Breadth first search on massive graphs*

Deepak Ajwani † Roman Dementiev ‡ Ulrich Meyer † Vitaly Osipov †

Abstract

We consider the problem of Breadth First Search (BFS) traversal on massive sparse undirected graphs. Despite the existence of simple linear time algorithms in the RAM model, it was considered non-viable for massive graphs because of the I/O cost it incurs. Munagala and Ranade [29] and later Mehlhorn and Meyer [27] gave efficient algorithms (refered to as MR_BFS and MM_BFS, respectively) for computing BFS level decompositions in an external memory model. Ajwani et al. [3] implemented MR_BFS and the randomized variant of MM_BFS using the external memory library STXXL and gave a comparative study of the two algorithms on various graph classes. In this paper, we review and extend that result demonstrating the effectiveness and viability of the BFS implementations on various other synthetic and real world benchmarks. Furthermore, we present the implementation of the deterministic variant of MM_BFS and show that in most cases, it outperforms the randomized variant.

1 Introduction

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

Large graphs arise naturally in many applications and very often we need to traverse these graphs for solving the inherent optimization problems. Typical real-world applications of BFS on large graphs (and some of its generalizations like shortest paths or A^*) include crawling and analyzing the WWW [30, 32], route planning using small navigation devices with flash memory cards [22], state space exploration [20], etc. Since most of the large real world graphs are sparse, we mainly concentrate on the problem of computing a BFS level decomposition for massive sparse undirected graphs.

1.1 Computational model

Traditionally, the performance of algorithms has been analyzed in the RAM model, where it is assumed that there is an unbounded amount of memory with unit cost for accessing any location. Unfortunately, the predicted performance of algorithms on this model significantly deviates from their actual run-times for large graphs, which do not fit in the main memory and thus, have to be

*This work was partially supported by the DFG grants SA 933/1-3 and ME 2088/1-3.

†Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany. E-mail: {ajwani,umeyer,osipov}@mpi-inf.mpg.de

‡University Karlsruhe, Am Fasanengarten 5, 76131 Karlsruhe, Germany. Email: dementiev@ira.uka.de

stored on hard-disk(s). While modern processor speeds are measured in GHz, average hard disk latencies are in the range of a few milliseconds [23]. Hence, the cost of accessing a data element (an I/O) from the hard-disk is around a million times more than the cost of an instruction. Therefore, it comes as no surprise that the runtimes of even basic graph traversal strategies on these graphs are I/O dominant. Since the RAM model does not capture the I/O costs, we consider the commonly accepted external memory model by Aggarwal and Vitter [2]. It assumes a two level memory hierarchy with a fast internal memory and a slow external memory. We define M ($< n + m$) to be the number of vertices/edges that fit into internal memory, and B to be the number of vertices/edges that fit into a disk block. In an I/O operation, one block of data is transferred between disk and internal memory. The measure of performance of an algorithm is the number of I/Os it performs. The number of I/Os needed to read N contiguous items from disk is $\text{scan}(N) = \Theta(N/B)$. The number of I/Os required to sort N items is $\text{sort}(N) = \Theta((N/B)\log_{M/B}(N/B))$. For all realistic values of N, B, and M, $\text{scan}(N) < \text{sort}(N) \ll N$. Advanced models include parallel disks [36] or hide the parameters M and B from the algorithms (cache-oblivious model [21]). A comprehensive list of results for the I/O-model and memory hierarchies have been obtained – for recent surveys see [15, 28, 35] and the references therein.

1.2 Algorithms

BFS is well-understood in the RAM model. There exists a simple linear time algorithm [14] (hereafter refered as IM_BFS) for the BFS traversal in a graph. IM_BFS keeps a set of appropriate candidate nodes for the next vertex to be visited in a FIFO queue Q. Furthermore, in order to find out the unvisited neighbours of a node from its adjacency list, it marks the nodes as either visited or unvisited. Unfortunately as reported in [3], even when half of the graph fits in the main memory, the running time of this algorithm deviates significantly from the predicted RAM performance (*hours* as compared to *minutes*) and for massive graphs, such approaches are simply non-viable. As discussed before, the main cause for the catastrophic performance of this algorithm on massive graphs is the number of I/Os it incurs. Remembering visited nodes needs $\Theta(m)$ I/Os in the worst case and the unstructured indexed access to adjacency lists may result in $\Theta(n)$ I/Os.

The algorithm by Munagala and Ranade [29] (refered as MR_BFS) ignores the second problem but addresses the first by exploiting the fact that the neighbours of a node in BFS level i are all in BFS levels $i + 1$, i or $i - 1$. Thus, the set of nodes in level $i + 1$ can be computed by removing all nodes in level i and $i - 1$ from the neighbours of nodes in level i. The resulting worst-case I/O-bound is $O(n + \text{sort}(n + m))$.

Mehlhorn and Meyer suggested another approach [27] (MM_BFS) which involves a preprocessing phase to restructure the adjacency lists of the graph representation. It groups the vertices of the input graph into disjoint clusters of small diameter and stores the adjacency lists of the nodes in the cluster contiguously on the disk. Thereafter, an appropriately modified version of MR_BFS is run. MM_BFS exploits the fact that whenever the first node of a cluster is visited then the remaining nodes of this cluster will be reached soon after. By spending only one random access (and possibly, some sequential access depending on cluster size) in order to load the whole cluster and then keeping the cluster data in some efficiently accessible data structure (pool) until it is all used up, on sparse graphs the total amount of I/O can be reduced by a factor of up to $\sqrt{B}$. The neighouring nodes of a BFS level can be computed simply by scanning the pool and not the whole graph. Though some edges may be scanned more often in the pool, unstructured I/O in order to fetch adjacency lists is considerably reduced, thereby saving the total number of I/Os.

The preprocessing of MM_BFS comes in two variants: randomized and deterministic (refered as MM_BFS_R and MM_BFS_D, respectively). In the randomized variant, the input graph is partitioned by choosing master nodes independently and uniformly at random with a probability $\frac{1}{\sqrt{B}}$ and running a BFS like routine with joint adjacency list queries from these master nodes "in parallel".

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

The randomized variant incurs an expected number of $O(\sqrt{n \cdot (n+m) \cdot \log(n)/B} + \mathrm{sort}(n+m))$ I/Os, while the deterministic variant incurs $O(\sqrt{n \cdot (n+m)/B} + \mathrm{sort}(n+m)) + ST(n+m))$ I/Os, where $ST(n+m)$ is the number of I/Os required for computing a spanning tree of a graph with n nodes and m edges. Arge et al. [4] show an upper bound of $O((1+\log\log(D \cdot B \cdot n/m)) \cdot \mathrm{sort}(n+m))$ I/Os for computing such a spanning tree.

1.3 Related Work

Brodal et al. [8] gave a cache oblivious algorithm for BFS achieving the same worst case I/O bounds as MM_BFS_D. Their preprocessing is similar to that in MM_BFS_D, except that it produces a hierarchical clustering using the cache oblivious algorithms for sorting, spanning tree, Euler tour and list ranking. The BFS phase uses a data-structure that maintains a hierarchy of pools and provides the set of neighbours of the nodes in a BFS level efficiently.

Other external memory algorithms for BFS are restricted to special graphs classes like trees [11], grid graphs [5], planar graphs [26], outer-planar graphs [24], and graphs of bounded tree width [25].

Ajwani et al. [3] gave the first empirical comparison between MR_BFS and MM_BFS_R and concluded the following:

- Both the external memory BFS algorithms MR_BFS and MM_BFS_R are significantly better (*minutes* as compared to *hours*) than the RAM BFS algorithm, even if half the graph resides in the internal memory.
- The usage of these algorithms along with disk parallelism and pipelining can alleviate the I/O bottleneck of BFS on many large sparse graph classes, thereby making the BFS computation viable for these graphs. As a real world example, the BFS level decomposition of an external web-crawl based graph [37] of around 130 million nodes and 1.4 billion edges was computed in less than 4 hours using a single disk and 2.3 hours using four disks.
- MR_BFS performs better than MM_BFS_R on small-diameter random graphs saving a few *hours*. However, the better asymptotic worst-case I/O complexity of MM_BFS helps it to outperform MR_BFS for large diameter sparse graphs (computing in a few *days* versus a few *months*), where MR_BFS incurs close to its worst case of $\Omega(n)$ I/Os.

Independently, Frederik Christiani [13] gave a prototypical implementation of MR_BFS, MM_BFS_R as well as MM_BFS_D and reached similar conclusions regarding the comparative performance

between MR_BFS and MM_BFS_R. Some of the subroutines used in their algorithms are cache-oblivious. Their implementation of MR_BFS and MM_BFS_R is competetive and on some graph classes even better than [3]. Since their main goal was to design cache oblivious BFS, they used cache oblivious algorithms for sorting, minimum spanning tree and list ranking even for MM_BFS_D. As we discuss later, these algorithms slow down the deterministic preprocessing, even though they have the same asymptotic complexity as their external memory counterparts.

1.4 Our Contribution

Our new contributions in this paper are the following:

- We improve upon the MR_BFS and MM_BFS_R implementation described in [3] by reducing the computational overhead associated with each BFS level, thereby improving the results for large diameter graphs.
- We present the design and implementation of MM_BFS_D. This involved experimenting with various external memory connected component, spanning tree and list ranking algorithms used as a subroutine for the deterministic preprocessing.
- We conduct a comparative study of MM_BFS_D with other external memory BFS algorithms and show that for most graph classes, MM_BFS_D outperforms MM_BFS_R.
- We compare our BFS implementations with Christiani's implementations [13], which have some cache-oblivious subroutines. This gives us some idea of the loss factor that we will have in the performance of cache-oblivious BFS.
- We propose a heuristic for maintaining the pool in the BFS phase of MM_BFS. This heuristic improves the runtime of MM_BFS in practice, while preserving the worst case I/O bounds of MM_BFS.
- The graph generator tools have been extended to generate graphs in the DIMACS shortest path challenge format and our BFS implementations now support the DIMACS format as well.
- Putting everything together, we show that BFS traversals can also be done on moderate and large diameter graphs in a few *hours*, which would have taken the implementations of [3] and [13] several *days* and IM_BFS several *months*. Also, on low diameter graphs, the time taken by our improved MR_BFS is around one-third of that in [3]. Towards the end, we summarize our results (Table 11) by giving the state of the art implementations of external memory BFS on different graph classes.

Our implementations can be downloaded from http://www.mpi-sb.mpg.de/∼ajwani/em_bfs/.

2 Implementation Design

2.1 Graph generators and BFS decomposition verifier

We designed and implemented I/O efficient graph generators for generating large graphs in the DIMACS shortest path challenge format. Our generator tool is now a part of the benchmark for the challenge. The graph classes generated by our tools include:

- *Random graph*: On a n node graph, we randomly select m edges with replacement (i.e., m times selecting a source and a target node such that the source and the target are different nodes) and remove the duplicate edges to obtain random graphs.
- *MR_worst graph*: This graph consists of B levels, each having $\frac{n}{B}$ nodes, except the level 0 which contains only the source node. The edges are randomly distributed between consecutive levels, such that these B levels approximate the BFS levels. The initial layout of the nodes on the disk is random. This graph causes MR_BFS to incur its worst case of $\Omega(n)$ I/Os.
- *Grid graph ($x \times y$)*: It consists of a $x \times y$ grid, with edges joining the neighouring nodes in the grid.
- *MM_BFS worst graph*: This graph [7] causes MM_BFS_R to incur its worst case of $\Theta(n \cdot \sqrt{\frac{\log n}{B}} + \text{sort}(n))$ I/Os.
- *Line graphs*: A line graph consists of n nodes and $n-1$ edges such that there exists two nodes u and v, with the path from u to v consisting of all the $n-1$ edges. We took two different initial layouts - simple, in which all blocks consists of B consecutively lined nodes and the random in which the arrangement of nodes on disk is given by a random permutation.
- *Web graph*: As an instance of a real world graph, we consider an actual crawl of the world wide web [37], where an edge represents a hyperlink between two sites. This graph has around 130 million nodes and 1.4 billion edges. It has a core which consists of most of its nodes and behaves like random graph. In our graph generator tool, we include a translator for converting this graph into the DIMACS format.

For an I/O-efficient random permutation needed in the generation process of many graphs, we use [31]. The results were verified by an I/O efficient routine. More details for the graph generators, graph classes and BFS verifier routine can be found in [3].

2.2 Design issues in MR_BFS and MM_BFS_R

STXXL: We use the external memory library STXXL [16, 18] for our implementations. Although we use some special features of this library, we believe that modulo some constants, the results should carry over to other external memory libraries as well. The key component of STXXL used by us is the stream sorter, which runs in two phases - **Runs Creator (RC) Phase**, in which the input vector/stream is divided into chunks of M elements and each chunk is sorted within itself, thereafter written to the disk space and **Runs Merger (M) Phase**, in which the first blocks of all the sorted chunks are brought to internal memory and merged there to produce the output stream which does not necessarily have to be stored on the disk and can be pipelined to the next algorithmic routine.

Data Structures: The input graph is stored as two vectors containing nodes and edges with iterators from nodes to the locations in the edge vector marking the beginning of their adjacency array. Similarly, the output BFS level decomposition is also kept as two vectors containing the BFS level and the nodes in the level. The partitioned input graph is stored as three vectors containing the cluster indices, nodes and adjacency arrays, respectively. The iterators between them mark the beginning of clusters and the beginning of adjacency arrays.

Although we reduced the amount of information kept with node and edge elements in this data-structure, our implementation is still generic: it can handle graphs with arbitrary number of nodes and the graph template is basic and can be used for other graph algorithms as well.

Pipelining: The key idea behind pipelining is to connect a given sequence of algorithms with an interface so that the data can be passed-through from one algorithm to another without needing any external memory intermediate storage. Our BFS implementations make extensive use of this feature. For more details on the usage of pipelining as a tool to save I/Os, refer to [17]. More details about the data-structures and the usage of pipelining in external memory BFS algorithms can be found in [3].

2.3 Improvements in the previous implementations of MR_BFS and MM_BFS_R

The computation of each level of MR_BFS involves sorting and scanning of neighbours of the nodes in the previous level. Even if there are very few elements to be sorted, there is a certain overhead associated with initializing the external sorters. In particular, while the STXXL stream sorter (with the flag DSTXXL_SMALL_INPUT_PSORT_OPT) does not incur an I/O for sorting less than B elements, it still requires to allocate some memory and does some computation for initialization. This overhead accumulates over all levels and for large diameter graphs, it dominates the running time. This problem is also inherited by the BFS phase of MM_BFS. Since in the pipelined implementation of [3], we do not know in advance the exact number of elements to be sorted, we can't switch between the external and the internal sorter so easily. In order to get around this problem, we first buffer the first B elements and initialize the external sorter only when the buffer is full. Otherwise, we sort it internally.

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

2.4 Designing MM_BFS_D

There are three main components for the deterministic variant of MM_BFS:

- Sorting
- Connected component/ Minimum spanning tree
- List ranking

The MM_BFS_D implementation of Frederik Christiani in [13] uses the cache-oblivious lazy funnel-sort algorithm [10] (CO_sort). As Table 1 shows, the STXXL stream sort (STXXL_sort) proved to be much faster on external data. This is in line with the observations of Brodal et al. [9], where it is shown that an external memory sorting algorithm in the library TPIE [6] is better than their carefully implemented cache-oblivious sorting algorithm, when run on disk.

Regarding connected components and minimum spanning forest, Christiani's implementations [13] use the cache oblivious algorithm given in [1] (CO_MST). Empirically, we found that the external memory implementation of [19] (EM_MST) performs better than the one in [1].

Christiani uses the algorithm in [12] for list ranking the Euler tour. We adapted the algorithm in [33] to the STXXL framework. While Christiani's cache oblivious list ranking implementation takes around 14.3 *hours* for ranking a 2^{29} element random list using 3 GB RAM, our tuned external memory implementation takes less than 40 *minutes* in the same setting.

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

Table 1: Timing in minutes for sorting n elements using CO_sort and with using STXXL_sort

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

Table 2: Time taken (in hours) by the BFS phase of MM_BFS_D with long and random clustering

To summarize, our STXXL based implementation of MM_BFS_D uses our adaptation of [33] for list ranking the Euler tour around the minimum spanning tree computed by EM_MST. The Euler tour is then chopped into sets of $\sqrt{B}$ consecutive nodes which after duplicate removal gives the requisite graph partitioning. The BFS phase remains similar to MM_BFS_R.

Quality of the spanning tree

The quality of the spanning tree computed can have a significant impact on the clustering and the disk layout of the adjacency list after the deterministic preprocessing, and consequently on the BFS phase. For instance, in the case of grid graph, a spanning tree containing a list with elements in a snake-like row major order produces long and narrow clusters, while a "random" spanning tree is likely to result in clusters with low diameters. Such a "random" spanning tree can be attained by assigning random weights to the edges of the graph and then computing a minimum spanning tree or by randomly permuting the indices of the nodes. The nodes in long and narrow clusters tend to stay longer in the pool and therefore, their adjacency lists are scanned more often. This causes the pool to grow external and results in larger I/O volume. On the other hand, low diameter clusters are evicted from the pool sooner and are scanned less often reducing the I/O volume of the BFS phase. Consequently as Table 2 shows, the BFS phase of MM_BFS_D takes only 28 hours with clusters produced by "random" spanning tree, while it takes 51 hours with long and narrow clusters.

2.5 A Heuristic for maintaining the pool

As noted in Section 1.2, the asymptotic improvement and the performance gain in MM_BFS over MR_BFS is obtained by decomposing the graph into low diameter clusters and maintaining an efficiently accessible pool of adjacency lists which will be required in the next few levels. Whenever the first node of a cluster is visited during the BFS, the remaining nodes of this cluster will be reached soon after and hence, this cluster is loaded into the pool. For computing the neighbours of the nodes in the current level, we just need to scan the pool and not the entire graph. Efficient management of this pool is thus, crucial for the performance of MM_BFS. In this section, we propose heuristics for efficient management of the pool, while keeping the worst case I/O bounds of MM_BFS.

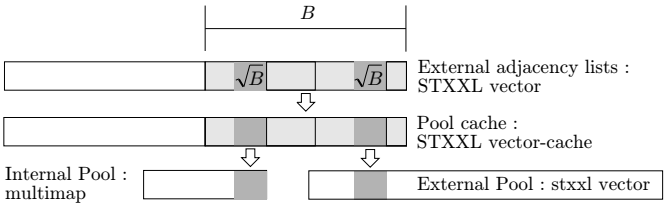

Figure 1: Schema depicting the implementation of our heuristic

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

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

We represent the adjacency lists of nodes in the graph as a STXXL vector. STXXL already provides a fully associative vector-cache with every vector. Before doing an I/O for loading a block of elements from the vector, it first checks if the block is already there in the vector-cache. If so, it avoids the I/O loading the elements from the cache instead. Increasing the vector-cache size of the adjacency list vector with a layout computed by the deterministic preprocessing and choosing the replacement policy to be LRU provides us with an implementation of the pool-cache. Figure 1 depicts the implementation of our heuristic.

3 Experiments

Configuration

We have implemented the algorithms in C++ using the g++ 4.02 compiler (optimization level -O3) on the *GNU/Linux* distribution with a 2.6 *kernel* and the external memory library STXXL version 0.77. Our experimental platform has two 2.0 GHz Opteron processors (we use only one), 3 GB of RAM, 1 MB cache and 250 GB Seagate Baracuda hard-disks [34]. These hard-disks have 8 MB

buffer cache. The average seek time for read and write is 8.0 and 9.0 msec, respectively, while the sustained data transfer rate for outer zone (maximum) is 65 MByte/s. This means that for graphs with 2^{28} nodes, n random read and write I/Os will take around 600 and 675 hours, respectively. In order to compare better with the results of [3], we restrict the available memory to 1 GB for our experiments and use only one processor and one disk.

Note that some of the results shown in this section have been interpolated using the symmetry in the graph structure.

Comparing MM_BFS_R

Graph class	n	m	MM_BFS_R of [3]		Improved MM_BFS_R	
			Phase 1	Phase 2	Phase 1	Phase 2
Random	2^{28}	2^{30}	5.1	4.5	5.2	3.8
MM_worst	$\sim 4.3 \cdot 10^7$	$\sim 4.3 \cdot 10^7$	6.7	26	5.2	18
MR_worst	2^{28}	2^{30}	5.1	45	4.3	40
Grid ($2^{14} \times 2^{14}$)	2^{28}	2^{29}	7.3	47	4.4	26
Simple Line	2^{28}	$2^{28} - 1$	85	191	55	2.9
Random Line	2^{28}	$2^{28} - 1$	81	203	64	25
Webgraph	$\sim 1.4 \cdot 10^8$	$\sim 1.2 \cdot 10^9$	6.2	3.2	5.8	2.8

Table 3: Timing in hours taken for BFS by the two MM_BFS_R implementations

Graph class	n	m	MM_BFS_R of [3]		Improved MM_BFS_R	
			I/O wait	Total	I/O wait	Total
MM_worst	$\sim 4.3 \cdot 10^7$	$\sim 4.3 \cdot 10^7$	13	26	16	18
Grid ($2^{14} \times 2^{14}$)	2^{28}	2^{29}	46	47	24	26
Simple Line	2^{28}	$2^{28} - 1$	0.5	191	0.05	2.9
Random Line	2^{28}	$2^{28} - 1$	21	203	21	25

Table 4: I/O wait time and the total time in hours for the BFS phase of the two MM_BFS_R implementations on moderate to large diameter graphs

Table 3 shows the improvement that we achieved in MM_BFS_R. As Table 4 shows, these improvements are achieved by reducing the computation time per level in the BFS phase. On I/O bound random graphs, the improvement is just around 15%, while on computation bound line graphs with random disk layout, we improve the running time of the BFS phase from around 200 hours to 25 hours. Our implementation of the randomized preprocessing in the case of the simple line graphs additionally benefits from the way clusters are laid out on the disk as this layout reflects the order in which the nodes are visited by the BFS. This reduces the total running time for the BFS phase of MM_BFS_R on simple line graphs from 191 hours to 2.9 hours. The effects of caching are also seen in the I/O bound BFS phase on the grid graphs, where the I/O wait time decreases from 46 hours to 24 hours.

Comparing MR_BFS

Graph class	n	m	MR_BFS of [3]		Improved MR_BFS	
			I/O wait	Total	I/O wait	Total
Random	2^{28}	2^{30}	2.4	3.4	1.2	1.4
Webgraph	$\sim 1.4 \cdot 10^8$	$\sim 1.2 \cdot 10^9$	3.7	4.0	2.5	2.6
MM_worst	$\sim 4.3 \cdot 10^7$	$\sim 4.3 \cdot 10^7$	25	25	13	13
Simple line	2^{28}	$2^{28} - 1$	0.6	10.2	0.06	0.4

Table 5: Timing in hours taken for BFS by the two MR_BFS implementations

Improvements in MR_BFS are shown in the Table 5. On random graphs where MR_BFS performs better than the other algorithms, we improve the runtime from 3.4 hours to 1.4 hours. Similarly for the web-crawl based graph, the running time reduces from 4.0 hours to 2.6 hours. The other graph class where MR_BFS outperforms MM_BFS_R is the MM_worst graph and here again, we improve the performance from around 25 hours to 13 hours.

Penalty for cache obliviousness

We compared the performance of our implementation of MM_BFS_D with Christiani's implementation [13] based on cache-oblivious subroutines. Table 6 show the results of the comparison on the two extreme graph classes for the preprocessing and the BFS phase respectively. We observed that on both graph classes, the preprocessing time required by our implementation is significantly less than the one by Christiani.

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

Graph class	Christiani's implementation	Our implementation
Random graph	107	5.2
Line graph with random layout on disk	47	3.3

Table 6: Timing in hours for computing the deterministic preprocessing of MM_BFS by the two implementations of MM_BFS_D

Comparing MM_BFS_D with other external memory BFS algorithm implementations

Table 7 shows the performances of our implementations of different external memory BFS algorithms. While MR_BFS performs better than the other two on random graphs saving a few *hours*, MM_BFS_D outperforms MR_BFS and MM_BFS_R on line graphs with random layout on disk saving a few *months* and a few *days*, respectively. On the later graph class, the speed up factor

Graph class	MR_BFS	MM_BFS_R	MM_BFS_D
Random graph	1.4	8.9	8.7
Line graph with random layout on disk	4756	89	3.6

Table 7: Timing in hours taken by our implementations of different external memory BFS algorithms.

Graph class	Randomized	Deterministic
Random graph	500	630
Line graph with random layout on disk	10500	480

Table 8: I/O volume (in GB) of the two preprocessing variants of MM_BFS.

has been improved from around 50 between MR_BFS and MM_BFS_R to around 1300 between MR_BFS and MM_BFS_D. Line graphs with random disk layouts are an example of a tough input for external memory BFS as they not only have a large diameter (more number of BFS levels), but also their layout on the disk makes the random accesses to adjacency lists very costly. MM_BFS_D also performs better on graphs with moderate diameter.

On large diameter sparse graphs such as line graphs, the randomized preprocessing scans the graph $\Omega(\sqrt{B})$ times, incurring an expected number of $O(\sqrt{n \cdot (n+m) \cdot \log(n)/B})$ I/Os. On the other hand, the I/O complexity of the deterministic preprocessing is $O((1 + \log\log D \cdot B \cdot n/m) \cdot \text{sort}(n+m))$, dominated by the spanning tree computation. Note that the Euler tour computation followed by list ranking only requires sort(m) I/Os. This asymptotic difference shows in the I/O volume of the two preprocessing variants (Table 8), thereby explaining the better performance of the deterministic preprocessing over the randomized one (Table 9). On low diameter random graphs, the diameter of the clusters is also small and consequently, the randomized variant scans the graph fewer times leading to less I/O volume.

As compared to MM_BFS_R, MM_BFS_D provides dual advantages: First, the preprocessing itself is faster and second, for most graph classes, the partitioning is also more robust, thus leading to better worst-case runtimes in the BFS phase. The later is because the clusters generated by the deterministic preprocessing are of diameter at most $\sqrt{B}$, while the ones by randomized preprocessing can have a larger diameter.

Graph class	Randomized	Deterministic
Random graph	5.2	5.2
Line graph with random layout on disk	64	3.2

Table 9: Time (in hours) required for the two preprocessing variants.

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

Table 10: Time taken (in hours) by the two phases of MM_BFS_D with our heuristic

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

Table 11: The best total running time (in hours) for BFS traversal on different graphs with the best external memory BFS implementations

Results with heuristic

Table 10 shows the results of MM_BFS_D with our heuristic on different graph classes. On moderate diameter grid graphs as well as large diameter random line graphs, MM_BFS_D with our heuristic provides the fastest implementation of BFS in the external memory.

Summary

The current state of the art implementations of external memory BFS on different graph classes are shown in Table 11.

Our improved MR_BFS implementation outperforms the other external memory BFS implementations on low diameter graphs or when the nodes of a graph are arranged on the disk in the order required for BFS traversal. For random graphs with 256 million nodes and a billion edges, our improved MR_BFS performs BFS in just 1.4 hours. Similarly, improved MR_BFS takes only 2.6 hours on webgraphs (whose runtime is dominated by the short diameter core) and 0.4 hours on line graph with contigous layout on disk. On moderate diameter square grid graphs, the total time for BFS is brought down from 54.3 hours for MM_BFS_R implementation in [3] to 21 hours for our implementation of MM_BFS_D with heuristics, an improvement of more than 60%. For large diameter graphs like random line graphs, MM_BFS_D along with our heuristic computes the BFS in just about 3.6 *hours*, which would have taken the MM_BFS_R implementation in [3] around 12

days and MR_BFS and IM_BFS a few *months*, an improvement by a factor of more than 75 and 1300, respectively.

4 Conclusion

We implemented the deterministic variant of MM_BFS and showed its comparative analysis with other external memory BFS algorithms. We propose a heuristic that improves upon the results of MM_BFS_D. Together with our earlier implementations of MR_BFS and MM_BFS_R, it provides viable BFS traversal on different classes of massive sparse graphs.

Acknowledgements

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

References

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

[2] 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.

[3] D. Ajwani, R. Dementiev, and U. Meyer. A computational study of external-memory BFS algorithms. *Seventeenth annual ACM-SIAM Symposium On Discrete Algorithms (SODA)*, pages 601–610, 2006.

[4] L. Arge, G. Brodal, and L. Toma. On external-memory MST, SSSP and multi-way planar graph separation. In *Proc. 8th Scand. Workshop on Algorithmic Theory (SWAT)*, volume 1851 of *LNCS*, pages 433–447. Springer, 2000.

[5] L. Arge, L. Toma, and J. S. Vitter. I/O-efficient algorithms for problems on grid-based terrains. *In Proc. Workshop on Algorithm Engeneering and Experiments (ALENEX)*, 2000.

[6] L. Arge et.al. `http://www.cs.duke.edu/TPIE/`.

[7] G. Brodal. Personal communication between Gerth Brodal and Ulrich Meyer.

[8] G. Brodal, R. Fagerberg, U. Meyer, and N. Zeh. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In *Proc. 9th Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 3111 of *LNCS*, pages 480–492. Springer, 2004.

[9] G. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. In *Proc. 6th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 4–17. SIAM, 2004.

[10] G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. *29th International Colloquium on Automata, Languages and Programming*, pages 426–438, 2002.

[11] A. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. Westbrook. On external memory graph traversal. In *Proc. 11th Ann. Symposium on Discrete Algorithms (SODA)*, pages 859–860. ACM-SIAM, 2000.

[12] Y. J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamasia, D. E. Vengroff, and J. S. Vitter. External memory graph algorithms. In *Proc. 6th Ann.Symposium on Discrete Algorithms (SODA)*, pages 139–149. ACM-SIAM, 1995.

[13] Frederik Juul Christiani. Cache-oblivious graph algorithms, 2005. Master's thesis, Department of Mathematics and Computer Science, University of Southern Denmark.

[14] T. H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[15] E. Demaine. Cache-oblivious algorithms and data structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets*, LNCS. Springer, 2002.

[16] R. Dementiev. Stxxl homepage. `http://i10www.ira.uka.de/dementiev/stxxl.shtml`.

[17] R. Dementiev, L. Kettner, and P. Sanders. Stxxl: Standard Template Library for XXL Data Sets. Technical Report 18, Fakultät für Informatik, University of Karlsruhe, 2005.

[18] R. Dementiev and P. Sanders. Asynchronous parallel disk sorting. In *Proc. 15th Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 138–148. ACM, 2003.

[19] R. Dementiev, P. Sanders, D. Schultes, and J. Sibeyn. Engineering an external memory minimum spanning tree algorithm. In *Proc. 3rd International Conference on Theoretical Computer Science (TCS)*, pages 195–208. Kluwer, 2004.

[20] S. Edelkamp, S.. Jabbar, and S. Schrödl. External A*. In *Proc. KI 2004*, volume 3238 of *LNAI*, pages 226–240. Springer, 2004.

[21] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297. IEEE Computer Society Press, 1999.

[22] 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.

[23] P.C. Guide. Disk Latency. `http://www.pcguide.com/ref/hdd/perf/perf/spec/posLatency-c.html`.

[24] A. Maheshwari and N. Zeh. External memory algorithms for outerplanar graphs. In *Proc. 10th Intern. Symp. on Algorithms and Computations (ISAAC)*, volume 1741 of *LNCS*, pages 307–316. Springer, 1999.

[25] A. Maheshwari and N. Zeh. I/O-efficient algorithms for graphs of bounded treewidth. In *Proc. 12th Ann. Symp. on Discrete Algorithms (SODA)*, pages 89–90. ACM-SIAM, 2001.

[26] A. Maheshwari and N. Zeh. I/O-optimal algorithms for planar graphs using separators. In *Proc. 13th Ann. Symp. on Discrete Algorithms (SODA)*, pages 372–381. ACM-SIAM, 2002.

[27] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proc. 10th Ann. European Symposium on Algorithms (ESA)*, volume 2461 of *LNCS*, pages 723–735. Springer, 2002.

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

[29] K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *In Proc. 10th Ann. Symposium on Discrete Algorithms (SODA)*, pages 687–694. ACM-SIAM, 1999.

[30] M. Najork and J. Wiener. Breadth-first search crawling yields high-quality pages. In *Proc. 10th International World Wide Web Conference*, pages 114–118, 2001.

[31] P. Sanders. Random permutations on distributed, external and hierarchical memory. *Information Processing Letters*, 67:305–309, 1998.

[32] V. Shkapenyuk and T. Suel. Design and implementation of a high-performance distributed web crawler. In *Proc. International Conference on Data Engineering (ICDE)*. IEEE, 2002.

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

[34] Seagate Technology. `http://www.seagate.com/cda/products/discsales/marketing/detail/0,1081,628,00.html`.

[35] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM computing Surveys, 33*, pages 209–271, 2001.

[36] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two level memories. *Algorithmica*, 12(2–3):110–147, 1994.

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