baseline.

Thus, each basic block can produce a path weight of $\Omega(k^{1/3}\cdot k^{5/3-\varepsilon}) = \Omega(k^{2-\varepsilon})$, with constant probability, essentially independent of the other basic blocks because of the separation by the x-doublefans at the connection points.
Therefore, the expected approximated diameter of the graph after the second shrinking is $\mathrm{E}[diam(G'')] = \Omega(y\cdot k^{2-\varepsilon})$ while the actual diameter of G is $y\cdot k^{2/3}$ for sufficiently large y with $y > k$.
The diameter has been blown up by a factor of $k^{4/3-\varepsilon}$ which is worse than $(\sqrt{k})^2$ if the 2 phases were just independent.
Remark: The proof does not yet consider the effects of deterministic masters. On the other hand, since G is a tree, the positions of deterministic masters are rather predictable, at least for the first clustering. Therefore the shape of the graph after the first phase will be similar to Figure 4.11. However, it is still unclear how strong the shape after the second phase will differ from the random case. $\square$

A trivial upper bound would be $O(\sqrt{k}\cdot k)$. The edge weights cannot be taller than $O(k)$ after the first step because each vertex has at most distance of k to a master vertex. And from the unweighted scenario it is known that the expected multiplicative error is at most $(\sqrt{k})$. Combining both facts leads to the trivial upper bound.
It is an open question if this lower bound is tight. The problem is to find a structure which is bad in

both steps. The distances between master vertices cannot be arbitrarily large with high probability. Furthermore, the expected error for the first phase is bounded by $O(\sqrt{k})$.
Nevertheless the experiments have shown that the error for the 2-step approach is not that bad as in theory. Details will be shown in Section 5.3.7.

Chapter 5

Experiments and Results

In this chapter the results of the experiments with the implementation of the "Parallel clustering growing approach" (short: PAR_APPROX) and the recursive one (short: PAR_APPROX_R) will be presented. Also, PAR_APPROX and PAR_APPROX_R are compared to the existing approaches as a study: the external-memory BFS implementation (short: EM_BFS_DSLB) [27], the external-memory MST based heuristic (short: SPAN) which was presented in Section 3.4 and the Fringe algorithm (short: DSLB_UP_BOUND) [2] (refer to Section 3.3).
The various data sets, which were used for experiments, are introduced in Section 5.1. Subsequently the configuration of the machines which have been utilized will be described. During the presentation of the results, some observations will be reported, which were made during the initial tests. Based on these observations, PAR_APPROX_R has been developed.

A subset of the findings were also presented in "I/O-efficient approximation of graph diameters by parallel cluster growing – a first experimental study.". This publication was written by Ajwani, Beckmann, Meyer and Veith at the end of the year 2011 [7]. It was accepted at "PASA 2012" (10^{th} Workshop on Parallel Systems and Algorithms).

5.1 Graph classes

Four different graphs were chosen to create four different test scenarios:

- sk-2005: This graph is a real world graph and belongs to the class of web graphs. Web graphs typically have a small diameter. This web graph was used as input for the Fringe heuristic in [2]. It has 50,634,118 vertices, 1,810,050,743 edges and the diameter is 40. The web graph sk-2005 was chosen as input because real world graphs typically have different structures compared to synthetic graphs. Also, sk-2005 was the biggest data set used for tests in [2] and sk-2005 which is larger than the main memory of a typical workstation.

- $\sqrt{n}$-level graph: This graph is a synthetic graph. It has 268,435,456 vertices, 1,127,310,556

edges and the diameter is 16,385. It was generated to check the behavior of different approaches for graphs with diameter $\Theta(\sqrt{n})$.

- $\Theta(n)$-level graph: This graph is also synthetically. It has 268,435,456 vertices, 903,876,452 edges and the diameter is 67,108,864. The behavior of different approaches to large diameter will be checked with this graph.

- worst_PAR_APPROX: It is expected that this graph will elicit a bad approximation with the approach PAR_APPROX. It has 268,435,420 vertices, 268,435,419 edges and the diameter is 2,440,341.

While sk-2005 is based on real data[1], $\sqrt{n}$-level graph, $\Theta(n)$-level graph and worst_PAR_APPROX are synthetic. $\sqrt{n}$-level graph and $\Theta(n)$-level graph were generated with the same generator, only the parameters differed. The EM_BFS_DSLB implementation by [27] contains a set of various graph generators.[2] This set of generators also contains a so-called "k-level" generator which generates a graph similar to a tree with k-levels. The diameter of such a k-level graph is $O(k)$. The output of the k-level graph generator in the EM_BFS_DSLB code often generates a disconnected graph. Usually a few vertices are singletons because edges are generated randomly. With a small probability vertices will not be part of any generated edge. A possible shape of a k-level graph is shown in Figure 5.1.

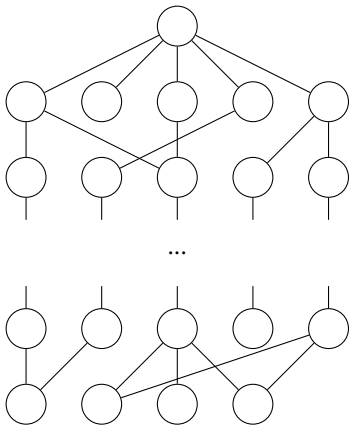

Figure 5.1: A possible shape of the k-level graph. The number of vertices in the last level can differ from the number of vertices in the previous levels.

SPAN does not work on disconnected input graphs. Therefore an additional generator has been written to generate connected k-level graphs. This generator works in the the same way as the generator in the EM_BFS_DSLB implementation, but it ensures that each vertex is connected using a simple trick: each vertex of the subsequent level x is connected to an arbitrary vertex in the previous level $x-1$. After the connection is ensured some random edges might be added between these levels. The

[1] See http://law.dsi.unimi.it/webdata/sk-2005 for further information.

[2] During the work on this thesis the code basis of the generators used by EM_BFS_DSLB was improved. Also a new method was added to read existing graphs and the argument management was improved so that the graph does not have to be in a special folder.

number of random edges depends on computed parameters.
After the generation of the graph the indices of the vertices are permuted.[3] The generator has some special options. One of them is to connect vertices inside the same level. For the $\sqrt{n}$-level graph this option has been used. Two vertices in the same level with adjacent indices were connected with a probability of 0.2.[4] For $\Theta(n)$-level graph this option was not used.
The synthetic graphs $\sqrt{n}$-level and $\Theta(n)$-level are similar to the B–level random graphs in [26].

The graph worst_PAR_APPROX was generated with a new generator, which was added to the set of graph generators in the external-memory BFS implementation. The graph structure was defined to produce a worst case scenario for PAR_APPROX. An input for PAR_APPROX is difficult if many master vertices can reach the diameter determining path P_d. If many master vertices can reach the path P_d, this will blow up the path length and therefore the diameter very much.
To amplify this effect a special graph structure was created (Figure 5.2). It might not be the worst possible case but it generates bad results:

- The graph has three parameters: k_1, k_2 and k_3 satisfying $n = k_3 \cdot (k_1 + k_2)$.
- $k_3 = \left\lfloor \frac{n}{k_1+k_2} \right\rfloor$ is the length of the main chain C_0.
- k_1 is the length of the side chains C_i with $1 \leq i \leq k_3$. Each chain is connected to one vertex of C_0 and on the other end there is a K_2-fan.
- k_2 is the size of the fan-out at the end of the side chain C_i.
- The diameter is $k_1 + k_3 + k_1 - 1 = 2 \cdot k_1 + k_3 - 1$.
- In worst case the diameter can be blown up to $O(k_1 \cdot k_3)$ because the path P_d is connected to $\Theta(k_3)$ side chains of length k_1.

[3] For this a lookup table is computed which contains the new index for each vertex. The new index is computed through generating a random number for each vertex. The vertices are sorted by their random number and then the position of the vertex in this sorted sequence is the new index.

[4] This option was used for the $\sqrt{n}$-level graph to study the influence of connections inside a level. Later tests with different probabilities (e. g. 0 and 0.5) showed that this does not make a big difference for the behavior of PAR_APPROX. It only influenced the number of generated edges by a little. The diameter of the graphs was similar.

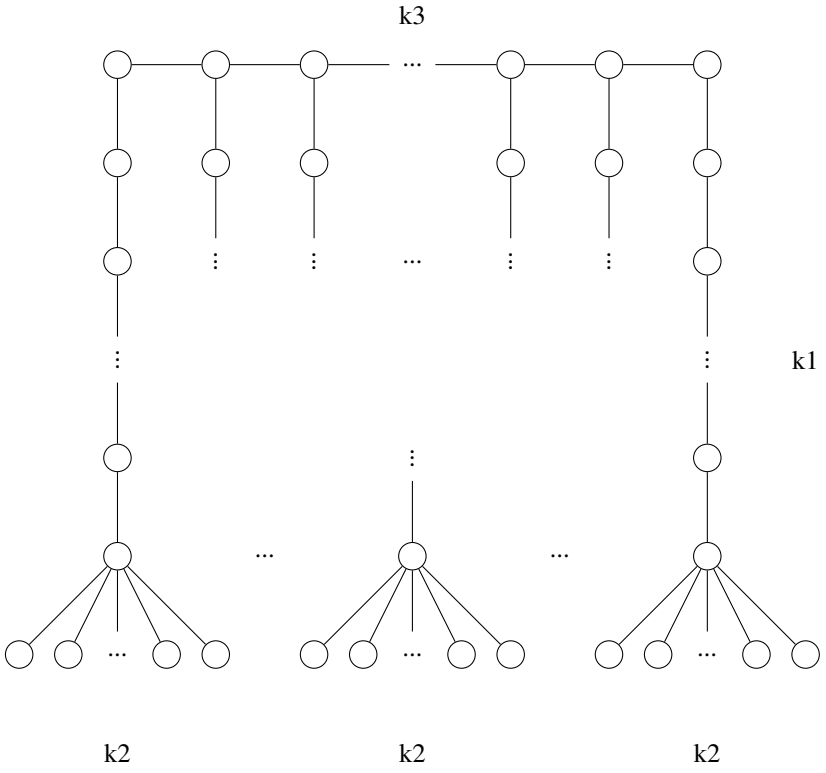

Figure 5.2: Sketch of the worst case graph for PAR_APPROX.

Note, that for fixed values of the parameters k_1, k_2 and k_3 the behavior of PAR_APPROX cannot be worse in each case. There is a relationship of the behavior to worst_PAR_APPROX in the master-probability. The parameters of worst_PAR_APPROX have the values $n = 268,435,420$, $k_1 = 10$, $k_2 = 100$ and $k_3 = 2,440,322$. These values are arbitrarily chosen. However, a small k_1 ensures that the worst case can also be occurred for a huge number of master vertices.

5.2 Configuration

The experiments were performed on two different architectures:

1. As architecture for the external-memory experiments a relatively cheap workstation was used with an Intel dual core E6750 processor @ 2.66 GHz, 4 Gigabyte main memory (around 3.5 Gigabyte was available for the application) and four hard disks with 500GB in a RAID-0. From each of these four disks only the 250GB from the outer tracks were used as STXXL hard disks (as four parallel disks). Hence, only around one Terabyte was available as external-memory for STXXL. The inner tracks were used as storage for graph files, log files etc. The reason for this is that the data transfer rate of the disk is larger in the outer tracks.
   The operation system was Debian GNU/Linux amd64 ‘wheezy’ (testing) with kernel 3.0. The code was compiled using GCC 4.4 in C++0x mode with optimization level 3 was used.

2. For applications which were designed for main memory usage, a machine (Ino) from the HPC (High Performance Computing) cluster at Goethe Universität was used. The machine hosts four quad-core AMD Opteron™ processor 8384 @ 2.7 GHz, but only one core has been used at all. The internal-memory prototype of PAR_APPROX ran on this machine and also DSLB_UP_BOUND. As the running time of an approach on this machine is no indication of its respective running time in an external-memory setting these times are not mentioned in the results.

5.3 Results

5.3.1 EM_BFS_DSLB, SPAN and DSLB_UP_BOUND

The implementation of EM_BFS_DSLB was modified to use double sweep lower bound. For that reason the second phase of the implementation was executed twice. DSLB_UP_BOUND was executed with the default settings (one experiment with ten iterations, double sweep lower bound and Fringe). The result of DSLB_UP_BOUND is the exact diameter for the tested graphs classes (see Table 5.1). That means that the computed lower bound is equal to the computed upper bound. For SPAN two iterations were executed.

	EM_BFS_DSLB	SPAN	DSLB_UP_BOUND
sk-2005	39	60	**40**
$\sqrt{n}$-level graph	**16,385**	46,262	**16,385**
$\Theta(n)$-level graph	**67,108,864**	86,488,096	**67,108,864**
worst_PAR_APPROX	**2,440,341**	3,982,472	**2,440,341**

Table 5.1: Diameters computed by EM_BFS_DSLB, DSLB_UP_BOUND and SPAN. Exact diameters are marked in boldface.

For sk-2005 EM_BFS_DSLB did not compute the exact diameter of 40 with source 0 as first source and 519,830 as source for the second BFS computation. With the sources 1,000,000 and 16,307,150 the exact diameter was found but this result was not used for Table 5.1, because in practice the user would usually not execute it with a special source. This was only done to check that the diameter of 40 can be found and if it makes a difference in the execution time. The difference was only a few minutes.

	EM_BFS_DSLB	SPAN
sk-2005	5.27	7.65
$\sqrt{n}$-level graph	10.64	7.74
$\Theta(n)$-level graph	4.75	4.81
worst_PAR_APPROX	1.66	3.34

Table 5.2: Running times (in hours) of EM_BFS_DSLB and SPAN.

The times of SPAN in Table 5.2 can be improved by using a bucket based MST algorithm in the

preprocessing instead of the MST algorithm using a priority queue. Also the results are closer to the actual diameter.[5]

The results of EM_BFS_DSLB and DSLB_UP_BOUND are close to the actual diameter or in many test cases the exact diameter. As mentioned in Section 3.3 an external-memory approach similar to DSLB_UP_BOUND is too costly because many BFS computations would be needed to get the exact diameter. Even if the number of Breadth-First Searches could be reduced to a small fraction of the currently executed number of Breadth-First Searches it would be too costly. The target to select the best algorithm for EM_BFS_DSLB would be lost and also no speedup would be gained which was the second purpose of PAR_APPROX[6].

5.3.2 Internal-memory prototype of PAR_APPROX

As mentioned before the internal-memory prototype of PAR_APPROX was developed to have a look at the behavior of the new approach without the restrictions that can show up during external-memory experiments - especially the larger execution time.
The results in the Tables 5.3, 5.4, 5.5 and 5.6 are showing that the expectation that the diameter is decreasing with a growing number of master vertices is almost true. For sk-2005 the resulting diameter fluctuates much. One reason for this behavior may be that d is very close the half of the actual diameter in many cases until the number of master vertices is 16777216. Another reason is that many master vertices reach the same path and thus blow up the length of this path.

For the graphs $\sqrt{n}$-level graph and $\Theta(n)$-level graph the results are also promising for a small fraction of vertices as masters. The ratio between the approximated diameter and the actual diameter is less or equal to 1.5. For 1024 and more master vertices the ratio between the approximated and the actual diameter is around 1.02.

For worst_PAR_APPROX the results show that there are cases in which the results are close to the actual diameter but there also is an interval where the approximated diameter is growing and it seems that between $1,048,576$ and $4,194,304$ a maximum ϕ can be found. The occurred error in between is as big as possible. At ϕ the computed diameter is in the order of $O(k_1 \cdot k_3)$.

[5] These results are not presented in this thesis in more detail because the improvement of SPAN is a current research topic.

[6] compare Section 1.1

number of masters	d	**computed diameter**
2	23	53
4	21	49
8	21	53
16	18	57
32	20	56
64	20	54
128	20	55
256	20	59
512	20	66
1024	20	69
2048	20	74
4096	20	75
8192	20	75
16384	20	82
32768	20	89
65536	20	95
131072	20	93
262144	20	111
524288	20	103
1048576	20	104
2097152	20	117
4194304	19	125
8388608	19	119
16777216	13	101
33554432	7	61
50634118	0	39

Table 5.3: Results for sk-2005. The resulting diameter is fluctuating much. Either for a few or many master vertices the results are close.

number of masters	d	**computed diameter**
2	11703	24611
4	5310	18218
8	3158	17792
16	2037	16671
32	1376	16456
64	467	17114
128	274	16771
256	149	16711
512	119	16720
1024	69	16711
2048	43	16662
4096	21	16480
8192	10	16403
16384	8	16399
32768	8	16400
65536	7	16398
131072	7	16398
262144	7	16398
524288	6	16396
1048576	6	16397
2097152	5	16396
4194304	5	16396
8388608	5	16397
16777216	5	16397
33554432	4	16395
67108864	4	16395
134217728	3	16393
268435456	0	16385

Table 5.4: Results for $\sqrt{n}$-level graph. The resulting diameter is close to the actual diameter.

number of masters	d	**computed diameter**
2	26415603	80048947
4	26415603	93205163
8	8827267	75616827
16	8827267	75616827
32	7141510	67359763
64	1912297	69979907
128	2079249	68783184
256	895936	68594997
512	415970	67864213
1024	228249	67546538
2048	149286	67388623
4096	77638	67255072
8192	33069	67167702
16384	18401	67138713
32768	11980	67128199
65536	6900	67121749
131072	3103	67116317
262144	1698	67115306
524288	897	67118032
1048576	439	67127687
2097152	253	67156088
4194304	121	67233693
8388608	73	67419470
16777216	31	67718582
33554432	17	67848160
67108864	8	67515774
134217728	3	67168412
268435456	0	67108864

Table 5.5: Results for $\Theta(n)$-level graph. The resulting diameter is close to the actual diameter in many cases.

number of masters	*d*	**computed diameter**
2	1528273	3686550
4	421159	2727959
8	315012	2515745
16	282092	2464484
32	125644	2560355
64	125644	2560935
128	47005	2498841
256	23544	2489948
512	13907	2475624
1024	7626	2472742
2048	4277	2485446
4096	3157	2521684
8192	1332	2596264
16384	900	2750204
32768	476	3052952
65536	249	3639200
131072	124	4746736
262144	78	6726590
524288	50	9822432
1048576	38	13436852
2097152	28	14945315
4194304	24	11316021
8388608	22	6071011
16777216	21	4399106
33554432	11	4108649
67108864	8	3770069
134217728	7	3269382
268435420	0	2440341

Table 5.6: Results for worst_PAR_APPROX. The resulting diameter is very close in many tested cases. But there are cases in which the resulting diameter is growing until it reaches a maximum. After this maximum the resulting diameter is shrinking again close to the actual diameter.

5.3.3 PAR_APPROX with internal-memory SSSP

In this subsection the results from PAR_APPROX with internal-memory SSSP will be presented. The results from sk-2005 (see Table 5.7) are promising. The execution time is increasing with increasing number of master vertices because the preprocessing step of the internal-memory SSSP needs more time with an increasing number of vertices. Nevertheless, also the slowest case needed 47 Minutes which is very fast compared to the 5.27 hours of EM_BFS_DSLB (see Table 5.2). Only considering this case, PAR_APPROX seems to be a good preprocessing step for selecting the right external-memory BFS algorithm.
However, the results of the experiments for $\sqrt{n}$-level graph (Table 5.8), $\Theta(n)$-level graph (Table 5.9) and worst_PAR_APPROX (Table 5.10) are showing a different behavior. The time for cases with a small fraction of vertices being masters is even taller than the execution time of EM_BFS_DSLB. Using the $\Theta(n)$-level graph for a tall number of master vertices the execution time reaches an acceptably small duration of 1.26 hours for $16,774,408$ master vertices. For the $\sqrt{n}$-level graph it was not possible to use more than around $O(2^{18})$ master vertices with internal-memory SSSP. But then PAR_APPROX would need about two days to compute the diameter of the $\Theta(n)$-level graph. This reasoned by the fact that d is very large for the $\Theta(n)$-level graph if only a small fraction of the vertices is chosen as a master and then the I/O complexity of the condensing the graph is very huge as described in Subsection 4.1.2.
A solution would be to tune the parameters that are related to the diameter for each graph class. But this is a contradiction. The tuning can only be done if the diameter is known. Therefore another approach has been tried: PAR_APPROX with semi external-memory SSSP. If the input is too big that the internal-memory SSSP can be used, the semi external-memory SSSP is used instead. The results are presented in Subsection 5.3.4.

masters	diameter	ratio	time[h]	d
233	42	1.05	0.46	12
987	51	1.28	0.51	13
4,103	68	1.70	0.62	17
16,385	79	1.98	0.68	17
65,457	106	2.65	0.73	17
261,723	113	2.83	0.76	22
1,047,527	98	2.45	0.78	18
4,193,085	119	2.98	0.77	16

Table 5.7: Results for sk-2005.

masters	diameter	ratio	time[h]	d
1,020	16,836	1.0275	4.29	156
4,051	16,519	1.0082	1.30	35
16,511	16,413	1.0017	0.87	15
65,794	16,409	1.0015	0.87	13
262,215	16,408	1.0014	0.85	12

Table 5.8: Results for $\sqrt{n}$-level graph.

masters	diameter	ratio	time[h]	d
262,215	67,118,479	1.00014	41.60	3,444
1,048,506	67,128,342	1.00029	12.33	814
4,195,701	67,233,297	1.00185	3.68	233
16,774,408	67,717,702	1.00907	1.26	60

Table 5.9: Results for $\Theta(n)$-level graph.

masters	diameter	ratio	time[h]	d
65,794	3,643,615	1.49	5.12	449
262,215	6,729,783	2.76	1.87	144
1,048,506	13,461,919	5.52	0.92	50
4,195,700	11,265,297	4.62	0.73	28
16,774,407	4,399,657	1.80	0.58	22
67,105,245	3,774,597	1.55	0.39	17

Table 5.10: Results for worst_PAR_APPROX.

5.3.4 PAR_APPROX with semi external-memory SSSP

As mentioned in Subsection 5.3.3 the input for the SSSP phase can be too big for the internal-memory SSSP. So the semi external-SSSP is used instead. The semi external-memory SSSP was tested for three graphs: sk-2005, $\sqrt{n}$-level graph and $\Theta(n)$-level graph.
In fact, the duration is lower but not as much in order to use this approach as an additional preprocessing step for EM_BFS_DSLB. Nevertheless, in order to compute the diameter it is faster to use PAR_APPROX with semi external-memory SSSP than EM_BFS_DSLB (Table 5.11). However, the gain is so small that the recursive approach was tried to lower the computation time.

graph	sk-2005	sk-2005	$\sqrt{n}$-level	$\sqrt{n}$-level	$\sqrt{n}$-level	$\Theta(n)$-level	$\Theta(n)$-level	$\Theta(n)$-level
masters	$\sim 2^{22}$	$\sim 2^{24}$	$\sim 2^{22}$	$\sim 2^{24}$	$\sim 2^{26}$	$\sim 2^{22}$	$\sim 2^{24}$	$\sim 2^{26}$
diameter	119	90	16,407	16,403	16,401	67,233,297	67,717,702	67,515826
ratio	2.98	2.25	1.0013	1.0011	1.0010	1.00185	1.00907	1.00606
time[h]	1.09(0.32)	2.78(1.73)	6.96 (5.51)	8.38 (6.80)	9.41 (7.72)	3.94 (0.06)	1.6 (0.29)	2.47 (1.68)
d	16	13	10	8	7	233	60	16

Table 5.11: Results for sk-2005, $\sqrt{n}$-level graph and $\Theta(n)$-level graph with PAR_APPROX with semi external-memory SSSP.

5.3.5 Internal-memory prototype of PAR_APPROX_R

Similar to the implementation of PAR_APPROX a prototype has been developed for PAR_APPROX_R. This prototype had the task to check if the recursive approach is viable and if the results are promising enough to implement an external-memory approach. The proof in Section 4.5 showed that the expected multiplicative error of the recursive approach can be worse than $O(k)$ / $O(d)$. Nevertheless the results of the recursive approach are not as bad as expected. Only for sk-2005 (Table 5.12) the resulting diameter is off by a factor of 5.4 in one case while for the graphs $\sqrt{n}$-level graph (Table 5.13) and $\Theta(n)$-level graph (Table 5.14) promising results have been reached. Also for worst_PAR_APPROX (Table 5.15) the results varied only by a small amount compared to PAR_APPROX.

In the tables the number of master vertices in the first and in the second step are mentioned and also the the tallest distance of a vertex to a master vertex in the second step as d_2.

#masters 1st step	#masters 2nd step	d_2	diameter	difference
2	2	0	53(53)	0
4	2	5	59(49)	10
8	2	5	61(53)	8
16	6	9	61(57)	4
32	12	8	73(56)	17
64	25	13	86(54)	32
128	41	7	69(55)	14
256	71	10	76(59)	17
512	128	10	86(66)	20
1024	262	16	115(69)	46
2048	509	20	130(74)	56
4096	997	19	117(75)	42
8192	2037	22	132(75)	57
16384	4091	20	127(82)	45
32768	8251	30	179(89)	90
65536	16423	75	215(95)	120
131072	32849	26	180(93)	87
262144	65812	25	180(111)	69
524288	131267	24	192(103)	89
1048576	262364	27	201(104)	97
2097152	524179	26	187(117)	70
4194304	1047326	29	192(125)	67
8388608	2094977	21	202(119)	83
16777216	4192099	20	194(101)	93
33554432	8384647	18	154(61)	93
50634118	12657770	15	107(39)	68

Table 5.12: Results for sk-2005.

#masters 1st step	#masters 2nd step	d_2	diameter	difference
2	2	0	24611(24611)	0
4	3	1205	20628(18218)	2410
8	3	3543	22146(17792)	4354
16	5	4170	22772(16671)	6101
32	4	3211	17864(16456)	1408
64	19	2415	19218(17114)	2104
128	31	933	18171(16711)	1400
256	65	611	17726(16711)	1015
512	129	357	17372(16720)	652
1024	293	281	17358(16711)	647
2048	540	104	17089(16662)	427
4096	1076	72	16812(16480)	332
8192	2166	33	16563(16403)	160
16384	4169	24	16466(16399)	67
32768	8288	13	16429(16400)	29
65536	16611	13	16424(16398)	26
131072	33146	12	16422(16398)	24
262144	65800	11	16419(16398)	21
524288	131438	11	16417(16396)	21
1048576	262348	11	16420(16397)	23
2097152	525006	13	16422(16396)	26
4194304	1048206	13	16424(16396)	28
8388608	2097875	13	16427(16397)	30
16777216	4195291	13	16427(16397)	30
33554432	8392344	11	16424(16395)	29
67108864	16777624	11	16424(16395)	29
134217728	33548348	9	16417(16393)	24
268435456	67090630	4	16395(16385)	10

Table 5.13: Results for $\sqrt{n}$-level graph.

#masters 1st step	#masters 2nd step	d_2	diameter	difference
2	2	0	80048947(80048947)	0
4	3	22657921	115863084(93205163)	22657921
8	3	15455762	99957044(75616827)	24340217
16	5	18281627	87327147(75616827)	11710320
32	4	13160133	75244586(67359763)	7884823
64	19	8260457	78240364(69979907)	8260457
128	31	4464271	76958620(68783184)	8175436
256	65	2686227	73948270(68594997)	5353273
512	129	1789819	70865737(67864213)	3001524
1024	293	1046698	69635908(67546538)	2089370
2048	540	433612	68031441(67388623)	642818
4096	1076	293517	67840590(67255072)	585518
8192	2166	134209	67341868(67167702)	174166
16384	4169	66063	67267671(67138713)	128958
32768	8288	38278	67201587(67128199)	73388
65536	16611	23254	67164811(67121749)	43062
131072	33146	15767	67144337(67116317)	28020
262144	65800	7454	67130379(67115306)	15073
524288	131438	3035	67122135(67118032)	4103
1048576	262348	1858	67133664(67127687)	5977
2097152	525006	1186	67168330(67156088)	12242
4194304	1048206	496	67273492(67233693)	39799
8388608	2097875	271	67573199(67419470)	153729
16777216	4195291	139	68266382(67718582)	547800
33554432	8392344	71	69364210(67848160)	1516050
67108864	16777624	40	70183784(67515774)	2668010
134217728	33548348	22	69813122(67168412)	2644710
268435456	67090630	8	67515718(67108864)	406854

Table 5.14: Results for $\Theta(n)$-level graph.

#masters 1st step	#masters 2nd step	d_2	diameter	difference
2	2	0	3686550(3686550)	0
4	2	842299	4412557(2727959)	1684598
8	2	842319	2625216(2515745)	109471
16	4	740115	3379768(2464484)	915284
32	7	479196	3278833(2560355)	718478
64	23	219444	2974775(2560935)	413840
128	27	182914	2623624(2498841)	124783
256	62	115318	2656341(2489948)	166393
512	130	83046	2616367(2475624)	140743
1024	292	48360	2568060(2472742)	95318
2048	540	18465	2518312(2485446)	32866
4096	1085	10666	2542502(2521684)	20818
8192	2158	5087	2603951(2596264)	7687
16384	4168	3297	2757352(2750204)	7148
32768	8285	2249	3058353(3052952)	5401
65536	16609	1281	3649638(3639200)	10438
131072	33146	711	4773474(4746736)	26738
262144	65797	600	6824253(6726590)	97663
524288	131440	457	10149434(9822432)	327002
1048576	262351	369	14331503(13436852)	894651
2097152	525005	320	16966513(14945315)	2021198
4194304	1048203	235	14911235(11316021)	3595214
8388608	2097884	128	11085230(6071011)	5014219
16777216	4195295	64	10233633(4399106)	5834527
33554432	8392347	52	10169386(4108649)	6060737
67108864	16777634	38	8710221(3770069)	4940152
134217728	33548345	28	5832304(3269382)	2562922
268435456	67090613	8	3772310(2440341)	1331969

Table 5.15: Results for worst_PAR_APPROX.

5.3.6 An addition to the input size for the second step of PAR_APPROX_R

Before the results from PAR_APPROX_R will be presented, it will be shown that after two clustering phases the input size is reasonably small so that the internal-memory SSSP can use it for more cases as PAR_APPROX. Table 5.16 shows results with a probability of $p = 1/4$ for a vertex being a master in the second clustering step. Until $\sim 2^{23}$ master vertices for the first phase the output of the second phase is small enough to fit into main memory. So the internal-memory SSSP on the test machine can compute a result in reasonable time.[7] With PAR_APPROX this was possible for each graph class only until around 2^{18} master vertices.[8] Thus it can be expected that the running time over all graphs for a carefully selected number of master vertices should be smaller for PAR_APPROX_R than for PAR_APPROX.

	$\sim 2^{18}$	$\sim 2^{19}$	$\sim 2^{20}$	$\sim 2^{21}$	$\sim 2^{22}$	$\sim 2^{23}$	$\sim 2^{24}$	$\sim 2^{25}$	$\sim 2^{26}$
sk-2005	91.5	196.2	390.2	737.9	1379.7	2795.8	7223.5	21393.7	42201.4
second:	10.5	22.6	46.2	99.5	199.8	385.4	908.5	2357.0	4867.6
Ratio:	0.11	0.11	0.12	0.13	0.14	0.14	0.13	0.11	0.12
$\sqrt{n}$-level	711.9	2239.9	5906.6	11188.6	16128.3	18802.5	19910.7	20830.7	22189.1
second:	47.9	147.9	389.4	780.5	1353.1	2629.8	5253.4	9551.2	16050.1
Ratio:	0.07	0.07	0.07	0.07	0.08	0.14	0.26	0.46	0.72
$\Theta(n)$-level	10.0	20.0	40.0	80.3	162.5	337.4	745.7	1806.6	4584.0
second:	2.5	5.0	10.0	20.0	40.1	80.6	164.0	345.8	770.7
Ratio:	0.25	0.25	0.25	0.25	0.25	0.24	0.22	0.19	0.17
worst_PAR_APPROX	10.0	20.0	40.0	80.0	160.0	320.0	640.0	1280.0	2560.0
second:	2.5	5.0	10.0	20.0	40.0	80.0	160.0	320.1	640.0
Ratio:	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25

Table 5.16: Size of the output of the two clustering phases in Megabyte for all four graph classes. The size of the output of the first clustering phase is in the line with the name of the graph.

[7] Compare Section 5.2.
[8] Compare Section 5.3.3.

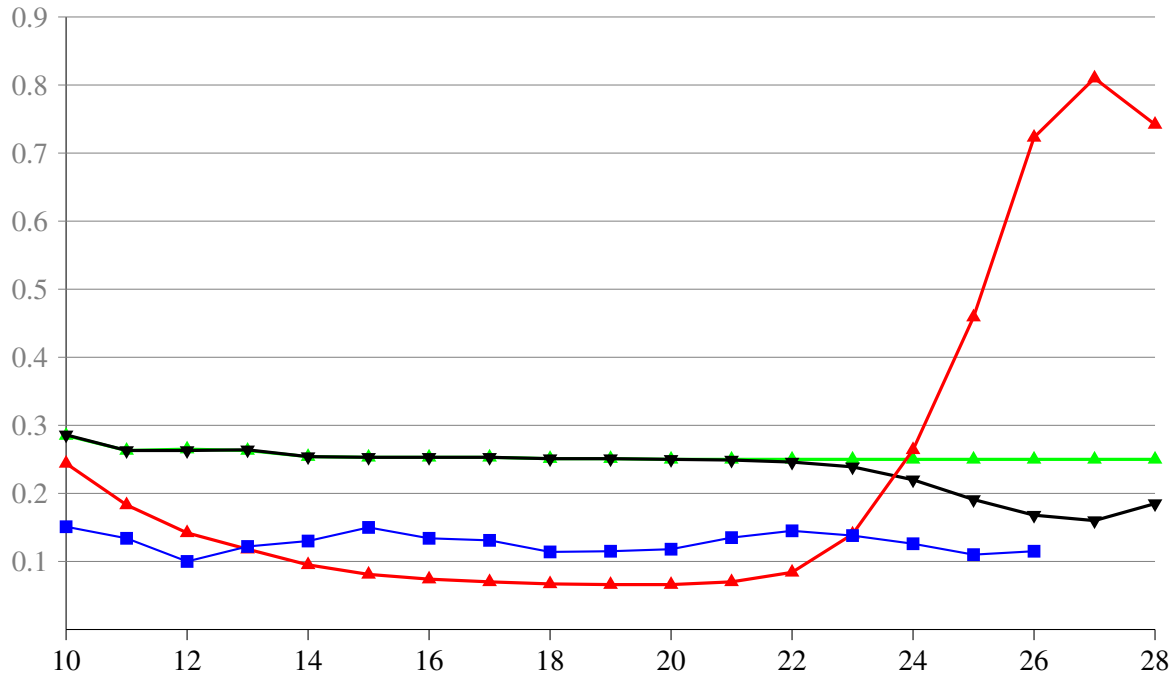

Figure 5.3: The ratio (y-value) of the graph size after the first and the second clustering ($\frac{second}{first}$). The curve of sk-2005 is blue, the cure of $\sqrt{n}$-level graph red, the curve of $\Theta(n)$-level graph black and the curve of worst_PAR_APPROX green. The scale on the x-axis is logarithmic in the number of master vertices.

The curves in Figure 5.3 are showing why it is difficult for $\sqrt{n}$-level graph with many master vertices to find parameters so that the output fits into the main memory. Together with the sizes of the graphs after the first clustering phase in Table 5.16 it is obvious why it is difficult or perhaps not possible for PAR_APPROX to use many master vertices for all graph classes with internal-memory SSSP. It seems that the graph classes are varying in complexity. While trees (worst_PAR_APPROX) and graphs with a small (sk-2005) or tall diameter ($\Theta(n)$-level graph) produce a small output graph after each clustering phase, the output of the graph with a diameter of $\Theta(\sqrt{n})$ is still very tall after a single clustering phase compared to the other graphs. Even after the second phase the output stays tall if the number of vertices is only reduced by a small fraction.

5.3.7 PAR_APPROX_R

The first results of PAR_APPROX_R (Tables 5.17, 5.18, 5.19 and 5.20) were produced with probability p_1 for the first phase equal to the probability p_2 of the second phase. So $p_1 = p_2 = p = \frac{w}{n}$, where w denotes the number of master vertices which should be selected in the first phase and n denotes the number of vertices of the input graph.

With this test scenario it was possible to grow the number of master vertices in the first phase up to around 2^{23}. After the second shrinking step the graph fit into the main memory and it was possible to use internal-memory SSSP in reasonable time.

The slowest case needed 2.805 hours instead of 6.96 hours with around 2^{22} master vertices with the semi external-memory SSSP. Also the resulting diameters are close to the real diameter in many cases for $\sqrt{n}$-level graph and $\Theta(n)$-level graph. For the web graph sk-2005 and for worst_PAR_APPROX the approximated diameter differs from the actual diameter by a factor of 5. This factor is not much taller than the expected error in the single clustering approach PAR_APPROX (compare Subsections 5.3.3 and 5.3.4).

masters step 1	edges step 1	d_1	masters step 2	edges step 2	d_2	diameter	time[h]
32,639	360,192	16	23	149	17	88	0.857
65,457	773,037	17	91	1,395	21	108	0.866
130,995	1,696,456	22	386	9,608	21	128	0.902
261,723	3,799,995	22	1,387	43,953	26	148	0.905
523,308	8,239,006	22	5,403	145,946	37	199	0.912
1,047,527	16,473,867	18	21,611	416,804	33	180	0.926
2,095,915	31,420,311	18	86,570	1,375,518	37	203	0.933
4,193,085	58,008,681	16	34,5904	4,545,748	32	188	0.967
8,387,274	119,281,920	15	1,388,209	17,084,801	26	195	1.090
16,774,091	298,868,555	13	5,554,394	68,158,602	19	185	1.490

Table 5.17: Results for sk-2005.

masters step 1	edges step 1	d_1	masters step 2	edges step 2	d_2	diameter	time[h]
33,061	729,414	14	8	7	7,680	27,070	1.054
65,794	2,607,862	13	18	17	2,752	19,885	1.039
131,451	9,242,749	12	65	64	1,499	18,476	1.040
262,215	30,958,592	12	258	275	324	16,910	1.056
523,572	97,330,056	12	1,036	1,770	115	16,886	1.101
1,048,506	257,456,401	11	3,999	18,584	37	16,615	1.339
2,096,782	487,552,860	10	16,303	249,082	23	16,451	1.691
4,195,701	702,067,655	10	65,751	3,189,088	21	16,448	2.062
8,387,968	815,926,322	9	262,338	36,086,645	19	16,445	2.283

Table 5.18: Results for $\sqrt{n}$-level graph.

masters step 1	edges step 1	d_1	masters step 2	edges step 2	d_2	diameter	time[h]
1,048,506	1,050,414	814	3,999	3,998	135,605	67,339,400	16.170
2,096,782	2,111,432	543	16,303	16,302	43,734	67,234,760	9.096
4,195,701	4,305,371	233	65,751	65,750	10,613	67,259,278	4.737
8,387,968	9,145,224	135	262,338	262,366	3,254	67,469,729	2.805
16,774,408	21,392,325	60	1,048,450	1,050,660	914	67,978,215	1.751
33,554,608	56,559,377	34	4,193,477	4,321,309	259	68,959,950	1.246
67,105,247	155,525,706	16	16,775,155	21,701,263	63	70,255,824	1.141

Table 5.19: Results for $\Theta(n)$-level graph.

masters step 1	edges step 1	d_1	masters step 2	edges step 2	d_2	diameter	time[h]
65,794	65793	449	18	17	834,516	5,096,489	6.418
131,451	131450	223	65	64	385,387	5,406,272	3.770
262,215	262214	144	258	257	174,046	7,062,013	2.443
523,572	523571	84	1,036	1035	61,419	9,926,627	1.611
1,048,506	1048505	50	3,999	3998	38,184	13,577,505	1.219
2,096,782	2096781	37	16,303	16302	9,636	15,234,203	1.014
4,195,700	4195699	28	65,752	65751	3,370	12,557,505	0.892
8,387,969	8387968	24	262,337	262,336	626	10,118,415	0.852
16,774,407	16774406	22	1,048,449	1,048,448	275	11,506,548	0.782

Table 5.20: Results for worst_PAR_APPROX.

graph	masters step 1	edges step 1	d_1	masters step 2	edges step 2	d_2	diameter	time[h]
sk-2005	4,193,085	58,008,681	16	41,763	729,662	31	174	0.980
sk-2005	8,387,274	119,281,920	15	83,592	1,300,219	34	191	1.062
sk-2005	16,774,091	298,868,555	13	167,467	2,479,420	27	159	1.412
$\sqrt{n}$-level	4,195,701	702,067,655	10	42,174	1,427,737	21	16,447	2.059
$\sqrt{n}$-level	8,387,968	815,926,322	9	84,122	4,990,673	20	16,445	2.266
$\sqrt{n}$-level	16,774,408	858,741,691	8	168,573	17,027,083	20	16,443	2.334
$\Theta(n)$-level	4,195,701	4,305,371	233	42,174	42,173	17,389	67,269,876	4.948
$\Theta(n)$-level	8,387,968	9,145,224	135	84,122	84,122	8,520	67,460,611	3.019
$\Theta(n)$-level	16,774,408	21,392,325	60	168,573	168,582	5,076	67,827,503	2.319
worst_PAR_APPROX	4,195,700	4,195,699	28	42,175	42,174	3,386	121,52,755	0.884
worst_PAR_APPROX	8,387,969	8,387,968	24	84,123	84,122	1,432	7,915,132	0.875
worst_PAR_APPROX	16,774,407	16,774,406	22	168,574	168,573	495	7,687,867	0.796

Table 5.21: Results for the second test scenario for PAR_APPROX_R.

In a second test scenario it was tested if the gain can be improved for PAR_APPROX_R if p_1 and p_2 differ and $p_2 < p_1$. In this scenario p_1 was still $\frac{w}{n}$ but now $p_2 = 0.01$. The results in Table 5.21 show that w could be improved. With a taller w the output of the first clustering phase is computed in a reasonable time. With an improved $W \sim 2^{24}$ and 2.319 hours for the slowest case for this w the execution time has been improved by 17.3% compared to the first scenario.

Chapter 6

Conclusion and perspective

Different approaches have been investigated during the work on this thesis. The existing external-memory implementation for diameter approximation with STXXL, SPAN (see Chapter 3) gives results comparable to EM_BFS_DSLB regarding the execution time.

The approaches PAR_APPROX and PAR_APPROX_R are faster than EM_BFS_DSLB and SPAN. Compared to EM_BFS_DSLB a speedup between 51 and 78 percent is achieved.[1]

However, PAR_APPROX and PAR_APPROX_R behave differently for varying graph classes. For small diameter graphs the execution time does not depend on the number of master vertices as much as for large diameter graphs. This is because of the fact that the size of d (e. g. for the $\Theta(n)$-level graph with large diameter) influences the I/O complexity. Hence, for large diameter graphs many master vertices are needed to be faster than EM_BFS_DSLB. This is mitigated by using recursion. However, the constants of the recursive approach PAR_APPROX_R are greater than the constants of PAR_APPROX. For small diameter graphs such as web graphs, PAR_APPROX is a better choice. For general graphs with a priori unknown diameter PAR_APPROX_R is the better choice. However, a tight bound on the expected error for PAR_APPROX_R is still an open question. The lower bound proof in Section 4.5 constitutes a first important step on worst-case inuputs but it is still unclear if this lower bound is tight or if not how many types of inputs are covered by this bound. However, the results of the experiments in Chapter 5 showed that in practice the results of PAR_APPROX_R are viable.

The graph class which led to the greatest problems in quickly finding a viable approximation of the diameter is the $\sqrt{n}$-level graph. The $\sqrt{n}$-level graph has the property that for many master vertices the size of the condensed graph is very big so that it does not fit into main memory. Therefore, in the future it should be tried to improve PAR_APPROX_R with adaptive rules as a mechanism to estimate

[1] Refer to Tables 5.2 and 5.21. For PAR_APPROX_R the results for approximately 2^{24} master vertices for the calculation of the gain were considered.

the probability of being a master in the following clustering step to improve the behavior for each graph class. Then, the gain should be also improved.

Appendix A

I/O Volume of EM_BFS_DSLB and SPAN

	sk-2005	$\sqrt{n}$-level graph	$\Theta(n)$-level graph	worst_PAR_APPROX
read	384.024 *GiB*	539.618 *GiB*	298.521 *GiB*	207.169 *GiB*
write	292.671 *GiB*	347.481 *GiB*	253.741 *GiB*	172.263 *GiB*

Table A.1: I/O-volume of a single external-memory BFS with EM_BFS_DSLB.

	sk-2005	$\sqrt{n}$-level graph	$\Theta(n)$-level graph	worst_PAR_APPROX
read	560.372 *GiB*	786.473 *GiB*	330.762 *GiB*	221.550 *GiB*
write	384.711 *GiB*	405.914 *GiB*	258.400 *GiB*	177.160 *GiB*

Table A.2: I/O-volume of two BFS (double sweep lower bound) with EM_BFS_DSLB. Phase 2 is executed two times. Phase 1 (preprocessing) was executed once.

	sk-2005	$\sqrt{n}$-level graph	$\Theta(n)$-level graph	worst_PAR_APPROX
read	1.733 *TiB*	1.381 *TiB*	647.126 *GiB*	339.666 *GiB*
write	1.736 *TiB*	1.378 *TiB*	642.146 *GiB*	333.823 *GiB*

Table A.3: I/O-volume of SPAN.

One Gigabyte (GiB) are 2^{30} Byte.

One Terabyte (TiB) are 2^{10} Gigabyte.

Appendix B

I/O Volume of PAR_APPROX

$\sim 2^8$	$\sim 2^{10}$	$\sim 2^{12}$	$\sim 2^{14}$	$\sim 2^{16}$	$\sim 2^{18}$	$\sim 2^{20}$	$\sim 2^{22}$
144	151	166	183	198	198	187	170

Table B.1: I/O-volume in Gigabyte for different number of master vertices of the real world graph sk-2005 with diameter 40 for reading.

$\sim 2^8$	$\sim 2^{10}$	$\sim 2^{12}$	$\sim 2^{14}$	$\sim 2^{16}$	$\sim 2^{18}$	$\sim 2^{20}$	$\sim 2^{22}$
44	49	55	60	63	66	67	67

Table B.2: I/O-volume in Gigabyte for different number of master vertices of the real world graph sk-2005 with diameter 40 for writing.

$\sim 2^{10}$	$\sim 2^{12}$	$\sim 2^{14}$	$\sim 2^{16}$	$\sim 2^{18}$
2,777	669	310	285	257

Table B.3: I/O-volume in Gigabyte for different number of master vertices of the $\sqrt{n}$-level graph with diameter 16358 for reading.

$\sim 2^{10}$	$\sim 2^{12}$	$\sim 2^{14}$	$\sim 2^{16}$	$\sim 2^{18}$
339	106	70	68	68

Table B.4: I/O-volume in Gigabyte for different number of master vertices of the $\sqrt{n}$-level graph with diameter 16358 for writing.

$\sim 2^{18}$	$\sim 2^{20}$	$\sim 2^{22}$	$\sim 2^{24}$
23,098	7,164	2,114	642

Table B.5: I/O-volume in Gigabyte for different number of master vertices for different number of master vertices of the $\Theta(n)$-level graph with diameter 67,108,864 for reading.

$\sim 2^{18}$	$\sim 2^{20}$	$\sim 2^{22}$	$\sim 2^{24}$
4,309	1,264	375	123

Table B.6: I/O-volume in Gigabyte for different number of master vertices of the $\Theta(n)$-level graph with diameter 67,108,864 for writing.

$\sim 2^{16}$	$\sim 2^{18}$	$\sim 2^{20}$	$\sim 2^{22}$	$\sim 2^{24}$	$\sim 2^{26}$
2,135	833	401	291	233	103

Table B.7: I/O-volume in Gigabyte for different number of master vertices of the worst_PAR_APPROX with diameter 2,440,341 for reading.

$\sim 2^{16}$	$\sim 2^{18}$	$\sim 2^{20}$	$\sim 2^{22}$	$\sim 2^{24}$	$\sim 2^{26}$
770	292	139	101	83	43

Table B.8: I/O-volume in Gigabyte for different number of master vertices of worst_PAR_APPROX with diameter 2,440,341 for writing.

One Gigabyte are 2^{30} Byte.

Appendix C

File size of the different graphs

graph	size[Bytes]
sk-2005	14,480,405,944
$\sqrt{n}$-level graph	18,036,968,896
$\Theta(n)$-level graph	14,462,023,232
worst_PAR_APPROX	4,294,966,704

Table C.1: Size of the different graph files on the disk. Note that the graphs were stored such that only one direction of an edge is stored in this file. The other direction will be generated while reading the graph. Therefore, at least twice the size of the file as free memory is needed.

Bibliography

[1] U. Meyer and V. Osipov, "Design and implementation of a practical I/O-efficient shortest paths algorithm," in *Proceedings of the annual conference on Algorithm Engineering and Experiments (ALENEX)*, pp. 85–96, SIAM, 2009.

[2] P. Crescenzi, R. Grossi, C. Imbrenda, L. Lanzi, and A. Marino, "Finding the diameter in real-world graphs - experimentally turning a lower bound into an upper bound," in *Proceedings of the 18th annual European Symposium on Algorithms (ESA)*, vol. 6346 of *Lecture Notes in Computer Science*, pp. 302–313, Springer, 2010.

[3] Y. Peres, D. Sotnikov, B. Sudakov, and U. Zwick, "All-pairs shortest paths in $o(n^2)$ time with high probability," *Foundations of Computer Science, Annual IEEE Symposium on*, vol. 0, pp. 663–672, 2010.

[4] J. S. Vitter and E. A. M. Shriver, "Algorithms for parallel memory i: Two-level memories," in *Algorithmica*, vol. 12(2/3), pp. 110–147, 1994.

[5] K. Mehlhorn and U. Meyer, "External-memory Breadth-First Search with sublinear I/O," in *Proceedings of the 10th annual European Symposium on Algorithms (ESA)*, vol. 2461 of *LNCS*, pp. 723–735, Springer, 2002.

[6] U. Meyer, "On trade-offs in external-memory diameter-approximation," in *Proceedings of the 11th Scandinavian Workshop on Algorithm Theory (SWAT)*, pp. 426–436, 2008.

[7] D. Ajwani, A. Beckmann, U. Meyer, and D. Veith, "I/o-efficient approximation of graph diameters by parallel cluster growing – a first experimental study.," 2011.

[8] C. Stegbauer and R. Häußling, eds., *Handbuch Netzwerkforschung*. VS Verlag für Sozialwissenschaften, 2010.

[9] K. Madduri, D. Bader, J. Berry, and J. Crobak, "An experimental study of a parallel shortest path algorithm for solving large-scale graph instances.," in *In Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX 2007)*, ACM-SIAM, 2007.

[10] U. Meyer, *Design and analysis of sequential and parallel single-source shortest-paths algorithms*. PhD thesis, Universität des Saarlandes & Max-Planck Institute (MPI), 2002.

[11] C. Bell, S. Clark, and R. Radcliff, "Mastering eda environments with high performance memory technology," 2011.

[12] Hewlett-Packard Development Company, *Memory technology evolution: an overview of system memory technologies*, 9 ed., 2010.

[13] Texas Memory Systems, Texas Memory Systems, Inc. 10777 Westheimer Rd., Suite 600 Houston, TX 77042, USA , *Increase Application Performance with Solid State Disks*, april 2008 ed., 2008.

[14] Western Digital, "`http://www.wdc.com/wdproducts/library/SpecSheet/ENG/2879-701284.pdf`," 2011. Visited: 01.12.2011.

[15] Seagate, "`http://www.seagate.com/staticfiles/support/docs/manual/enterprise/savvio/Savvio%2015K.3/100629381c.pdf`," 2011. Visited: 01.12.2011.

[16] Kingston, "`http://www.valueram.com/datasheets/KVR1066D3S8N7_2G.pdf`," 2011. Visited: 01.12.2011.

[17] K. Munagala and A. Ranade, "I/O-complexity of graph algorithms," in *Proceedings of the 10th Annual Symposium on Discrete Algorithms (SODA)*, pp. 687–694, ACM-SIAM, 1999.

[18] R. Diestel, *Graph Theory*. pringer Berlin Heidelberg, 2010.

[19] T. H. Cormen, C. E. Leiserson, R. Rivest, and C. Stein, *Algorithmen - Eine Einführung*. München, Deutschland: Oldenbourg Wissenschaftsverlag GmbH, 2007.

[20] W. D. Wallis, *A Beginner's Guide to Graph Theory*. Birkhäuser Boston, 2007.

[21] C. Magnien, M. Latapy, and M. Habib, "Fast computation of empirically tight bounds for the diameter of massive graphs," *Journal of Experimental Algorithmics*, vol. 13, pp. 1.10–1.9, 2009.

[22] J. S. Vitter, "External memory algorithms and data structures: dealing with massive data," *ACM Comput. Surv.*, vol. 33, pp. 209–271, June 2001.

[23] L. Arge, M. Knudsen, and K. Larsen, "A general lower bound on the i/o-complexity of comparison-based algorithms," in *Algorithms and Data Structures* (F. Dehne, J.-R. Sack, N. Santoro, and S. Whitesides, eds.), vol. 709 of *Lecture Notes in Computer Science*, pp. 83–94, Springer Berlin / Heidelberg, 1993. 10.1007/3-540-57155-8_238.

[24] R. Dementiev, L. Kettner, , and P. Sanders, "Stxxl: standard template library for xxl data sets," *Software: Practice and Experience*, vol. 38, pp. 589–637, 2008.

[25] A. Beckmann, R. Dementiev, and J. Singler, "Building a parallel pipelined external memory algorithm library," *IPDPS*, 2009.

[26] D. Ajwani, R. Dementiev, and U. Meyer, "A computational study of external memory bfs algorithms," in *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 601–610, 2006.

[27] D. Ajwani, U. Meyer, and V. Osipov, "Improved external memory BFS implementation," in *Proceedings of the workshop on Algorithm Engineering and Experiments (ALENEX)*, pp. 3–12, 2007.

[28] O. Borůvka, "O jistém problému minimálním," *Práce Mor. Přírodověd. Spol. v Brně*, vol. 3, pp. 37 –58, 1926.

[29] V. Jarník, "O jistém problému minimálním," *Práce Mor. Přírodověd. Spol. v Brně*, vol. 6, pp. 57 –63, 1930.

[30] R. C. Prim, "Shortest connection networks and some generalizations," *THE BELL SYSTEM TECHNICAL JOURNAL*, pp. 1389–1401, November 1957.

[31] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proc. Amer. Math. Soc.*, vol. 7, pp. 48–50, 1956.

[32] M. L. Fredman and D. E. Willard, "Trans-dichotomous algorithms for minimum spanning trees and shortest paths," *Journal of Computer and System Sciences*, vol. 48, no. 3, pp. 533 – 551, 1994.

[33] J. Abello, A. Buchsbaum, and J. Westbrook, "A functional approach to external graph algorithms," in *Algorithms - ESA' 98* (G. Bilardi, G. Italiano, A. Pietracaprina, and G. Pucci, eds.), vol. 1461 of *Lecture Notes in Computer Science*, pp. 1–1, Springer Berlin / Heidelberg, 1998. 10.1007/3-540-68530-8_28.

[34] A. Rényi and G. Szekeres, "On the height of trees," *Journal of the Australian Mathematical Society*, vol. 7, pp. 497–507, 1967.

[35] I. I. Brudaru, "Heuristics for average diameter approximation with external memory algorithms," Master's thesis, Universität des Saarlandes, October 2007.

[36] S. Singh, *Fermats letzter Satz: Die abenteuerliche Geschichte eines mathematischen Rätsels*. München, Deutschland: Deutscher Taschenbuch Verlag, 2000.

[37] M. Bender and M. Farach-Colton, "The lca problem revisited," in *LATIN 2000: Theoretical Informatics* (G. Gonnet and A. Viola, eds.), vol. 1776 of *Lecture Notes in Computer Science*, pp. 88–94, Springer Berlin / Heidelberg, 2000. 10.1007/10719839_9.

[38] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter, "External-memory graph algorithms," in *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, SODA '95, (Philadelphia, PA, USA), pp. 139–149, Society for Industrial and Applied Mathematics, 1995.

[39] J. JáJá, *An introduction to parallel algorithms*. Addison-Wesley Pub. Co., 1992.

[40] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Nummerische Mathematik 1*, pp. 269–271, 1959.

[41] D. E. Knuth, *The Art of Computer Programming Volume 3: Sorting and Searching - Second Edition*, vol. 3. Addison-Wesley, June 2008. Second Edition.

[42] U. Meyer, "Single-source shortest-paths on arbitrary directed graphs in linear average-case time," in *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, SODA '01, (Philadelphia, PA, USA), pp. 797–806, Society for Industrial and Applied Mathematics, 2001.

[43] U. Meyer and N. Zeh, "I/O-efficient undirected shortest paths," in *Proceedings of the 11th annual European Symposium on Algorithms (ESA)*, vol. 2832 of *Lecture notes in Computer Science (LNCS)*, pp. 434–445, Springer, 2003.

[44] H. Chernoff, "A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations," *Ann. Math. Statist.*, vol. 23, no. 4, pp. 493–507, 1952.

[45] A. Klenke, *Wahrscheinlichkeitstheorie*. Springer, 2008.

[46] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook, "On external memory graph traversal," in *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, SODA '00, (Philadelphia, PA, USA), pp. 859–860, Society for Industrial and Applied Mathematics, 2000.

[47] D. G. Corneil, F. F. Dragan, M. Habib, and C. Paul, "Diameter determination on restricted graph families," *Discrete Applied Mathematics*, vol. 113, no. 2-3, pp. 143–166, 2001.

[48] Y. J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamasia, D. E. Vengroff, and J. S. Vitter, "External memory graph algorithms," in *Proceedings of the 6th annual Symposium on Discrete Algorithms (SODA)*, pp. 139–149, ACM-SIAM, 1995.

[49] M. Metzler, "Ergebnisüberprüfung bei graphenalgorithmen.," Master's thesis, Universität des Saarlandes, 1997.