A Computational Study of External-Memory BFS Algorithms*

Deepak Ajwani † Roman Dementiev ‡ Ulrich Meyer †

Abstract

Breadth First Search (BFS) traversal is an archetype for many important graph problems. However, computing a BFS level decomposition for massive graphs was considered nonviable so far, because of the large number of I/Os it incurs. This paper presents the first experimental evaluation of recent external-memory BFS algorithms for general graphs. With our STXXL based implementations exploiting pipelining and disk-parallelism, we were able to compute the BFS level decomposition of a web-crawl based graph of around 130 million nodes and 1.4 billion edges in less than 4 hours using single disk and 2.3 hours using 4 disks. We demonstrate that some rather simple external-memory algorithms perform significantly better (*minutes* as compared to *hours*) than internal-memory BFS, even if more than half of the input resides internally.

1 Introduction

Solving real world optimization problems often boils down to traversing graphs in a structured way. Breadth First Search (BFS) is among the most fundamental such traversal strategies. 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.

BFS is well-understood in the *von Neumann* model which assumes uniform memory access costs. Actual machines, however, increasingly deviate from this model. While waiting for a memory access, modern microprocessors can execute a number of register operations. Even worse, in an external-memory (EM) setting where the input graph is too big to fully fit in internal memory certain accesses have to be served by hard disks, thus incurring an I/O loss factor of up to seven orders of magnitude.

Typical real-world applications of EM BFS (and some of its generalizations like shortest paths or A^*) include crawling and analyzing the WWW [29, 32], route planning using small navigation devices with flash memory cards [20], state space exploration [18], etc. In some of these cases the I/O-bottleneck is alleviated by heuristics, precomputations, accepting approximate solutions or profiting from special graph properties. In contrast, despite major efforts, I/O-efficient exact BFS for general sparse graphs was considered nonviable until recently: the MM_BFS approach in [26] is the first to yield an $o(n)$-I/O bound for general undirected sparse graphs. However, we are not aware of any empirical study to evaluate the practical merits of either MM_BFS or its alternatives (most prominently MR_BFS [28]).

In this paper, we attempt at narrowing the gap between theory and practice by engineering and extensively testing implementations of MR_BFS and MM_BFS on a low-cost machine. The code will become publicly available to be used with the STXXL library [15, 16]. Even though we use some special features of STXXL, we conjecture that (modulo some constants) our results transfer to other libraries that efficiently implement scanning and sorting like, e.g., TPIE [4, 6].

In a nutshell, our findings for EM BFS on sparse graphs are as follows: Most importantly, our study demonstrates that I/O-efficient external-memory BFS is feasible at all: for example, we computed the BFS level decomposition of an external web-crawl based graph of around 130 million nodes and 1.4 billion edges in less than 4 hours using a single disk and 2.3 hours using 4 disks. It turned out that both MR_BFS and MM_BFS perform significantly better (*minutes* as compared to *hours*) than internal-memory BFS, even if less than half of the input resides externally. Furthermore, MR_BFS exhibits some advantages over MM_BFS on low-diameter graphs (saving a few *hours*) whereas MM_BFS wins hands-down on most large-diameter graphs (a few *days* versus a few *months*). Altogether, in many situations, the moderate overhead of MM_BFS on 'easy' instances seems to be an acceptable investment given its significant gains on really 'tough' inputs. We expect our results to prove helpful in judging the potential gains and limitations of more complicated (future) approaches, too.

The rest of the paper is organized as follows: In Section 2, we give a short description of the external memory model and the BFS algorithms considered in

*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}@mpi-inf.mpg.de

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

this study. In Section 3, we deal with various implementation issues - STXXL, data-structures, pipelining. We also describe our framework for generating graphs, BFS traversal routines and BFS verifier. In Section 4 we discuss our results on various graph classes.

2 Models and Algorithms

In connection with graph algorithms, the commonly accepted external-memory model by Aggarwal and Vitter [1] defines some parameters: M $(< n + m)$ is the number of vertices/edges that fit into internal memory, and B is 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 [35] or hide the parameters M and B from the algorithms (cache-oblivious model [19]): due to their generality, cache-oblivious algorithms are efficient on each level of the memory hierarchy. A comprehensive list of results for the I/O-model and memory hierarchies have been obtained – for recent surveys see [14, 27, 34] and the references therein.

EM-BFS Algorithms covered in this study. In this paper we compare the performance of three BFS algorithms on large inputs: the standard internal-memory approach (refered as IM_BFS) [12], an algorithm by Munagala and Ranade (hereafter refered as MR_BFS) [28], and an approach by Mehlhorn and Meyer (MM_BFS) [26]. MR_BFS and MM_BFS are restricted to undirected graphs.

IM_BFS visits the vertices in a one-by-one fashion; appropriate candidate nodes for the next vertex to be visited are kept in a FIFO queue Q. After a vertex v is extracted from Q, the adjacency list of v is examined in order to append unvisited nodes to Q. IM_BFS in external memory has two drawbacks: (1) remembering visited nodes needs $\Theta(m)$ I/Os in the worst case; (2) unstructured indexed access to adjacency lists may result in $\Theta(n)$ I/Os. MR_BFS neglects (2) but addresses (1) by I/O-efficiently comparing the BFS level under creation with the two most recently completed BFS-levels. The resulting worst-case I/O-bound is $O(n + \text{sort}(n + m))$.

MM_BFS applies a preprocessing phase in which it groups the vertices of the input graph into disjoint clusters of small diameter and then runs an appropriately modified version of MR_BFS. It 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 $\sqrt{B}$: edges may be scanned more often in the pool but unstructured I/O in order to fetch adjacency lists is reduced. The preprocessing of MM_BFS can be done in several ways. We implemented a simple randomized variant which yields an overall expected I/O-bound of $O(\sqrt{n \cdot (n + m) \cdot \log(n)/B} + \text{sort}(n + m))$.

EM-BFS Algorithms not (yet) covered in this study – and why. An important goal of our study was to prove that I/O-efficient external-memory BFS is feasible at all. Therefore, we chose to implement those approaches with presumably small constant factors first. Also, due to the intrinsic time consumption for large-scale series of tests in external-memory, a certain self-restraint was unavoidable.

The next more complicated approaches are the counterparts of MR_BFS for directed graphs [10]. While applying extra data structures to remember visited nodes, these algorithms share the weakness of MR_BFS concerning unstructured access to adjacency lists, thus incurring $\Omega(n)$ I/Os in the worst case. On non worst-case instances we expect performance similar to that of MR_BFS. An equivalent to MM_BFS featuring $o(n)$ I/Os on general directed sparse graphs is still to be discovered (if it exists).

There is also a bunch of EM-BFS algorithms featuring $O(\text{sort}(n))$ I/Os on special graphs classes like trees [10], grid graphs [5], planar graphs [25], outer-planar graphs [23], and graphs of bounded tree width [24]. One might be tempted to run the planar BFS approach on general graphs - accepting to loose the worst-case guarantee of small separators, thus causing additional I/O. Leaving aside probably large constants the approach of [3, 25] as it stands takes $\Omega(n \cdot B)$ internal operations, which more or less translate in the time needed to perform $\Omega(n)$ I/Os. On the other hand, MR_BFS and MM_BFS take $O(m \cdot \log n)$ and $O(m \cdot \log n + min\{\log n, \sqrt{\frac{n \cdot B \cdot D \cdot \log n}{n+m}}\} \cdot scan(n + m))$ internal operations, respectively.

Another general technique to obtain I/O-efficient algorithms applies some kind of simulation (e.g., [11, 13, 21]) of suitable parallel approaches. As for general BFS this failed so far due to the lack of appropriate parallel BFS algorithms (as a rule of thumb they need to be fast and work-efficient at the same time).

Finally, there is a recent paper [8] transferring MM_BFS into a cache-oblivious algorithm of similar theoretical performance. It would be interesting to see how the performance of this approach executed on virtual memory compares to our implementation of MM_BFS. It might even become interesting for internal-memory graph traversal [22]. The algorithm is somewhat involved but with more and more cache-oblivious basic routines like high-performance sorting becoming available (e.g., [9]), a prototypical implementation of cache-oblivious BFS is on our short-list for future work.

3 Implementation Design

STXXL. 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.

The STXXL stream sorter (from version 0.75 onwards) does not need any I/O if $n < B$. Also, for this case, the internal work is proportional to $n \log n$, independent of B. Converting a vector into stream or initialization of runs creator or runs merger do not cause any I/O. Since for graphs with $O(n)$ BFS levels, we don't want to spend one I/O per level and all these functions are called in each level, these features are crucial for our implementation.

Data Structures. In the STXXL design framework, an adjacency array can be implemented using two vectors N and E. N contains the iterators to locations in E, marking the beginning of a new adjacency array. Each edge is stored twice in E - once in the adjacency array of each adjacent node. This representation allows scanning the edge vector in scan$(2m)$ I/Os. To ensure that we do not spend one I/O per level, but rather incur at most $\frac{2n}{B}$ I/Os in total, for writting the output, even the output itself — the BFS level decomposition — is stored in a similar format.

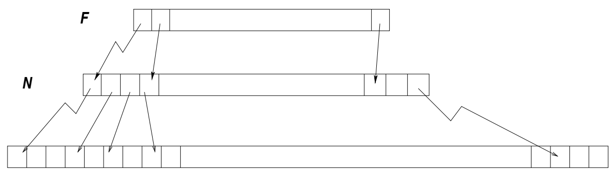


Figure 1: I/O-efficient data structure to represent a partitioned graph

For the case of MM_BFS, we also need another vector to store iterators to different clusters in order to access an arbitrary cluster in $O(1) + \frac{\text{cluster size}}{B}$ I/Os. Each cluster consists of an adjacency array containing nodes and edges belonging to that cluster after the preprocessing phase. A better way (from the I/O efficiency perspective) to handle the graph-partitioning is to store the partitioned input graph as three vectors $\mathcal{F}$, N, and E (as shown in Figure 1), containing the cluster iterators, nodes and adjacency arrays, respectively. N and E are kept sorted according to cluster indices.

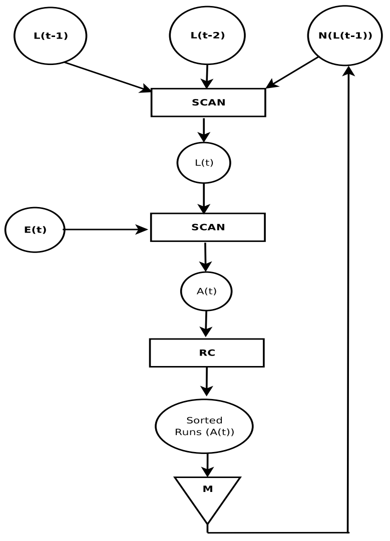
Although we reduced the amount of information kept with node and edge elements in this data-structure, our implementation is still generic: it can handle graphs with arbitrary number of nodes and the graph template is basic and can be used for other graph algorithms as well.

Pipelining. Pipelining is a well known principle used to design faster algorithms. 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. This coalesced algorithm is still manageable because none of these algorithms needs to know about the inner structure of the other algorithms. Though pipelining saves some constant factors in the I/O complexity of an algorithm, it usually increases the computational cost. Therefore, the correct extent of pipelining needs to be carefully determined. For more details on the usage of pipelining as a tool to save I/Os, refer to [17].

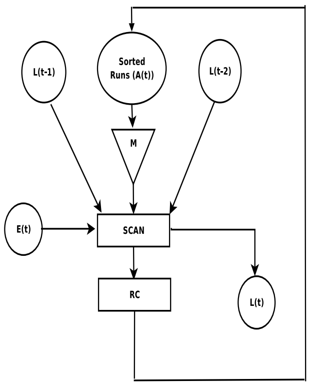
MR_BFS. Figure 2 displays the flowchart of the pipelined and the non-pipelined versions of the main loop of our implementation of MR_BFS. Let $L(t)$ denote the set of nodes in BFS level t and $N(S)$ denote the set of neighbors of nodes in S. The complexity of the pipelined MR_BFS mainly lies in it's scanner. The scanner reads the nodes in $L(t-1)$, $L(t-2)$ and the adjacency lists $E(t)$ of nodes in $L(t)$ from the disk and scans through the stream of $N(L(t-1))$ and in just one pass outputs the nodes in the current level $L(t)$ and the multi-set $A(t)$ of neighbor nodes of $L(t)$. The stream $A(t)$ is passed directly to runs creator and sorted runs are written on the disk. These are later merged and passed to the scanner as $N(L(t))$ for the next iteration.

In this case, pipelining reduces the worst case number of I/Os from $\sum_t (L(t-1) + L(t-2) + 2L(t) + E(t) + 2A(t) + \text{sorted runs of } A(t) + 2N(L(t-1))) = n + \text{scan}(4n + 14m)$ to $\sum_t (L(t-1) + L(t-2) + L(t) + E(t) + \text{sorted runs of } A(t)) = n + \text{scan}(3n + 6m)$.

MM_BFS. Figure 3 shows the flow-chart of the pipelined version of both phases of MM_BFS. The randomized graph paritioning phase begins with randomly selecting each node to be a master node with a probability μ. The main scanner of this phase takes the sorted sequence of the nodes on the fringe of expanding clusters, stores the cluster index (by including the fringe nodes into their corresponding clusters) with

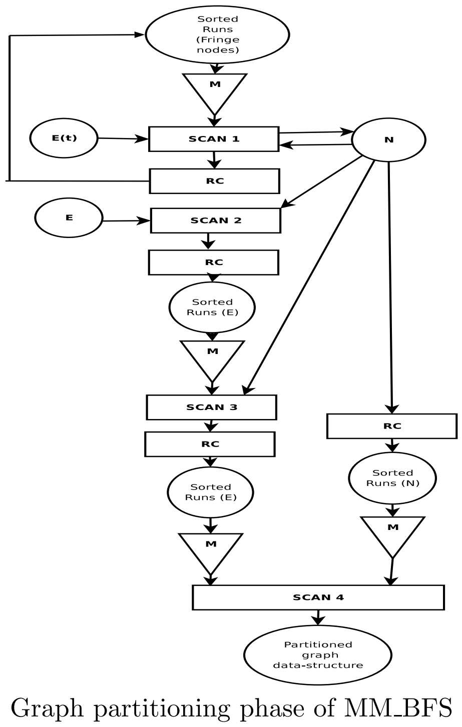


Flow-chart of MR_BFS

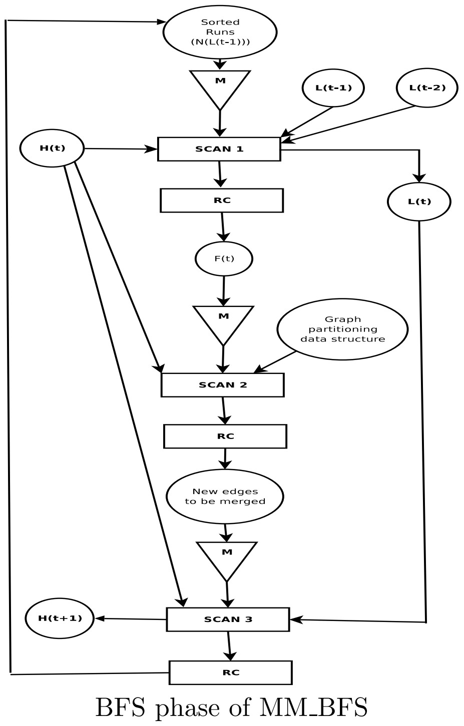


Flow-chart of pipelined version of MR_BFS

Figure 2: Flow-chart of MR_BFS implementation



Graph partitioning phase of MM_BFS



BFS phase of MM_BFS

Figure 3: Flow-chart of MM_BFS implementation

these nodes and reads their adjacency lists to compute the new sequence of fringe nodes to be sent to the two-phase sorter. After the partitioning of nodes into clusters is complete, we store the cluster index of the incident nodes with each edge, sort N and E w.r.t. the cluster index and then adjust the cluster and the node iterators appropriately. Since the diameters of any cluster is less than $\frac{\log n}{\mu}$ w.h.p., the total number of I/Os for this phase is bounded by $\text{scan}(16m + 6n + \mu n + \frac{2\cdot(m+n)\cdot\log n}{\mu})$ w.h.p.

In the pipelined BFS phase, the first scanner receives the sorted sequence $N(L(t-1))$ of neighbor nodes of $L(t-1)$ from the merger pipe, reads $L(t-1)$, $L(t-2)$ from the disk and the adjacency lists of nodes in $L(t-1)$ from the hot pool $\mathcal{H}$ and computes $F(t)$ - the multi-set of cluster indices of nodes in $L(t)$ - and in the process, also writes $L(t)$ to disk. The second scanner takes the sorted $F(t)$ and the hot pool $\mathcal{H}$ to compute the multi-set of cluster indices that need to be merged into $\mathcal{H}$. The next scanner reads the sorted sequence of $F(t)$, eliminates duplicate cluster-indices, computes $A(t)$ - the neighbor multi-set of $L(t)$ - and updates $\mathcal{H}$:

$$\mathcal{H}_{new} = \mathcal{H}_{old} + \mathcal{H}1(\text{Merged clusters}) - Adj(L(t))$$

where $Adj(S)$ represents the adjacency arrays of nodes in S. The total number of I/Os for this phase is bounded by $\mu n + \text{scan}(14m + 4n + \frac{8m\log m}{\mu})$ w.h.p.

Graph generators. For the purpose of this study, we designed and implemented a pipelined version of an I/O efficient framework for generating large graphs. The key steps of this framework are edge-sequence creation, sorting the edge-sequence, duplicate removal and conversion into requisite adjacency array representation. For an I/O-efficient random permutation needed in the generation process of many graphs, we use [31]. More details for this framework are available in [2].

BFS decomposition verifier. As another side tool, we designed an I/O efficient verifier routine to determine whether or not a BFS level decomposition is correct for a given graph.

For an undirected graph, any BFS level labeling is correct if it satisfies the following criterion:

- BFS level 0 contains the source node only.
- Every node has a unique BFS level.
- $\forall(u,v) \in E, |bfs\_level(u) - bfs\_level(v)| \leq 1$.
- $\forall u \in V$ in BFS level k ($k \neq 0$), $\exists$ $edge(u,v)$ such that v is in BFS level $k-1$.

All these conditions can be checked in $O(\text{sort}(n+m))$ I/Os.

4 Experiments

In this section, we consider the total running time and the I/O wait time - the total time spent by an implementation waiting for an I/O to complete, and not I/O time - the total time spent by an implementation on I/Os. This distinction is necessary as STXXL maximizes the overlap of I/O with computation.

Configuration. We have implemented the algorithms in C++ using the g++ 3.3.2 compiler (optimization level -O3) on the *GNU/Linux* distribution *Debian* with a 2.4 *kernel* and the external memory library STXXL version 0.77. Our experimental platform has two 2.0 GHz Intel Xeon processors (we use only one), one GByte of RAM, 512KB cache and 250 GByte Seagate Barracude hard-disks [33]. 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 2^{28} random read and write I/Os will take 596.5 and 671.1 hours, respectively.

Graph classes considered. *Random graph*: On a n node graph, we randomly select m edges with replacement (i.e., m times selecting a source and target node such that source $\neq$ target) and remove the duplicate edges to obtain random graphs.
B-level random graph: This graph consists of B levels, each (except the level 0 containing only the source node) having $\frac{n}{B}$ nodes. 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 such that the $\frac{n}{B}$ nodes in the same level are all in different blocks. This graph causes MR_BFS to incur it's worst case of $\Omega(n)$ I/Os.
B-level spider web graph: This graph class is a specialization of web graph defined in [37]. Again, it consists of B levels, each having $\frac{n}{B}$ nodes. All nodes in a level are connected in a cyclic fashion and a node has an edge to it's corresponding node in the level before and after. The initial layout of the nodes on the disk is random.
Grid graph: It consists of a $\lfloor n \rfloor \times \lceil n \rceil$ grid.
MM BFS worst graph: This graph [7] causes the (randomized pre-processing variant) MM_BFS to incur it's 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 three different initial layouts - simple, in which all blocks consists of B consecutively lined nodes; B-interleaved in which the consecutive nodes are all in different but consecutive blocks; random in which the arrangement of nodes on disk is given by a random permutation.

Fine-tuning Parameters. Most practitioners of

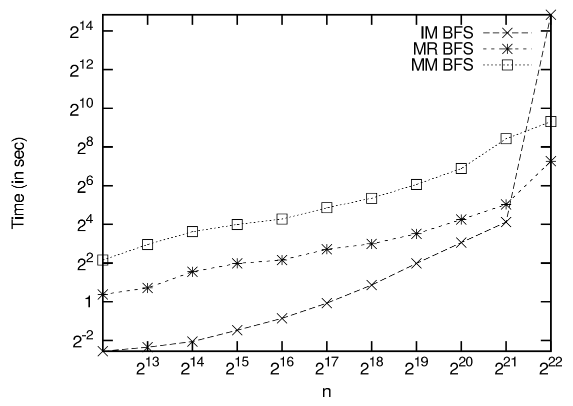


Figure 4: Variation of running time of IM_BFS, MR_BFS and MM_BFS (in logarithmic scale) with graph size (also in logarithmic scale)

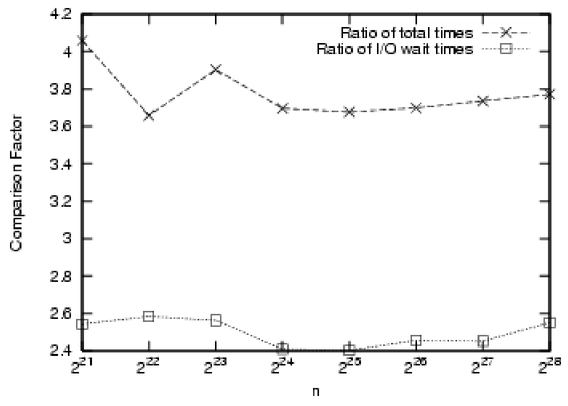


Figure 5: Variation of comparison factor (MM_BFS/MR_BFS) of running time and I/O wait time with random-graph size

external memory algorithms know that the block size B is a parameter that needs to be finely tuned for optimal performance. This is all the more relevant in the STXXL design framework, as the STXXL vector is organized as a collection of blocks (of size B) residing on the external storage media (parallel disks). Access to the external blocks is organized through the fully associative *cache* which consists of Pg_Nr in-memory pages where a page is a collection of Blk_Nr consecutive blocks. The internal memory consumption of a vector is $B \cdot Blk\_Nr \cdot Pg\_Nr$. Another important parameter to be fine-tuned is M_use - the internal memory reserved for a runs creator (or equivalanetly, runs merger). While tuning these parameters, a key constraint is that the internal memory allocated for all the vectors, runs creator and runs merger active simultaneouly, at any time, should be less than the internal memory available for the user. Typically, half of the internal memory is kept for OS requirements. For our implementations, we chose $B = 512$ KB (after experimenting between 1 KB and 8 MB), $Pg\_Nr = 4$, $Blk\_Nr =$ number of parallel disks in use, allocation strategy = randomized striping and LRU page replacement strategy.

Another important parameter to be optimized for MM_BFS is μ - the probability of choosing a node to be a master node. For worst case optimality, we choose $\mu = \sqrt{\frac{(m+n)\cdot\log(n)}{n\cdot B\cdot D}}$. On the other hand, the number of I/Os in the BFS phase for graphs with low (high) diameter can be quite less (high) and therefore, the optimal μ value for these graphs is much higher (lower). However, in order to exploit this fact, we need to assume some a priori knowledge of the graph structure. We consider both the cases - one in which we choose our μ value independent of the graph-structure (common μ) and one in which we assume a priori knowledge of it (graph-structure dependent μ).

IM_BFS looses fast. Figure 4 shows the total running time of IM_BFS, MR_BFS, and MM_BFS on random graphs of varying sizes (keeping $m = 4n$). An important point to note here is that even when half of the graph fits in internal memory, the performance of IM_BFS is much worse than that of external BFS algorithms. For this case (2^{22} nodes and 2^{24} edges), the I/O wait time of IM_BFS (8.09 *hours*) dominates the total running time (8.11 *hours*), thereby explaining the worse behavior of IM_BFS. On the other hand, MR_BFS and MM_BFS have much less I/O wait time (1.55 and 4.93 *minutes*) and consequently, the total runnning time (2.57 and 10.6 *minutes*) is also small. This clearly establishes the need for efficient implementations of external memory BFS algorithms.

Note that some of the results in Table 1- 9 have been interpolated using the symmetry in the graph structure.

Single disk - common μ. Table 1 and 2 show the I/O wait time and running time for different graphs in the single disk common μ case. Note that for these large graphs, even the efficient implementations of external memory algorithms are I/O dominant.

Let's first consider the case of random graphs. The total time for BFS traversal on random graphs is much less than that for most other graph classes. This is explained by the fact that MR_BFS incurs $O(\text{sort}(n+m))$ I/Os per level, and the number of BFS levels in random graphs (typically 10-15 for the graph sizes we studied) are very few. Infact, it's known [30] that a random graph $G(n, c/n)$ has an expected diameter $O(\log n)$. On the other hand, since μ for MM_BFS is $\sim \sqrt{\frac{\log n}{B}}$ for the asymptotic worst case I/O complexity, the edges remain in hot pool for quite long and MM_BFS

Graph class	n	m	MR_BFS		MM_BFS	
			I/O wait time	Total time	I/O wait time	Total time
Random	2^{28}	2^{30}	2.4	3.4	7.3	9.6
Grid	2^{28}	$\sim 2^{29}$	4733.0	4736.0	52.1	53.9
B-level random	2^{28}	2^{30}	3989.8	3994.8	42.2	49.7
B-level spider web	2^{28}	$\sim 2^{29}$	3364.2	3366.5	36.5	39.8
MM Worst	2^{25}	$\sim 2^{25}$	25.2	25.4	19.5	32.4
Simple line	2^{28}	2^{28}	0.6	10.2	84.8	275.9
Rand line	2^{28}	2^{28}	4156.2	4167.7	100.2	283.3
B-interleaved line	2^{28}	2^{28}	4210.3	4222.6	97.5	280.8

Table 1: Single Disk Common μ - I/O wait time and running time (in hours) of MR_BFS and MM_BFS

Graph class	n	m	MM_BFS Phase 1		MM_BFS Phase 2	
			I/O wait time	Total time	I/O wait time	Total time
Random	2^{28}	2^{30}	3.7	5.1	3.6	4.5
Grid	2^{28}	$\sim 2^{29}$	6.6	7.3	45.5	46.6
B-level random	2^{28}	2^{30}	3.6	5.1	38.6	44.6
B-level Spider Web	2^{28}	$\sim 2^{29}$	6.6	7.3	29.9	32.5
MM Worst	2^{25}	$\sim 2^{25}$	6.5	6.7	13.0	25.7
Simple line	2^{28}	2^{28}	84.3	85.1	0.5	190.8
Rand line	2^{28}	2^{28}	79.4	80.6	20.8	202.7
B-interleaved line	2^{28}	2^{28}	79.3	80.4	18.1	200.4

Table 2: Single Disk Common μ - I/O wait time and running time (in hours) of different phases of MM_BFS

Graph class	n	m	MR_BFS		MM_BFS	
			I/O wait time	Total time	I/O wait time	Total time
Random	2^{28}	2^{30}	0.9	1.3	1.9	4.4
B-level Random	2^{28}	2^{30}	2094.3	2105.1	18.3	26.0
B-level Spider Web	2^{28}	$\sim 2^{29}$	1492.0	1497.9	13.5	17.1
MM Worst	2^{25}	$\sim 2^{25}$	13.4	13.7	6.5	10.5
Rand line	2^{28}	2^{28}	4702.8	4730.5	56.4	239.9
B-interleaved line	2^{28}	2^{28}	1243.0	1258.7	56.4	239.9

Table 3: Multi disk common μ - I/O wait time and running time (in hours) of MR_BFS and MM_BFS

Graph class	n	m	MM_BFS Phase 1		MM_BFS Phase 2	
			I/O wait time	Total time	I/O wait time	Total time
Random	2^{28}	2^{30}	0.9	2.5	1.0	1.9
B-level random	2^{28}	2^{30}	0.9	2.5	17.4	23.5
B-level Spider Web	2^{28}	$\sim 2^{29}$	2.4	3.2	11.1	13.9
MM Worst	2^{25}	$\sim 2^{25}$	5.3	5.6	1.2	4.9
Rand line	2^{28}	2^{28}	49.0	50.5	7.5	189.4
B-interleaved line	2^{28}	2^{28}	48.4	49.0	6.1	28.1

Table 4: Multi disk common μ - I/O wait time and running time (in hours) of the two phases of MM_BFS

can still take $O(n \cdot \sqrt{\frac{\log n}{B}})$ I/Os. Consequently, as shown in Figure 5, MR_BFS not only outperforms MM_BFS in terms of total running time by a factor of ~ 3.8, but also in terms of I/O wait time by a factor of ~ 2.5.

As noted in Section 2, MR_BFS takes $O(n + \text{sort}(n + m))$ I/Os while MM_BFS takes $O(\text{sort}(n + m) + \sqrt{n \cdot \text{scan}(n + m) \cdot \log(n)})$ I/Os w.h.p.. So, if the two implementations are I/O bound and $\frac{m \cdot log(n)}{n} < B$, MM_BFS provides better running-time guarantees w.h.p.. In other words, we should be able to find some sparse graphs for which MM_BFS performs significantly better than MR_BFS. Our search for such a graph led

Graph class	n	m	MR_BFS		MM_BFS	
			I/O wait time	Total time	I/O wait time	Total time
Random	2^{28}	2^{30}	2.4	3.4	5.5	7.9
B-level Random	2^{28}	2^{30}	3989.8	3994.8	10.0	16.6
B-level Spider Web	2^{28}	$\sim 2^{29}$	3364.2	3366.5	25.1	29.3

Table 5: Single Disk, Graph structure dependent μ - I/O wait time and running time (in hours) of MR_BFS and MM_BFS

Graph class	n	m	MM_BFS Phase 1		MM_BFS Phase 2	
			I/O wait time	Total time	I/O wait time	Total time
Random	2^{28}	2^{30}	3.3	4.9	2.2	3.0
B-level Random	2^{28}	2^{30}	4.0	5.5	6.0	11.1
B-level Spider Web	2^{28}	$\sim 2^{29}$	12.9	13.7	12.2	15.6

Table 6: Single Disk, Graph structure dependent μ - I/O wait time and running time (in hours) of the two phases of MM_BFS

Graph class	n	m	MR_BFS		MM_BFS	
			I/O wait time	Total time	I/O wait time	Total time
Random	2^{28}	2^{30}	0.9	1.3	1.5	3.9
B-level Random	2^{28}	2^{30}	2094.3	2105.1	3.1	9.7
B-level Spider Web	2^{28}	$\sim 2^{29}$	1492.0	1497.9	9.7	13.9

Table 7: Multi disk graph structure dependent μ - I/O wait time and running time (in hours) of MR_BFS and MM_BFS

Graph class	n	m	MM_BFS Phase 1		MM_BFS Phase 2	
			I/O wait time	Total time	I/O wait time	Total time
Random	2^{28}	2^{30}	0.7	2.3	0.8	1.6
B-level random	2^{28}	2^{30}	1.1	2.6	2.0	7.1
B-level Spider Web	2^{28}	$\sim 2^{29}$	4.3	5.1	5.4	8.8

Table 8: Multi disk graph structure dependent μ - I/O wait time and running time (in hours) of the two phases of MM_BFS

	MR_BFS		MM_BFS - common μ		MM_BFS - Graph dep μ	
	I/O wait time	Total time	I/O wait time	Total time	I/O wait time	Total time
Single disk	3.7	4.0	7.4	9.4	6.3	8.4
Multiple disk	2.0	2.3	2.7	4.8	2.3	4.5

Table 9: I/O wait time and running time (in hours) of the two algorithms on web graph

us to B-level random graphs and B-level spider web graphs — high diameter sparse graphs, where MR_BFS takes $\Theta(n)$ I/Os. For these graphs, MR_BFS takes 90-95 times more I/O wait time than MM_BFS and as a result, while MM_BFS takes a total running time of 49.7 and 39.8 *hours* respectively, MR_BFS takes 166.5 and 140.3 *days*. A $\lceil\sqrt{n}\rceil \times \lfloor\sqrt{n}\rfloor$ grid is yet another instance of a high diameter $\lceil\sqrt{n}\rceil + \lfloor\sqrt{n}\rfloor$ sparse graph. Once again, MM_BFS outperforms MR_BFS by a factor of 90.8 in I/O wait time and 87.9 in total running time.

Apart from diameter, another important consideration affecting the relative performance of the two algorithms is the initial graph layout on the disk. The preprocessing phase of MM_BFS neutralizes the impact of an adverse layout. So, while we observe that the I/O wait time of MR_BFS (0.6 hours) is much less than 84.8 hours of MM_BFS (dominated by the 84.3 hours in the pre-processing phase) on a simple line graph, the I/O wait time of MR_BFS (167.6 and 177.7 days) is much more than that of MM_BFS (4.2 and 4.1 days) on a random and B-interleaved layouts. Thus, pre-processing makes MM_BFS provide better worst case guarantees (saving *months*) at the cost of loosing out on simple layouts (loosing *days*).

Except for the line graphs, the pre-processing phase dominates the run-time of MM_BFS for low diameter graphs while the BFS phase dominates it for large diameter graphs. For the line graphs, the hot pool is rarely bigger than B and therefore, always stays in internal memory.

Multi disk - common μ. We ran the same experiments with vectors striped over 4 disks. Although the usage of multiple disks allows us to handle larger volumes of data, herein we restrict ourselves to smaller sizes for better comparison with the single disk case. Table 3 and 4 show the running time of the two algorithms on different graphs. For most graphs, the usage of parallel I/O channels alleviates the I/O problem further and computation starts becoming the bottleneck, in particular for MM_BFS, which seems to gain more from the parallel I/O channels. However, with some new features of STXXL like a SMP multi-processor version of sorting routines, we hope to bring down the total running time fairly close to the I/O wait time. Besides, the computation speed increases at a much faster rate than the external memory bandwidth, thereby reducing the computation time relative to the I/O wait time.

While MR_BFS on random line hardly seems to have any benefit from the multiplicity of disks, it is almost four times better with four disks on B-interleaved line. This is because a random access to a block brings the neighoring blocks on other disks automatically to the internal memory and therefore, the access to the adjacency lists of next three nodes comes without any external I/O.

Single disk - graph structure dependent μ. Table 5 and 6 show the I/O wait time and running time for the two algorithms in the single disk case, where μ could be optimized based on the graph structure. Since for random graphs, there are very few BFS levels, it is better to choose a very high μ value ($\mu = 0.7$) so that a cluster stays in the hot pool for a short duration and therefore, an edge is considered in the hot pool for only one or two levels. With this μ, the I/O wait time of the BFS phase of MM_BFS is less than MR_BFS. So, in the case when we need to compute BFS from different source nodes knowing the graph-structure, the time for graph-partitioning can be amortized on multiple BFS runs, thereby making MM_BFS the preferable algorithm, even for random graphs. Since for B-level random and spider web graph, the hot pool almost always stays in the internal memory, the larger diameter of the cluster is no longer a problem. Consequently, we choose small μ value ($\mu = 0.00025 \simeq \frac{1}{B}$ and $\mu = 0.001$) in order to have fewer clusters and therefore, less time in loading them. In general, the μ value should be chosen so as to balance the I/O time of the two phases of MM_BFS.

Multi disk - graph structure dependent μ. Next, we experimented with running MM BFS on multiple-disk with graph-structure dependent μ to measure the extent to which we can alleviate the I/O bottleneck of BFS. Table 7 and 8 show the results for this setup.

Web graph - A real world graph. As an instance of a real world graph, we consider an actual crawl of the world wide web [36], where an edge represents a hyperlink between two sites. This graph has around 130 million nodes and 1.4 billion edges. For this graph, the total input and output volume of the BFS algorithm (with the data-structure described earlier) is around 23.5GB and 1.5GB, respectively. The bulk of the nodes are contained in the core of this web graph spread across 10-12 BFS levels (similar to random graphs). The remaining nodes are spread out over thousands of levels with 2-3 nodes per level (which behaves more like a line graph). However, the I/O wait time as well as the total running time for BFS traversal is dominated by the core of this graph and hence, the results are similar to the ones for random graphs. As Table 9 shows, both the algorithms can compute the BFS decomposition of this graph in a matter of few *hours.*

5 Conclusion

Empirical evidence suggests that MR_BFS performs better on small-diameter random graphs. However, the better asymptotic worst-case I/O complexity of MM_BFS helps it to outperform MR_BFS for large diameter sparse graphs, where MR_BFS incurs close to its worst case of $\Omega(n)$ I/Os.

References

[1] 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.

[2] D. Ajwani. Design, implementation and experimental study of external memory BFS algorithms, January, 2005. Masters Thesis, University of Saarland.

[3] 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.

[4] L. Arge, O. Procopiuc, and J. Vitter. Implementing I/O-efficient data structures using TPIE. In *Proc. 10th Ann. European Symposium on Algorithms (ESA)*, volume 2461 of *LNCS*, pages 88–100. Springer, 2002.

[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] 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.

[11] 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.

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

[13] F. Dehne, W. Dittrich, and D. Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. In *Proc. Symposium on Parallel Architectures and Algorithms (SPAA)*, pages 106–115, 1997.

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

[15] R. Dementiev. Stxxl homepage, documentation and tutorial. `http://stxxl.sourceforge.net`.

[16] R. Dementiev, L. Kettner, and P. Sanders. Stxxl: Standard template library for XXL data sets. In *13th Ann. European Symposium on Algorithms (ESA)*, volume 3669 of *LNCS*, pages 640–651. Springer, 2005.

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

[19] 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.

[20] 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.

[21] J. Gustedt. Towards realistic implementations of external memory algorithms using a coarse grained paradigm. In *International Conference on Computer Science and its Applications (ICCSA)*, volume 2668 of *LNCS*, pages 269–278. Springer, 2003.

[22] R. Ladner, J. Fix, and A. LaMarca. The cache performance of traversals and random accesses. In *Proc. 10th Ann. Symposium on Discrete Algorithms (SODA)*, pages 613–622. ACM-SIAM, 1999.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] J. Reif and P. Spirakis. Expected parallel time and sequential space complexity of graph and digraph problems. *Algorithmica*, 7:597–630, 1992.

[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] Seagate Technology. Barracuda 7200.8 ST3250823A disk specification. `http://www.seagate.com/cda/products/discsales/marketing/detail/0,1081,628,00.html`.

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

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

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

[37] E. W. Weisstein. "Web Graph" from MathWorld – A Wolfram Web Resource. `http://mathworld.wolfram.com/WebGraph.html`.