I/O-efficient approximation of graph diameters by parallel cluster growing – a first experimental study.[*]

Deepak Ajwani
Centre for Unified Computing
University College Cork
Cork, Ireland
deepak.ajwani@gmail.com

Andreas Beckmann Ulrich Meyer David Veith
Institut für Informatik
Goethe-Universität Frankfurt am Main
Frankfurt am Main, Germany
{beckmann,umeyer,dveith}@cs.uni-frankfurt.de

***Abstract*— A fundamental step in the analysis of a massive graph is to compute its diameter. In the RAM model, the diameter of a connected undirected unweighted graph can be efficiently 2-approximated using a Breadth-First Search (BFS) traversal from an arbitrary node. However, if the graph is stored on disk, even an external memory BFS traversal is prohibitive, owing to the large number of I/Os it incurs. Meyer [1] proposed a parametrized algorithm to compute an approximation of graph diameter with fewer I/Os than that required for exact BFS traversal of the graph. The approach is based on growing clusters around randomly chosen vertices 'in parallel' until their fringes meet. We present an implementation of this algorithm and compare it with some simple heuristics and external-memory BFS in order to determine the trade-off between the approximation ratio and running-time achievable in practice. Our experiments show that with carefully chosen parameters, the new approach is indeed capable to produce surprisingly good diameter approximations in shorter time. We also confirm experimentally, that there are graph-classes where the parametrized approach runs into bad approximation ratios just as the theoretical analysis in [1] suggests.**

I. Introduction

Massive graphs arise naturally in many applications. Social network graphs or the WWW graph have implicitly become part of our daily life. A whole branch of computer science deals with network analysis [2]. A fundamental step in the analysis of a massive graph is to compute its diameter: In this paper, we consider connected undirected unweighted large sparse graphs $G(V, E)$ where $n = |V|$ and $m = |E|$. The distance $d(u, v)$ between two nodes $u, v \in V$ is the number of edges in the shortest path connecting u and v. The eccentricity of a node v is defined as $\mathrm{ecc}(v) = \max_u d(v, u)$. Finally, $D = \max_{u,v} d(u, v) = \max_v \mathrm{ecc}(v)$ is called the diameter of G. We are particularly interested in the case when G is sparse ($m = O(n)$) but nevertheless too big to fit into the main memory of a single computing device. In that setting the typical strategies are: (i) to distribute the data over many computers and apply parallel algorithms (e. g., see [3], [4]) and/or (ii) store the data on secondary memory like hard disks or flash memory. In this paper we will concentrate on practically feasible diameter approximation algorithms for the latter (external memory) approach.

External memory model: The huge difference in time between accessing an element from the main memory and fetching an element from the disk is nicely captured by the external memory (EM) model (sometimes also called the I/O model), which was introduced by Aggarwal and Vitter [5]. It assumes a two level memory hierarchy. The internal memory is fast, but has a limited size of M elements (nodes/edges). In addition, there is an external memory which can only be accessed using I/Os that move B contiguous elements between internal and external memory. At any particular time, the computation can only use the data already present in the internal memory. The measure of performance of an algorithm is the number of I/Os it performs – the less I/Os the algorithm requires, the better it is. The number of I/Os required to scan n contiguously stored elements in the external memory is $\mathrm{scan}(n) = O(n/B)$ and the number of I/Os required for sorting n elements is $\mathrm{sort}(n) = O(\frac{n}{B} \log_{M/B} n/B)$. For realistic values of B, M and n, $\mathrm{scan}(n) < \mathrm{sort}(n) \ll n$ and the goal of designing external memory algorithms is often to reduce the I/O complexity from $O(n)$ to $O(\mathrm{sort}(n))$. Further discussions on realistic models for computing on large data can be found in a recent survey article by Ajwani and Meyerhenke [6].

Outline: Section II presents an overview of various algorithms and heuristics designed in recent years to compute the exact or approximate diameter including the approaches implemented in this project. Section III provides some implementation details. The experimental results are detailed in Section IV and we conclude in Section V.

II. Related Work

The problem of computing the diameter of large graphs, particularly for complex networks, has received considerable attention lately, both from an algorithmic and empirical point of view. For the exact computation of diameters in unweighted undirected graphs, breadth-first search (BFS) can be executed from all nodes of the graph and the longest height of a BFS tree is reported as the diameter. Doing so naively requires $O(n^2 + mn)$ operations. However, the method can be improved by logarithmic factors (e. g., [7]) in the standard RAM model

[*]Partially supported by the DFG grant ME 3250/1-3, and by MADALGO – Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

with logarithmic word size. There are also algebraic approaches for computing all pairs breadth-first search (AP-BFS) based on matrix-multiplication based algorithm (e. g., [8]). But the exact diameter computation based on either combinatorial or algebraic approaches remains computationally expensive and impractical for massive sparse graphs. Even the recent result by Peres et al. [9] who gave an $O(n^2)$ algorithm for computing all-pair shortest path with high probability is infeasible for such graphs.

A. Internal-Memory Small-Factor Approximations

As the exact computation of diameters for massive sparse graphs seems impractical, approximation algorithms and heuristics have received attention lately. In the following we review some strategies that rely on the fact that a rather small number of BFS computations can easily be afforded as long as the input graph fits into internal memory.

The trivial bounds: It is folklore that already a single BFS run rooted at an arbitrary source r yields trivial lower and upper bounds on the diameter: $\mathrm{ecc}(r) \leq D \leq 2 \cdot \mathrm{ecc}(r)$. Obviously, the choice of r determines which of these bounds is tight (if any). The approximation can be improved by performing k BFS explorations from k carefully chosen starting points [10]: in that case the additive error drops to $O(n/k)$ at the cost of increased running time (for k BFS runs).

The double sweep heuristic: In practice the trivial lower bound based on a BFS run with some random source r can frequently be improved significantly by just one additional BFS run from some source v' satisfying $d(r, v') = \mathrm{ecc}(r)$ and reporting the bound $d(v', v'') = \mathrm{ecc}(v')$ for some node v'' with maximal depth in the BFS tree for source v'. This technique is also called the *double sweep method* (e. g., see [11], [12]). On certain graph classes including trees the double sweep method even guarantees to yield a tight lower bound on the diameter [11]. On general graphs, the double sweep lower bound will at least not be worse than the trivial one. In addition, the double sweep method can be iterated for different sources but then previously applied sources for the respective second BFS runs should not be reused. The double sweep lower bound can also be used to yield improved *upper* bounds on the diameter of these BFS trees since it is able to derive the exact diameter of a tree (and hence also of a BFS tree) as mentioned above. Again iterating over several carefully chosen BFS trees may strengthen the upper bound even further. Observe, however, that there are graph classes (like rings) where any BFS tree of those graphs has a larger diameter than the respective original graph (up to a factor of two).

The fringe heuristic: As long as the input graph fits into main memory, the name of the game is to find matching upper and lower bounds for many graph classes while investing as few BFS traversals as possible. To the best of our knowledge, the most efficient approach of this kind is the *fringe* heuristic by Crescenzi et al. [13]. For some vertex u, the fringe of u, denoted $F(u)$, is set of all vertices $v \in V$ such that $d(u, v) = \mathrm{ecc}(u)$. The fringe heuristic uses the double sweep method to find a lower bound on the diameter and computes an upper bound on the diameter as follows:

1) Let r, v', and v'' be the vertices identified by double sweep method.
2) Find the vertex u that is halfway along the path connecting v' and v'' inside the BFS-tree rooted at v'.
3) Compute the BFS tree for source u and its eccentricity $\mathrm{ecc}(u)$.
4) If $|F(u)| > 1$, find the BFS trees for all sources $z \in F(u)$, and compute $B(u) = \max_{z \in F(u)} \mathrm{ecc}(z)$:
   a) If $B(u) = 2 \cdot \mathrm{ecc}(u) - 1$, return $2 \cdot \mathrm{ecc}(u) - 1$.
   b) If $B(u) < 2 \cdot \mathrm{ecc}(u) - 1$, return $2 \cdot \mathrm{ecc}(u) - 2$.
5) Return the diameter of the BFS tree rooted at u.

It is shown in [13] that the fringe algorithm correctly computes an upper bound on the diameter using at most $|F(u)| + 3$ BFS traversals. While $|F(u)| = \Omega(|V|)$ in the worst case, Crescenzi et al. demonstrate that $|F(u)|$ is often rather small (less than 20) for real world graphs. Additionally, for nearly all tested cases in [13] the fringe heuristic produced matching lower and upper bounds.

B. External-Memory Approaches

There has been a significant number of publications on external-memory graph algorithms; see [14], [15] for recent overviews. Exact computation of the diameter on unweighted undirected graphs (via All-Pairs Shortest-Paths, APSP) has been addressed in [16], [17]: both approaches require $\Theta(n \cdot \mathrm{sort}(n))$ I/Os for sparse graphs. Taking into account that current machines easily feature several gigabytes of RAM, in the external-memory setting where $n > M \gg B$, an algorithm spending $\Theta(n \cdot n/B) = \Omega(n^2/B)$ I/Os is practically useless.

Small-factor approximations: Chowdhury and Ramachandran [17] also gave an algorithm for computing approximate all-pairs shortest-paths with additive error. However, their approach only takes less I/O than exact EM APSP when $m \geq n \log n$, which is not the sparse graph case we are interested in, and even then the I/O volume is huge.

Of course, the simple BFS-based RAM approximation approaches discussed above can be implemented in external-memory using EM BFS. However, even for the trivial 2-approximation this takes $\Omega(n/\sqrt{B})$ I/Os in the worst-case [18]. What this means in practice will be discussed in Section IV.

Parametrized approximations: In the following, we review a recent parametrized approach by Meyer [1] to trade approximation quality with sub-BFS I/O time, which we will also experimentally evaluate in Section IV. The problem of computing an approximate diameter of the input graph G (with n nodes and m edges) is reduced to that of computing exact shortest paths on a weighted graph G' with $O(n/k)$ nodes and $O(m)$ edges. Graph G' is computed using a static external memory BFS [18] like preprocessing as follows: We first choose each node to be a master node with a probability $1/k$. Additionally, we select every k-th node in the Euler-tour traversal around an arbitrary spanning tree of G, to also be a master node. Thereafter, we grow the clusters "in parallel".

In each round, each master node tries to capture all unvisited neighbors of the current cluster. This is done by first sorting the nodes at the fringes of the clusters and then scanning the adjacency-lists of the nodes in the yet unexplored graph. Ties are broken arbitrarily.

Let $C(u)$ be the cluster containing u. An edge $\{u, v\} \in G$ results in an edge $\{C(u), C(v)\} \in G'$ if $C(u) \neq C(v)$. The weight of the created edge $\{C(u), C(v)\}$ is $d_c(u)+1+d_c(v)$, where $d_c(u)$ is the distance of u from its cluster center. We remove the parallel edges by keeping only the lightest edge between $C(u)$ and $C(v)$. We run single source shortest path (SSSP) from an arbitrary node s in G' and output the maximum distance from s to any other node in G'. Note that this is a constant-factor approximation to the weighted diameter of G'. Meyer [1] showed that the expected weighted diameter of G' satisfies $D_{G'} = O(\sqrt{k} \cdot D_G)$.

Since each k-th node on the Euler tour is a master node, each node $u \in G$ is at most distance k away from a master node and the clusters are grown for at most k rounds. As each cluster growing round requires $O(\text{scan}(m))$ I/Os to scan the adjacency lists of unexplored graph and each node appears only once as a fringe node of some cluster leading to a total of $O(\text{sort}(n))$ I/Os, the total complexity of computing G' is $O(k \cdot \text{scan}(n+m) + \text{sort}(n+m) + ST(n,m))$ I/Os, where $ST(n,m)$ is the I/O complexity of computing a spanning tree of an n node and m edge undirected graph: $O(\text{sort}(n+m))$ I/Os randomized [19] and $O(\text{sort}(n) \log\log \frac{n \cdot B}{m})$ I/Os with a deterministic spanning tree algorithm [20].

Computing single source shortest path on a graph with $O(n/k)$ nodes and $O(m)$ edges with the ratio between maximum and minimum edge weight being k requires $O(\sqrt{\frac{n \cdot m}{k \cdot B}} \log_2 k + \text{sort}(n+m) + ST(n,m))$ I/Os [21]. The total I/O complexity for this algorithm is thus $O(\sqrt{\frac{n \cdot m}{k \cdot B}} \log_2 k + k \cdot \text{scan}(n+m) + \text{sort}(n+m) + ST(n,m))$ I/Os.

Spanning tree heuristics: While the parametrized approximation discussed above still offers some (expected) approximation guarantees one might also go to the extreme: omit any kind of guarantee and just rely on an I/O-efficient heuristic. While (at least in theory) BFS computations on sparse graphs tend to spend much more I/O than spanning tree computations it is natural to ask if one could use a spanning tree rather than a BFS traversal for approximating the diameter. Unfortunately, the diameter of a spanning tree can be very far from the diameter of the graph. For instance, consider a cycle graph $u_1 \dots u_{n-1}$ of $n-1$ nodes and edges and a special node s with edges to all nodes of the cycle. The diameter of a spanning tree $\{s, u_1\}, \{u_1, u_2\}, \dots, \{u_{n-1}, u_n\}$ is $n-1$ while the diameter of the original graph is 2. Unfortunately, even the diameter of a random spanning tree can be very far away from the diameter of the graph. For instance, Rényi and Szekeres [22] showed that the expected diameter of a random spanning tree in the complete graph K_n is $O(\sqrt{n})$.

A simpler way to use randomization in this context is to consider the minimum spanning tree in the original graph with edge weights assigned independently and uniformly from the range $(0, 1]$. However, even the diameter of such a spanning tree can be quite large compared to the diameter of the graph. Nevertheless, our heuristic based on initial work by Brudaru [23] takes this spanning tree as the base case and iteratively refines it to approximate a BFS tree rooted at some arbitrary node s. Let T_i be the spanning tree after the i-th iteration (minimum spanning tree with random weights being T_0) and $h_i(u)$ be the distance of node u from s in T_i. Each iteration consists of carefully selecting the edges for T_{i+1} such that for each node u, $h_{i+1}(u) \leq h_i(u)$ and for at least one node v, $h_{i+1}(v) < h_i(v)$, thus, eventually the sequence $T_0, T_1, \dots, T_i$ converges to a BFS tree with root s.

An iteration consists of scanning the list of edges of the original graph and for each node u, selecting the edge $\{v, u\}$ such that $h_i(v)$ is minimum. This is done independently for all nodes. Note that although some neighbors of u may have found a shorter path to s in the course of this iteration, this is completely ignored as using this information naively may require random I/Os.

III. Implementation details

For our experimental study we implemented or modified three approaches: (i) we modified the external memory BFS [24] to use double sweep lower bound, (ii) we re-implemented the spanning tree heuristics from Section II-B, and (iii) we engineered a simplified version of the parametrized approximation algorithm from Section II-B.

Our C++ code uses the external memory library STXXL [25] ver. 1.3.1 for algorithms like sorting and data structures like priority queues. An additional benefit of using STXXL is the streaming interface of various algorithms that allows us to make extensive use of pipelining to save a factor of 2–3 in the total I/O volume.

Implementation of the parametrized approximation: The data structures and graph generators used in our code are similar to those of BFS implementation of Ajwani et al. [24]. This was done to ensure better comparison with the external memory BFS implementations. Also, we use their pipelined randomized clustering implementation in our approach to compute the clustering of the input graph.

Notwithstanding the theoretical description of the parametrized approximation approach in [1] and Section II-B we omitted to choose extra masters deterministically and only rely on the randomly chosen master vertices. We then create a condensed graph based on this clustering using a constant number of sorting and scanning rounds. If the condensed graph fits internally, we use an internal memory SSSP sub-routine that we implemented using STL. Otherwise, we use our adaptation of the *semi*-external memory SSSP sub-routine by Meyer and Osipov [26], SE_SSSP for short, to compute the diameter of the weighted condensed graph. Note that this code requires at least one bit per vertex of the condensed graph in internal memory and also drops strict performance guarantees for non-random edge weights, which occur in our application. We refer to our implementation of the parameterized approximation algorithm as PAR_APPROX.

Both SSSP-subroutines (internal or semi-external) apply the double sweep lower bound ideas [12] to find a source which guarantees reasonable values for the resulting diameter.

Implementation of the spanning tree heuristics: In her original work [23], Brudaru implemented these heuristics in internal memory using the LEDA library and also created an external memory prototype. We re-implemented the heuristics to maximally utilize the pipelining and other features offered by STXXL.

IV. Experimental results

The goal of our experiments is (i) to determine the trade-off between approximation quality and running time (dominated by I/Os) and (ii) to ascertain if the diameter for massive sparse graphs can be computed in a reasonable time (e. g. overnight running on a standard desktop PC). Our hope is that the insights learned during these experiments will assist a practitioner to determine the right technique for computing the diameter of a given graph.

Graph classes: We chose three different graph classes: one real-world graph with logarithmic diameter and two synthetic graph classes with diameter $\Theta(\sqrt{n})$ and $\Theta(n)$. The real-world graph sk-2005 has around 50 million vertices, about 1.8 billion edges and is based on a web-crawl. It was selected for better comparison with DSLB_UP_BOUND of Crescenzi et al. [13] and because it has a known diameter of 40.

The synthetic x-level graphs are similar to the B-level random graphs in [24]. The graph consists of x levels, each having $\frac{n}{x}$ vertices (except for level 0 which contains only one vertex). The edges are randomly distributed between consecutive levels, such that these x levels approximate the BFS levels if BFS were performed from the source vertex in level 0. We selected $x = \sqrt{n}$ and $x = \Theta(n)$ to generate $\sqrt{n}$ and $\Theta(n)$-level graphs with 2^{28} vertices and around 1 billion edges for our experiments. The generated graphs have 1,127,310,556 edges ($\sqrt{n}$-level graph) and 903,876,452 ($\Theta(n)$-level graph).

To elicit the worst-case approximation ratio from the PAR_APPROX approach, we also generated another graph from a class with three parameters: k_1, k_2 and k_3 satisfying $n = k_3 \cdot (k_1 + k_2)$. It consists of a list of length k_3. There are k_3 node-disjoint lists of length k_1 incident on each vertex of the original list (of size k_3). Each of the k_3 lists have a fan-out of k_2 at the other end. The main idea behind this graph class is as follows: for appropriately chosen k_j values, there are most masters of PAR_APPROX in the fans and only few masters in the lists so that many clusters meet with large edge weights at the original list of length k_3, thus blowing up the weighted diameter of the condensed graph significantly.

We randomize the layout of the synthetic graphs on the disk to ensure that the disk layout does not reveal any additional information that is exploitable. However, we use the ordering provided with sk-2005 graph for fair comparison with results reported in the literature.

Configuration: We performed our experiments on two different architectures.

(i) To determine the behavior of different techniques in an external memory setting, we used a machine with an Intel dual core E6750 processor @ 2.66 GHz, 4 GB internal memory (around 3.5 GB free), 4 hard-disks with 500 GB each as external memory for STXXL, and a separate disk for the operating system, application and storing data, logfiles etc. The operation system was Debian GNU/Linux amd64 'wheezy' (testing) with kernel 3.0. The programs were compiled with GCC 4.4 in C++0x mode using optimization level 3.

(ii) For running the heuristics of Crescenzi et al. [13] in internal memory, we used a machine (part of the HPC cluster at Goethe University) with 4 quad-core AMD Opteron$^{\text{TM}}$ processor 8384 @ 2.7 GHz (only one core was used) and 64 GB internal memory. Note that the purpose of these experiments is to determine the quality of approximation with their approach and to ascertain if it can be matched in an external memory setting. The running time of an approach on this machine is *no* indication of the running time in an external memory setting.

Selecting the correct number of master vertices for PAR_APPROX: Theoretically, as we increase the number of master vertices, the maximum distance d between them should decrease. This should help to reduce the running time of the clustering phase and improve the approximation quality (as the condensed graph better captures the structure of the original graph and the factor $2 \cdot d$ added to the calculated diameter is low). The penalty paid for this is the increased I/O time for external memory SSSP. Varying the number of master vertices gives a trade-off between approximation ratio and the running time.

However, most worst-case efficient external memory SSSP approaches are impractically sophisticated. As such, we have to rely on internal and semi-external memory SSSP (SE_SSSP). This imposes additional constraints on the maximum number of master vertices as for using the internal memory SSSP, the condensed graph should fit internally, while for using SE_SSSP the number of master vertices should be less than the main memory size in bits.

When using the internal memory SSSP, we would like to choose the largest number of master nodes such that the condensed graph fits internally. However, identifying this number is a non-trivial task because the graph density of the condensed graph depends on the structure of the input graph. The condensed graph of a random graph is significantly more dense than the one for a list graph. Therefore, we would like to select the number of master vertices based on the structure of the graph – fewer master vertices for random graphs than for $\Theta(n)$-level graphs. However, selecting the number of master vertices on the basis of graph type requires a priori information about the graph class, which runs contrary to our objective of analyzing a given graph by determining its diameter.

Thus, we have two alternatives: We can either start with a small (e. g., $O(\sqrt{M})$) number of master vertices such that the condensed graph is guaranteed to fit internally. Thereafter, we can adaptively compute the correct number of master vertices by increasing the number if the running time for the clustering is too high (and aborting and redoing the run with

	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 case for PAR_APPROX	**2,440,341**	3,982,472	**2,440,341**

TABLE I
DIAMETERS APPROXIMATED BY VARIOUS APPROACHES. EXACT DIAMETERS ARE MARKED IN BOLDFACE.

the new number) or by decreasing the number if the resultant condensed graph does not fit internally.

The other alternative is to choose a large number of master vertices and use SE_SSSP on the condensed graph. This is almost always faster than EM_BFS_DSLB. We can then reduce the number of master vertices to get a trade-off between approximation quality and runtime. We have used both of these alternatives in our experimental study.

Results: First we present the results of three different approaches: External memory BFS with double sweep lower bound (EM_BFS_DSLB), the spanning tree heuristic (SPAN) and the fringe approach from Crescenzi et al. with dslb method [13] (DSLB_UP_BOUND). The external BFS and the internal DSLB_UP_BOUND showed similar results. For sk-2005 we got a lower bound of 39 instead of 40. With a second experiment with a different carefully chosen source we have found the lower bound of 40, too.

For DSLB_UP_BOUND, we used ten iterations in one experiment. The other applications we executed with only one iteration.

The results of the SPAN heuristic were not that close to the real diameter and have a taller spread. The numbers in Table I are the resulting heights of the trees multiplied with factor of two as an upper bound.

We do not report the detailed running times of DSLB_UP_BOUND, since it was only executed on the big 64 GB internal memory machine and was merely used to get hold of the exact diameters. As can be seen in Table II, the SPAN heuristic was often slower than EM_BFS_DSLB. Only for the $\sqrt{n}$-level graph we obtained a better running time behavior. In correspondence with the results in [24], graphs sk-2005 (with a low diameter) and $\Theta(n)$-level (which is similar to a single chain) are easier for external BFS than the $\sqrt{n}$-level graph (which shares some characteristics with grid graphs).

	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 case for PAR_APPROX	1.66	3.34

TABLE II
RUNNING TIMES (IN HOURS) FOR EM_BFS_DSLB AND SPAN.

Results for the PAR_APPROX implementation: To learn more about the behavior of our new approach we ran a couple of different test scenarios. In Table III we report on the results when the condensed graph G' fits into internal memory. The running times are dominated by the clustering. The internal-memory SSSP for G' usually took only a few seconds.

As for sk-2005, PAR_APPROX was up to 10 times faster than EM_BFS_DSLB and SPAN. Interestingly, the best approximation guarantee (42 vs. 40 exact) was obtained for small numbers of master nodes. Running time and approximation deteriorate with more masters. We verified this phenomenon on the machine with 64 GB internal memory and more samples. When between ten and twenty percent of the original graph nodes are chosen as master vertices for sk-2005, there is a point where the approximation bound improves again. We saw this behavior more or less pronounced for other real world graphs tested in [13], too.

The results for the $\Theta(\sqrt{n})$-level graph showed just the opposite trend: best running times (up to 12 times faster than EM_BFS_DSLB) and approximation bounds (only 0.14 % away from the exact diameter but nearly three times better than SPAN) were obtained for the largest possible number of masters.

While for graphs with smaller diameter like sk-2005 a small number of master vertices like 2^{14} produces good running times, for the $\Theta(n)$-level graph applying too few master nodes would be dangerous: with a diameter of over 900 millions and only 2^{14} master nodes, more than 50,000 phases of the parallel cluster growing would be needed. The estimated time for this clustering is around a month. Fortunately, the condensed graph G' from $\Theta(n)$-level graph is rather small even for a high number of master vertices: G' fits into internal memory on a 4 GB machine until 2^{25} master vertices. While the approximation guarantee is still within 1 %, the running time gain for 2^{24} masters is only a factor of four.

Expectedly bad approximations bounds could be identified for our worst-case graph, especially with parameters $k_1 = 10$ and $k_2 = 100$ for $n = 2^{28}$, resulting in a diameter of about 2.44 million for $n = 2^{28}$. With 2^{20} master nodes, the PAR_APPROX overestimated the real diameter by more than a factor of five.

As can be seen in Table III for high diameter graph classes we need – and can afford – a lot of master vertices, but for $\Theta(\sqrt{n})$-level inputs or $\Theta(n)$-level inputs the resulting condensed graphs are too dense to be used in internal memory. Hence, we tested the behavior of PAR_APPROX for three different numbers of master vertices with SE_SSSP.

The results in Table IV show that also with SE_SSSP as a subroutine PAR_APPROX is faster than EM_BFS_DSLB but not very much. Nevertheless, what is also important, the resulting diameters are still very close to the real diameters.

masters	$\sim 2^8$	$\sim 2^{10}$	$\sim 2^{12}$	$\sim 2^{14}$	$\sim 2^{16}$	$\sim 2^{18}$	$\sim 2^{20}$	$\sim 2^{22}$	$\sim 2^{24}$
sk-2005									
computed approx. diameter	42	51	68	79	106	113	98	119	
approximation ratio	1.05	1.28	1.70	1.98	2.65	2.83	2.45	2.98	
time [h]	0.46	0.51	0.62	0.68	0.73	0.76	0.78	0.77	
d	12	13	17	17	17	22	18	16	
$\sqrt{n}$-level									
computed approx. diameter		16,836	16,519	16,413	16,409	16,408			
approximation ratio		1.0275	1.0082	1.0017	1.0015	1.0014			
time [h]		4.29	1.30	0.87	0.87	0.85			
d		156	35	15	13	12			
$\Theta(n)$-level									
diameter						67,118,479	67,128,342	67,233,297	67,717,702
approx. ratio						1.00014	1.00029	1.00185	1.00907
time [h]						41.60	12.33	3.68	1.26
d						3444	814	233	60
worst case									
computed approx. diameter					3,643,615	6,729,783	13,461,919	11,265,297	4,399,657
approximation ratio					1.49	2.76	5.52	4.62	1.80
time [h]					5.12	1.87	0.92	0.73	0.58
d					449	144	50	28	22

TABLE III

RESULTS OF PAR_APPROX FOR DIFFERENT NUMBERS OF MASTER VERTICES, WHEN THE CONDENSED GRAPH FITS INTO INTERNAL MEMORY. d IS THE MAXIMUM DISTANCE BETWEEN TWO MASTER VERTICES.

masters	$\sim 2^{22}$	$\sim 2^{24}$	$\sim 2^{26}$
sk-2005			
computed approx. diameter	119	90	
approximation ratio	2.98	2.25	
time [h]	1.09 (0.32)	2.78 (1.73)	
d	16	13	
vertices in G'	4,193,085	16,774,091	
edges in G'	58,008,681	298,868,555	
$\sqrt{n}$-level graph			
computed approx. diameter	16,407	16,403	16,401
approximation ratio	1.0013	1.0011	1.0010
time [h]	6.96 (5.51)	8.38 (6.80)	9.41 (7.72)
d	10	8	7
vertices in G'	4,195,701	16,774,408	67,105,247
edges in G'	702,067,655	858,741,691	924,712,355
$\Theta(n)$-level graph			
computed approx. diameter	67,233,297	67,717,702	67,515,826
approximation ratio	1.00185	1.00907	1.00606
time [h]	3.94 (0.06)	1.6 (0.29)	2.47 (1.68)
d	233	60	16
vertices in G'	4,195,701	16,774,408	67,105,247
edges in G'	4,305,371	21,392,325	155,525,706

TABLE IV

RESULTS OF PAR_APPROX WITH SEMI-EXTERNAL SSSP. THE RUNNING TIMES FOR SE_SSSP (TIMES FOR CLUSTERING WITH BFS PHASE I, SENDING WEIGHTS TO EDGES, 2 X SSSP) ARE REPORTED IN BRACKETS.

PAR_APPROX vs. SPAN vs. EM_BFS_DSLB: If the quality of the diameter approximation is most important then EM_BFS_DSLB could be the first choice. EM_BFS_DSLB produced reasonable results for all three graph classes sk-2005, $\Theta(\sqrt{n})$-level, and $\Theta(n)$ level. It would have done so for the (k_1, k_2, k_3) worst-case graph too, since that graph is just a tree. But if the running time is also important then rather PAR_APPROX should be chosen. It is faster than EM_BFS_DSLB in each tested case (sometimes more than a factor of ten) and its diameter approximation for each graph class was typically rather close – except for the carefully constructed worst-case graph. The spanning tree heuristic, however, could not convince in this test.

V. CONCLUSION

Our experiments have shown that the parametrized diameter approximation method is in fact faster than plain external-memory BFS and typically produces much better approximation bounds than the theory predicts. Nevertheless, it turns out that it is currently not suited as a section guide between different BFS approaches: as soon as the condensed graph does not fit into main memory, the overhead to run the semi-external memory SSSP is not worth the subsequent savings of a carefully chosen BFS approach. In fact, this is mostly a problem of the current interface in our SE_SSSP implementation, which causes some extra sorting steps for data conversion. For the future we plan to improve the SE_SSSP code in order to avoid these losses.

We are also interested in more sophisticated methods to condense the input graph. Currently, more master vertices speed-up the reduction time but result in a condensed graph that typically does not fit into main-memory, thus causing more I/O for the subsequent SSSP. Hierarchical clustering seems to be the natural choice but as the condensed graphs become weighted already after the first round, the parallel cluster growing of the next rounds needs to appropriately handle these weights, too. Currently, this step relies on the fact that the edges are unweighted.

REFERENCES

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

[2] U. Brandes and T. Erlebach, Eds., *Network Analysis: Methodological Foundations*, ser. LNCS, vol. 3418. Springer, 2005.

[3] D. A. Bader and K. Madduri, "Snap, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks," in *Proc. 22nd IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 2008, pp. 1–12.

[4] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. G. Chavarría-Miranda, "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," in *Proc. 23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 2009, pp. 1–8.

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

[6] D. Ajwani and H. Meyerhenke, "Realistic computer models," in *Algorithm Engineering*, ser. LNCS, vol. 5971. Springer, 2010, pp. 194–236.

[7] T. M. Chan, "All-pairs shortest paths for unweighted undirected graphs in *o(mn)* time," in *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms SODA*, 2006, pp. 514–523.

[8] R. Seidel, "On the all-pairs-shortest-path problem in unweighted undirected graphs," *Journal of Computer and System Sciences*, vol. 51, no. 3, pp. 400–403, 1995.

[9] Y. Peres, D. Sotnikov, B. Sudakov, and U. Zwick, "All-pairs shortest paths in $o(n^2)$ time with high probability," in *FOCS*, 2010, pp. 663–672.

[10] K. Boitmanis, K. Freivalds, P. Ledins, and R. Opmanis, "Fast and simple approximation of the diameter and radius of a graph," in *Proc. 5th WEA*, ser. LNCS, vol. 4007. Springer, 2006, pp. 98–108.

[11] 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.

[12] 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.

[13] 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)*, ser. LNCS, vol. 6346. Springer, 2010, pp. 302–313.

[14] U. Meyer, P. Sanders, and J. Sibeyn (Eds.), *Algorithms for Memory Hierarchies*, ser. LNCS. Springer, 2003, vol. 2625.

[15] J. S. Vitter, "Algorithms and data structures for external memory," *Foundations and Trends in Theoretical Computer Science*, vol. 2, no. 4, pp. 305–474, 2006.

[16] L. Arge, U. Meyer, and L. Toma, "External memory algorithms for diameter and all-pairs shortest-paths on sparse graphs," in *Proc. 31st ICALP*, ser. LNCS, vol. 3142. Springer, 2004, pp. 146–157.

[17] R. Chowdury and V. Ramachandran, "External-memory exact and approximate all-pairs shortest-paths in undirected graphs," in *Proc. 16th Ann. Symposium on Discrete Algorithms (SODA)*. ACM-SIAM, 2005, pp. 735–744.

[18] 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)*, ser. LNCS, vol. 2461. Springer, 2002, pp. 723–735.

[19] J. Abello, A. Buchsbaum, and J. Westbrook, "A functional approach to external graph algorithms," *Algorithmica 32(3)*, pp. 437–458, 2002.

[20] L. Arge, G. Brodal, and L. Toma, "On external-memory mst, sssp and multi-way planar graph separation," *J. Algorithms*, vol. 53, no. 2, pp. 186–206, 2004.

[21] U. Meyer and N. Zeh, "I/O-efficient undirected shortest paths," in *Proceedings of the 11th Annual European Symposium on Algorithms (ESA'03)*, ser. LNCS, vol. 2832. Springer, 2003, pp. 434–445.

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

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

[24] 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)*, 2006, pp. 601–610.

[25] R. Dementiev, L. Kettner, and P. Sanders, "STXXL: Standard Template library for XXL data sets," *Software: Practice and Experience*, vol. 38, no. 6, pp. 589–637, 2008.

[26] 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)*. SIAM, 2009, pp. 85–96.