Design and Implementation of a Practical I/O-efficient Shortest Paths Algorithm

Ulrich Meyer * Vitaly Osipov †

Abstract

We report on initial experimental results for a practical I/O-efficient Single-Source Shortest-Paths (SSSP) algorithm on general undirected sparse graphs where the ratio between the largest and the smallest edge weight is reasonably bounded (for example integer weights in $\{1, \ldots, 2^{32}\}$) and the realistic assumption holds that main memory is big enough to keep one bit per vertex. While our implementation only guarantees average-case efficiency, i.e., assuming randomly chosen edge-weights, it turns out that its performance on real-world instances with non-random edge weights is actually even better than on the respective inputs with random weights.

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

We are not aware of any previous I/O-efficient implementation for the classic general SSSP in a (semi) external setting: in two recent projects [10, 23], Kumar/Schwabe-like SSSP approaches on graphs of at most 6 million vertices have been tested, forcing the authors to artificially restrict the main memory size, M, to rather unrealistic 4 to 16 MBytes in order not to leave the semi-external setting or produce huge running times for larger graphs: for random graphs of 2^{20} vertices, the best previous approach needed over six hours. In contrast, for a similar ratio of input size vs. M, but on a 128 times larger and even sparser random graph, our approach was less than seven times slower, a relative gain of nearly 20. On a real-world 24 million node street graph, our implementation was over 40 times faster. Even larger gains of over 500 can be estimated for random line graphs based on previous experimental results for Munagala/Ranade-BFS. Finally, we also report on early results of experiments in which we replace the hard disk by a solid state disk (flash memory).

*Institute for Computer Science, Goethe University, 60325 Frankfurt/Main, Germany. Email: umeyer@cs.uni-frankfurt.de Partially supported by the DFG grant ME 3250/1-1, and by MADALGO - Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

†Universität Karlsruhe (TH), Fakultät für Informatik, 76128 Karlsruhe, Germany. Email: osipov@ira.uka.de Partially supported by the DFG grant SA 933/3-1.

1 Introduction

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

The classical SSSP-algorithm for general graphs is Dijkstra's algorithm [14]. Unfortunately, it performs poorly on massive graphs that do not fit into the main memory and are stored on disk. The reason is that Dijkstra's algorithm accesses the data in an unstructured fashion.

Much recent work has focused on algorithms for massive graphs, see [19, 27] for surveys. These algorithms are analyzed in the I/O-model [3], which assumes that the computer has a main memory that can hold M vertices or edges and that the graph is stored on disk. In order to process the graph, pieces of it have to be loaded into memory, which happens in blocks of B consecutive data items. Such a transfer is referred to as an *I/O-operation* (I/O). The complexity of an algorithm is the number of I/Os it performs; e.g., $\mathrm{sort}(N) = O((N/B)\log_{M/B}(N/B))$ I/Os to sort N numbers [3].

Previous results and related work. Little is known about solving SSSP on *directed* graphs I/O-efficiently. For *undirected* graphs, the algorithm of Kumar and Schwabe (KS_SSSP) [17] performs $O(n + (m/B)\log(n/B))$ I/Os. For dense graphs, the second term dominates; but for sparse graphs, the I/O-bound becomes $O(n)$. The SSSP-algorithm of Meyer and Zeh (MZ_SSSP) [20] extends the ideas of [18] for breadth-first search (BFS) to graphs with edge lengths between 1 and K, leading to an $O(\sqrt{nm\log K/B} + \mathrm{MST}(n, m))$ bound, where $\mathrm{MST}(n, m)$ is the cost of computing a

minimum spanning tree.[1] Recently [21], the result was further improved to $O(\sqrt{nm/B}\log n + \mathrm{MST}(n,m))$ I/Os, thus removing MZ_SSSP's dependence on the edge lengths in the graph. However, the latter approach is extremely involved and would probably suffer from very high constant factors in any realistic implementation setting.

When it comes to recent *internal-memory* SSSP implementations, the 9th DIMACS implementation challenge [1] provides a good overview. As for external-memory SSSP algorithms, to the best of our knowledge, none of the $o(n)$-I/O SSSP algorithms has ever been tried out. However, there are two recent papers [10, 23] reporting on external-memory experiments for KS_SSSP like approaches. Unfortunately, all results are for graphs of at most 6 million vertices, forcing the authors to artificially restrict the usably main memory size to rather unrealistic 4 to 16 MB in order not to leave the (semi-)external setting. Even then, computing SSSP for a random graph with $n \simeq 10^6$ vertices in the best case takes over 6 hours [23], which is more time than needed to do n I/Os.

Furthermore, using the I/O-library STXXL [12], Ajwani et al. [4, 6] studied implementations of external-memory BFS, i.e., the unweighted version of SSSP. They managed to compute BFS on different kinds of undirected graphs featuring over 250 million nodes and more than a billion of edges in less than 24 hours.

Another line of related research is algorithms for point-to-point shortest-path queries in (semi-)external memory using compression and extensive precomputation in internal memory. Typical representatives are, e.g., [9, 16, 25]. The success of these approaches crucially depends on the special characteristics of the input graphs (in particular road networks). In contrast we are interested in I/O-efficient general purpose SSSP computation without any structural assumptions on the input graph.

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

However, the performance of our STXXL [12] based implementation revealed just the opposite behavior: executed on real-world graphs with original non-random weights it was actually faster than on the same graphs with artificially assigned random weights.

While previous implementation studies [10, 23] for (semi-)external Kumar/Schwabe [17] kind SSSP approaches dealt with graphs having at most six million vertices, our study covers graphs of up to 250 million vertices and a billion edges. For random graphs of $n = 2^{20}$ vertices and $m = 8 \cdot n$ edges, the best previous approach needed over six hours. In contrast, for a similar ratio of $(n+m)/M$, but on larger and sparser random graphs of $n = 2^{28}$ vertices and $m = 4 \cdot n$ edges, our approach was less than seven times slower, a relative gain of nearly 20. On a real-world 24 million node street graph, our implementation was over 40 times faster. Even larger gains of over 500 can be estimated for random line graphs based on previous experimental results [6] for Munagala/Ranade-BFS [22].

2 Design and Implementation

Overview. Our SSSP approach is an I/O-efficient version of Dijkstra's algorithm [14]. Dijkstra uses a priority queue Q to store all vertices of G that have not been settled yet (a vertex is said to be *settled* when its final distance from s has been determined); the priority of a vertex v in Q is the length of the currently shortest known path from s to v. Vertices are settled one-by-one by increasing distance from s. The next vertex v to be settled is retrieved from Q using a delete_min operation. Then the algorithm relaxes the edges between v and all its non-settled neighbors, that is, performs a decrease_key$(w, dist(s,v) + c(v,w))$ operation for each such neighbor w whose priority is greater than $dist(s,v) + c(v,w)$.

An I/O-efficient version of Dijkstra's algorithm has to (a) avoid accessing adjacency lists at random, (b) deal with the lack of optimal decrease_key operations in current external-memory priority queues, and (c) efficiently remember settled vertices. Since we allow

[1] The current bounds for $\mathrm{MST}(n,m)$ are $O(\mathrm{sort}(m)\log\log(nB/m))$ [7] deterministically and $O(\mathrm{sort}(m))$ randomized [11].

ourselves one bit per node in internal memory problems (b) and (c) are easily solved. As for (c) the bit vector is used to keep track which vertices have been visited. Concerning (b) we allow up to degree(v) many entries for a vertex v in the priority-queue at the same time and when extracting them discard all but the first one with the help of the bit vector. As for (a) our approach forms clusters of vertices just like the EM-BFS algorithm of Mehlhorn and Meyer[18] (i.e., without considering the edge weights at all) and loads the adjacency lists of all vertices in a cluster into a number of "hot pools" of edges as soon as the first vertex in the cluster is settled. For integer edge weights from $\{1, \ldots, K\}$ we have $k = \lceil \log_2 K \rceil$ pools, where the i-th pool is reserved for category i edges, that is, edges of weight between 2^{i-1} and $2^i - 1$.

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

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

Graph Data Structure. Boost libraries [2] are considered as the next level of standardization over Standard Template Library (STL for short). Unfortunately, even though the Boost Graph Library (BGL for short) includes several graph classes, such as adjacency list or adjacency matrix, missing guaranties on the layout of edges on the hard drive make them inapplicable for I/O efficient algorithms. Therefore, we have implemented our own I/O-efficient graph representation that conforms to the BGL interface, thus providing the same level of generality.

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

Priority Queue. We store nodes with their tentative distances in the I/O efficient priority queue being part of the STXXL library. Each of its operations takes $O(1/B \log_{M/B} I/B)$ I/O amortized, where I denotes the total number of insertions [24]. Note that we may keep several entries with different priorities for some vertices at the same time.

Pipelining. Our implementation intensively uses pipelining. Conceptually, pipelining is a partitioning of the algorithm into practically independent parts that conform to a common interface, so that the data can be streamed from one part to the other without any intermediate external-memory storage. This way the I/O complexity may be reduced by up to a constant factor. Moreover, it also leads to a better structured implementation, while different parts of the pipeline only share a narrow common interface. On the other hand, the price one sometimes has to pay is higher computational costs and potentially somewhat larger debugging efforts. For more details on pipelining in the framework of I/O efficient algorithms, see [12].

Deterministic graph clustering. In the deterministic preprocessing we compute a spanning tree for the connected component containing the source node, obtain an Euler tour around that spanning tree, and eventually form the clusters based on subsequences of the Euler tour (generated by list-ranking and sorting). We apply the external-memory deterministic preprocessing implementation by Ajwani et al. [6], which in turn uses a spanning forest and connected components implementation by Dementiev et al. [13] with expected $sort(m)\lceil \log n/M \rceil$ I/O runtime [13]. Furthermore, they use an adaptation of Sibeyn's list ranking algorithm [26]. Both implementations are based on STXXL data structures and its sorting primitive. For more details on the deterministic preprocessing, refer to [6].

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

Then we check (point 2) if there are any previously settled nodes in some $relax\_i$, whose incident edges

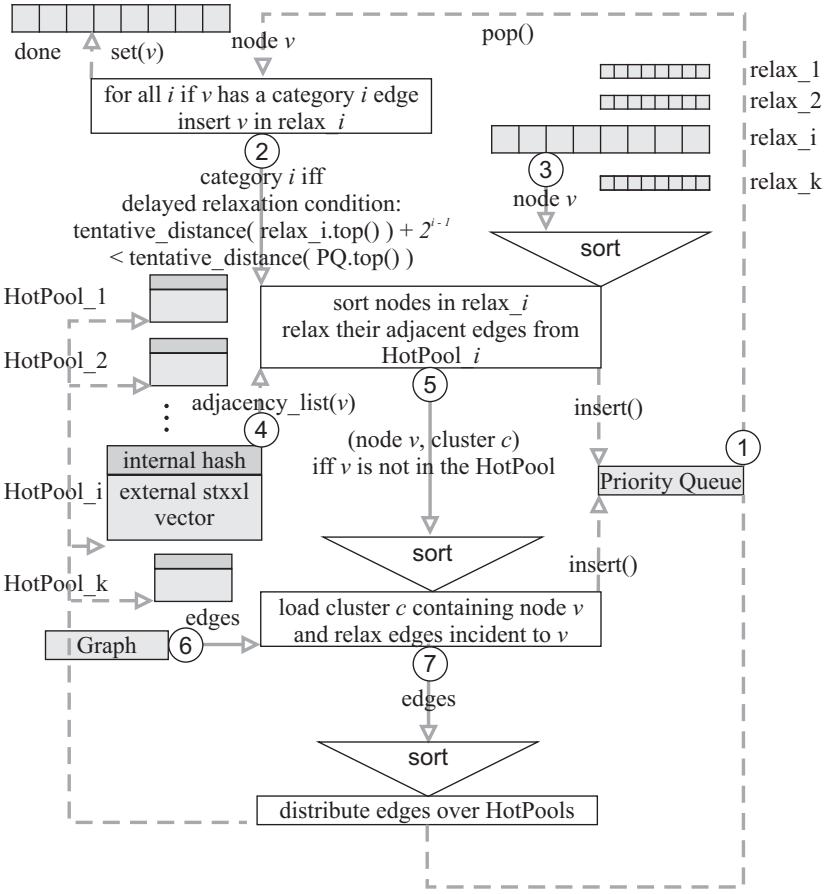

Figure 1: Flow-chart of the pipelined SSSP phase implementation. The empty elements of the pipeline conform to the common pipelining interface, while the solid lines denote the data stream through the pipeline. Shaded elements represent non-pipelined data structures with the dashed lines denoting the data exchange through their auxiliary methods. The numbered circles reflect the order in which the elements flow through the pipeline.

have to be relaxed before settling the next node at the top of the priority queue. Thus, we check the *delayed relaxation condition* in Figure 1 for the oldest node within each $relax\_i$ queue. Observe, that this is sufficient, since the distances associated with the elements of any of $relax\_i$ starting from its oldest element do not decrease.

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

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

A heuristic for maintaining the pool. The asymptotic improvement and performance gain in MZ_SSSP as compared to KS_SSSP is due to the partitioning of the input graph into the clusters and maintaining an efficiently accessible graph cache (hot pools) of adjacency lists, which are guaranteed to be requested soon after. Thus, efficient access patterns to the hot pools are crucial for the performance of MZ_SSSP.

Ajwani et al. [6] observed, that in the case of BFS for many large diameter graphs, the pool fits into the

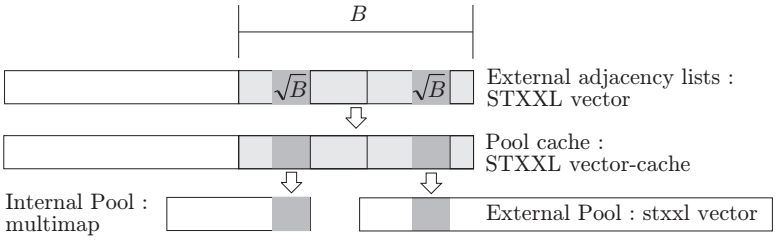

Figure 2: Scheme of the heuristic to partially keep hot pools in internal memory.

internal memory most of the time. They proposed maintaining it partially in an internal memory hash table, thus using efficient dictionary look up instead of computationally quite expensive scanning of all hot pool edges. Besides that, they observed that when the clusters are small enough ($O(\sqrt{B})$ for line graphs), it is worth caching all neighboring clusters that are anyway loaded into the main memory while reading B elements from the disk. The last fact is due to the special layout of clusters the deterministic preprocessing produces. As for implementation, they store adjacency lists in the STXXL vector, thus, loading the neighboring clusters in its internal memory cache using an LRU replacement strategy, see Figure 2 (in the appendix). This heuristic approach appeared to be particularly efficient for medium and large diameter grid and line graphs, see [6].

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

While Ajwani et al. had only *one* hot pool, we have k hot pools for k different categories of edges. As well as in the BFS case, we use a multi-map hash table to maintain $O(M)$ edges internally. Observe, that due to the relaxation condition, Figure 1, hot pools with the low category edges are likely to be requested more often than those of higher categories. Thus, it is worthwhile reserving more internal memory for the smaller category hot pools. For the comparative study of different memory allocation strategies refer to Section 3. As for the caching of neighboring clusters, we use the same technique as in [6] to benefit from the special cluster disk layout produced by the deterministic preprocessing, see to Figure 2 (in the appendix).

3 Experiments

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

Our experimental platform has two 2.0 GHz Opteron processors, 4 GB of RAM, 1 MB cache and 250 GB Seagate Baracuda hard disks. The hard drive buffer cache is 8 MB big. The average seek time for read and write is 8.0 and 9.0 msec respectively. The data transfer rate for outer zone (maximum) is 65 MByte/s. Therefore, for a graph with 2^{28} nodes n random read and write I/Os would take around 600 and 675 hours, respectively.

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

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

While being one of the most popular applications, SSSP on road networks is not necessarily the best illustration for our algorithm due to the following reasons: (1) even the European road network is rather small for realistic external-memory settings; (2) the special structure of road networks allows recent specialized approaches to outperform the general purpose Dijkstra algorithm by several orders of magnitude, e.g., see [9, 16]. Although no theoretical I/O bounds are given, the algorithm in [16] has been designed with the explicit goal of being efficient on devices with small internal memory and slow storage memories (e.g., flash memories) such as pocket PCs. Similarly, in recent work Sanders et al. [25] propose a highly efficient algorithm for point-to-point shortest path queries on mobile devices.

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

[2] provided for scientific use by Ortec company.

Web graph. As an instance of real world graphs we also consider a crawl of the world wide web [28]. The nodes of the web graph represent internet pages, while the edges correspond to the links between them. Our instance of the web graph has around 135 million nodes and 1.2 billion edges. Structurally the web graph is close to a random graph, with a small fraction of larger diameter branches. Therefore, the I/O runtime is similar to the one for random graphs.

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

Comparing BFS and SSSP. We compared our SSSP implementation against Ajwani et al.'s BFS implementation of [6]. The result in Table 1 indicates, that while Ajwani et al. perform BFS traversal for any of the graph classes within *one* day, we compute SSSP for the same graph classes with 16 and 32 bit random weights within just *two* days. Our SSSP approach was never slower than a factor of five, while for the most difficult graph class (grids) the difference was even less than a factor of two.

Comparing KS_SSSP and MZ_SSSP. If we try to relate different SSSP algorithms with their BFS counterparts, then KS_SSSP and the external-memory BFS algorithm by Munagala and Ranade (MR_BFS for short) [22] share similar ideas (and access patterns), whereas MZ_SSSP corresponds to Mehlhorn and Meyer's BFS algorithm (MM_BFS for short).

Ajwani et al. [6] showed that MR_BFS outperforms MM_BFS for low diameter graphs, such as random or web graphs, while medium and large diameter graph instances become practically infeasible for it (hours as opposed to months for line graphs). As for KS_SSSP and MZ_SSSP, we expect the later one to significantly outperform the former for the *whole* range of graphs that we consider. The reason for it is due to the incremental nature of Dijkstra's algorithm. Indeed, while MR_BFS extracts adjacency lists in a batched fashion level by level, KS_SSSP loads edges incident to the settled vertices consecutively vertex by vertex. Therefore, for the expected $O(\log n)$ levels of a random graph MR_BFS spends on average $O(n/B \log n)$ I/Os, while KS_SSSP requires one I/O per vertex, thus exhibiting worst case $\Omega(n)$ I/O performance in practice.

This observation is in line with the recent implementation of a Kumar/Schwabe-like approach by Sach and Clifford [23], who used a cache oblivious priority queue and an internal-memory bit array like us in our approach. They observed in practice that for random graphs the I/O complexity for extracting adjacency lists was a dominating factor over maintaining the priority queue.

Moreover, as it is shown in Table 2 the performance of their algorithm on the real world road networks also significantly depends on the layout of edges. As we already mentioned above, available real world road network instances initially incorporate spatial locality, thus facilitating more efficient adjacency lists extraction. Therefore, for original vertex numbering the runtime of their implementation only slightly depends on the internal memory available for the system. However, a random permutation of vertices has a significant impact on the performance, thus showing overwhelming dependence of the runtime on the layout of adjacency lists on a disk. On contrary, the runtime of our SSSP algorithm in Table 2 barely depends on the original vertex indices, which is a desirable feature for a general purpose SSSP solver.

As for large diameter graphs, Ajwani et al. [6] showed that MM_BFS drastically outperforms MR_BFS for random line graphs. Since the I/O performance for MR_BFS constitutes a lower bound for KS_SSSP, we directly compare the MR_BFS results from [6] with our SSSP approach in order to estimate an advantage of more than a factor of 500, see Table 4.

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

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

Delayed relaxation of edges. As we already discussed in Section 2, the delayed relaxation condition in Figure 1 allows postponing relaxation of longer edges. For each edge category we measured the number of batched relaxations of nodes having incident edges in this category, and compared it to the number of relaxations that would have been performed without the relaxation condition in use. In the series of diagrams

Graph class	n	m	BFS	SSSP
Random (16 bit)	2^{28}	2^{30}	8.6	36.0
Random (32 bit)	2^{28}	2^{30}	8.6	39.2
Grid ($2^{14} \times 2^{14}$, 16 bit)	2^{28}	2^{29}	21.0	33.6
Grid ($2^{14} \times 2^{14}$, 32 bit)	2^{28}	2^{29}	21.0	37.6
Random line (16 bit)	2^{28}	2^{28}	3.7	7.6
Webgraph (32 bit)	$\approx 135 \times 10^6$	$\approx 1.18 \times 10^9$	5.7	28.7

Table 1: Timing in hours for the currently best BFS implementation vs. our SSSP approach (both including preprocessing).

Node indices	$n/10^6$	$m/10^6$	RAM	SSSP by [23]		SSSP phase		Preprocessing	
				I/O wait	Total	I/O wait	Total	I/O wait	Total
original	24	29	1024	4550	4964	155	1414	420	547
original	24	29	512	4848	5222	191	1449	484	614
original	24	29	128	5059	5444	2815	4059	956	1123
permuted	24	29	2048	209350	209873	136	1417	428	589
permuted	24	29	1024	*	*	175	1458	455	611
permuted	24	29	512	*	*	187	1474	529	685
permuted	24	29	128	*	*	2892	4158	951	1139

Table 2: Timing in *seconds* for US road network with *original* or *permuted* node indices and original edge weights using RAM in *megabytes*. Fields marked with * are omited due to high computation cost.

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

Table 3: Timing in *seconds* for European road network with *original* or *permuted* node indices and *original* or *32-bit* random edge weights using 128 MB of RAM.

Graph class	n	m	MR_BFS [4]	SSSP
Random line	2^{28}	2^{30}	4760	7.6

Table 4: Timing in hours for MR_BFS and SSSP (including preprocessing, for *16-bit* random edge weights).

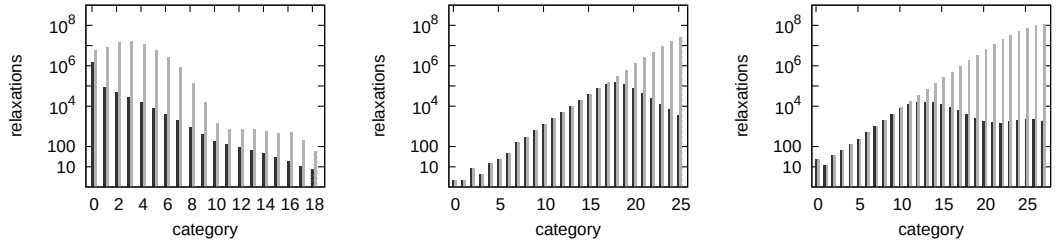

Figure 3: The comparison between the number of batched relaxations (black) with the total number of relaxations (gray) in logarithmic scale. From left to right: European road network graph with original edges, with random 32 bit weights and web graph with 32 bit random weights

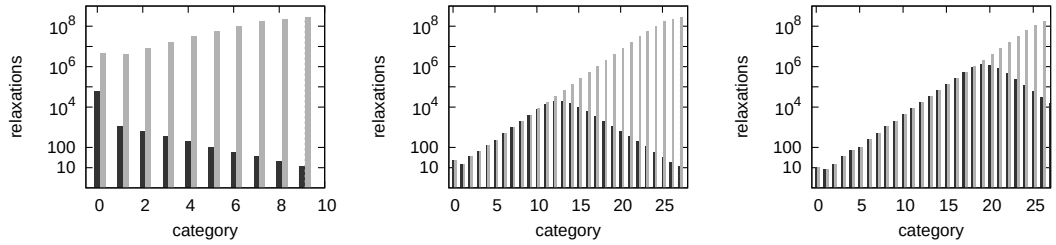

Figure 4: The comparison between the number of batched relaxations (black) with the total number of relaxations (gray) in logarithmic scale. From left to right: random graph with 16 bit random weights, random graph with 32 bit random weights and random grid graph with 32 bit random weights

3 and 4 we compare the number of batched relaxations (in black) with the their overall number (in gray) in logarithmic scale.

Note, that in case of the European road network with real distances, delayed relaxation is even more beneficial than for 32 bit random weights on the same graph (compare first and second histograms in Figure 3). Even in the first, most notable category, the number of batched relaxations is around 20% of the overall number. Thus, on average the algorithm relaxes a batch of first category edges incident to five different nodes at once. In the next categories the ratio drops to at most 1%, that is, on average at least 100 nodes at once. The higher ratio for the last few categories is only due to the low number of long distances in the road network.

On the contrary, the same road network graph with 32 bit random weights (second histogram in Figure 3) demonstrates a small ratio only for the upper half of the categories, while in the first categories the relaxations are performed in a one-by-one manner. This in turn leads to a significant performance loss, refer to Table 3 for the exact timing.

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

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

For the web graph (last histogram in Figure 3) and the random graph with 32 bit random weights (second column in Figure 4) we see similar behavior: one-by-one relaxations in the lower categories, and rapidly decreasing ratio in the higher categories.

As a rule of thumb for graphs with random edge weights, the bigger the diameter, the larger the number of categories, where relaxations have to be performed in a one-by-one fashion. For instance, for the random graph with 32 bit edge weights the ratio drops significantly already for the categories $14 - 15$, while for the grid this value can be found around the $18 - 19$th category (compare the second and the third histograms in Figure 4). The most extreme case is the line graph, where essentially all relaxations have to be performed consecutively one after the other.

Quality of the spanning tree. Ajwani et al. [6] observed that the shape of the spanning tree in the

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

Table 5: Quality of the spanning tree.

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

Table 6: Different memory allocation strategies for the heuristic.

preprocessing step plays an important role for the quality of the clustering. In line with [6] for the grid graph we observed a considerable improvement in the SSSP phase I/O runtime when the spanning tree is "randomized". The reason for it is, that a spanning tree with elements in a snake-like row major order produces long and narrow clusters, while a "random" one is more likely to result in low diameter clusters. The former clusters tend to stay in the hot pools longer, hence, increasing their sizes, that eventually results in a larger I/O volume for storing hot pools and for re-scanning them while retrieving adjacency lists. On the other hand, the latter ones are evicted from the hot pools sooner, thus reducing I/O runtime. Most notably, for a simple grid graph with random 16 bit weights, clustering with the randomized input of the spanning tree algorithm gives about 30% runtime improvement over the unrandomized one, Table 5.

Different memory allocation strategies for the heuristic. The delayed relaxation condition for random edge weights, Figure 1, implies that edges in the lower categories are relaxed more often than those in the higher categories. This suggests, that in general the hot pools storing the lower category edges should get more internal memory than those storing the higher category edges.

In most of our experiments, Table 1 in particular, we used common 128 MB of RAM for all hot pool caches and 64 MB for the adjacency list vector cache, see Figure 2. By default the overall 128 MB of RAM are split among the hot pools such that the category i hot pool receives only half of the memory that is available for the category $i-1$ hot pool. We call this strategy "decreasing".

As it is indicated in Figures 3 and 4, the number of categories, where relaxations have to be performed consecutively in a one by one fashion, increase with growing diameter. Therefore, for the middle range diameter grid graph and large diameter line graph it is worth distributing available memory equally throughout all hot pools. This strategy is denoted as "uniform".

For the random graph with 16 bit weights "decreasing" shows better results, since the bulk of batched relaxations is performed in the low level category hot pools, see Figure 4. As for the larger diameter grid and line graphs "uniform" appears to be the best choice. The reason for it is that due to the regular structure and small degree of these graph classes there are not that many (only one in case of line graph) paths between any two nodes, meaning that the algorithm needs to load edges quite often even from high category hot pools. This is in line with observation in Section 2 (Figure 4), that in case of grid (and especially line) graphs only nodes having high category edges are relaxed in a batched manner, while in the low categories nodes need to be relaxed basically one by one.

4 Early results on flash memory.

In very recent work we also performed preliminary tests of our SSSP implementation on modern flash memory also known as solid state disks (SSDs). These are non-volatile, reprogrammable memories, which have emerged as a new trend in storage device technology. Flash memory devices are lighter, more shock resistant and consume less power. Moreover, since random read accesses are faster on solid state disks compared to traditional mechanical hard-disks, flash memory is fast

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

Table 7: Preliminary results on flash memory.

becoming the dominant form of end-user storage in mobile computing. Market research company In-Stat predicted in July 2006 that 50% of all mobile computers would use flash (instead of hard disks) by 2013.

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

Previous and related work on flash. Most previous algorithmic work on flash memories deals with wear leveling, block-mapping and flash-targeted file systems (see [15] for a comprehensive survey). There exists very little work on algorithms designed to exploit the characteristics of flash memories. Wu et al. [29, 30] proposed flash-aware implementations of B-trees and R-trees without file system support by explicitly handling block-mapping within the application data structures. Other works include the use of flash memories for model checking [8] or for route planning (point-to-point shortest paths) on mobile devices [16, 25].

An adaptation of our previously mentioned EM-BFS implementation for flash memory was discussed in [5]. In there, a 32 GB Hama SSD (2.5" IDE) was used. Due to limited bandwidth of this device (less than 30 MB/s) only a combination of flash plus a traditional hard disk (in that case a 500 GB SEAGATE Barracuda 7200.11) was more profitable than using the hard disk alone: the small read-only graph clusters reside on flash from where they can be retrieved using fast random reads, whereas the hot pool with its frequent sequential rewriting of large data sequences stays on the hard disk in order to profit from higher throughput.

Preliminary results. We performed experiments on a newer machine featuring an Intel Quad Core Q6600 CPU, 8GB of RAM, a fast hard drive and a solid state disk. We used the gcc compiler 4.3.2 with optimization level O3 on a Debian Linux distribution and STXXL version 1.2.2. The available internal memory was restricted to at most 1 GB and only one processor was used. We observed that the performance of solid state disks has significantly improved over the last year: priced similarly as the 32 GB device purchased for [5] a year ago, our current 64 GB Hama SSD (3.5" SATA) not only offers double the capacity but also features significantly increased throughput of measured 84 MB/s (reading), and 75 MB/s (writing). Although our 500 GB SEAGATE Barracuda 7200.11 hard disk applied in these experiments still offers higher throughput (about 100 MB/s for sequential access of large blocks), now for medium diameter grid and large diameter random graphs even using a SSD alone results in faster SSSP execution, see Table 7. On the other hand, for random line graphs, the SSSP phase using the hard drive benefits from a larger throughput and sequential reading speed. Indeed, as observed in [6], for line graphs, the Euler-tour based preprocessing lays out the clusters on external memory storage in a way, that the clusters that are visited soon after each other during a BFS traversal are located sequentially, thus facilitating sequential reading. While the node visiting order for BFS and SSSP traversals may differ significantly in general, it is very similar for line graphs.

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

5 Conclusions

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

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

References

[1] Online resources of the 9th DIMACS Implementation Challenge: Shortest Paths, 2006. `http://www.dis.uniroma1.it/~challenge9/`.

[2] *The boost graph library: user guide and reference manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

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

[4] D. Ajwani, R. Dementiev, and U. Meyer. A computational study of external-memory BFS algorithms. In *SODA*, pages 601–610. ACM Press, 2006.

[5] D. Ajwani, I. Malinger, U. Meyer, and S. Toledo. Characterizing the performance of flash memory storage devices and its impact on algorithm design. In *Proc. 7th Int. Workshop on Experimental Algorithms (WEA)*, volume 5038 of *Lecture Notes in Computer Science*, pages 208–219. Springer, 2008.

[6] D. Ajwani, U. Meyer, and V. Osipov. Improved external memory BFS implementation. In *Proc. Workshop on Algorithm Engineering and Experiments, ALENEX*. SIAM, 2007.

[7] L. Arge, G. S. Brodal, and L. Toma. On external-memory MST, SSSP and multi-way planar graph separation. *J. Algorithms*, 53(2):186–206, 2004.

[8] J. Barnat, L. Brim, S. Edelkamp, D. Sulewski, and P. Šimeček. Can flash memory help in model checking? In *Proc. 13th International Workshop on Formal Methods for Industrial Critical Systems*, pages 159–174, 2008.

[9] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In transit to constant time shortest-path queries in road networks. In *Proc. Workshop on Algorithm Engineering and Experiments, ALENEX*. SIAM, 2007.

[10] M. Chen, R. A. Chowdhury, V. Ramachandran, D. L. Roche, and L. Tong. Priority queues and Dijkstra's algorithm. Technical Report TR-07-54, The University of Texas at Austin, Department of Computer Sciences, Oct. 2007.

[11] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *SODA*, pages 139–149, 1995.

[12] R. Dementiev, L. Kettner, and P. Sanders. : Standard template library for XXL data sets. In G. S. Brodal and S. Leonardi, editors, *ESA*, volume 3669 of *Lecture Notes in Computer Science*, pages 640–651. Springer, 2005.

[13] R. Dementiev, P. Sanders, D. Schultes, and J. F. Sibeyn. Engineering an external memory minimum spanning tree algorithm. In J.-J. Lévy, E. W. Mayr, and J. C. Mitchell, editors, *IFIP TCS*, pages 195–208. Kluwer, 2004.

[14] E. W. Dijkstra. A note on two problems in connexion with graphs. *Num. Math.*, 1:269–271, 1959.

[15] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2):138–163, 2005.

[16] A. Goldberg and R. Werneck. Computing point-to-point shortest paths from external memory. In *Proc. 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*. SIAM, 2005.

[17] V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *SPDP*, pages 169–176, Los Alamitos, CA, USA, 1996. IEEE Computer Society.

[18] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In R. H. Möhring and R. Raman, editors, *ESA*, volume 2461 of *Lecture Notes in Computer Science*, pages 723–735. Springer, 2002.

[19] U. Meyer, P. Sanders, and J. Sibeyn (Eds.). *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*. Springer, 2003.

[20] U. Meyer and N. Zeh. I/O-efficient undirected shortest paths. In G. D. Battista and U. Zwick, editors, *ESA*, volume 2832 of *Lecture Notes in Computer Science*, pages 434–445. Springer, 2003.

[21] U. Meyer and N. Zeh. I/O-efficient undirected shortest paths with unbounded edge lengths. In Y. Azar and T. Erlebach, editors, *ESA*, volume 4168 of *Lecture Notes in Computer Science*, pages 540–551. Springer, 2006.

[22] K. Munagala and A. G. Ranade. I/O-complexity of graph algorithms. In *SODA*, pages 687–694, 1999.

[23] B. Sach and R. Clifford. An empirical study of cache-oblivious priority queues and their application to the shortest path problem. Available online under `http://www.cs.bris.ac.uk/`

∼sach/COSP/, Feb. 2008.

[24] P. Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithms*, 5:7, 2000.

[25] P. Sanders, D. Schultes, and C. Vetter. Mobile route planning. In *Proc. 16th Annual European Symposium on Algorithms*, volume 5193 of *LNCS*, pages 732–743. Springer, 2008.

[26] J. F. Sibeyn. From parallel to external list ranking. Technical report, Max Planck Institut für Informatik, Saarbrücken, Germany, 1997.

[27] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM computing Surveys, 33*, pages 209–271, 2001. Revised version (April 2008) available online at `http://www.cs.purdue.edu/homes/∼jsv/Papers/Vit.IO/survey.pdf`.

[28] The stanford webbase project. `http://www-diglib.stanford.edu/∼testbed/doc2/WebBase/`.

[29] C.-H. Wu, L.-P. Chang, and T.-W. Kuo. An efficient R-tree implementation over flash-memory storage systems. In *Proc. 11th ACM International Symposium on Advances in Geographic Information Systems*, pages 17–24, 2003.

[30] C.-H. Wu, T.-W. Kuo, and L.-P. Chang. An efficient B-tree layer implementation for flash-memory storage systems. *ACM Transactions on Embedded Computing Systems*, 6(3), 2007.