An Implementation of I/O-efficient Dynamic Breadth-First Search using level-aligned hierarchical Clustering. [⋆]

Andreas Beckmann[1], Ulrich Meyer[2], and David Veith[2]

[1] Jülich Supercomputing Centre (JSC), Forschungszentrum Jülich, D-52425 Jülich, Germany
[2] Institut für Informatik, Goethe-Universität Frankfurt, Robert-Mayer-Str. 11–15, D-60325 Frankfurt am Main, Germany

Abstract. In the past a number of I/O-efficient algorithms were designed to solve a problem on a static data set. However, many data sets like social networks or web graphs change their shape frequently. We provide experimental results of the first external-memory dynamic breadth-first search (BFS) implementation based on earlier theoretical work [13] that crucially relies on a randomized clustering. We refine this approach using a new I/O-efficient deterministic clustering, which groups vertices in a level-aligned hierarchy and facilitates easy access to clusters of changing sizes during the BFS updates. In most cases the new external-memory dynamic BFS implementation is significantly faster than recomputing the BFS levels after an edge insertion from scratch.

1 Introduction

Breadth first search (BFS) is a fundamental graph traversal strategy. It can be viewed as computing single source shortest paths on unweighted graphs. BFS decomposes the input graph $G = (V, E)$ of $n = |V|$ nodes and $m = |E|$ 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.

The objective of a dynamic graph algorithm is to efficiently process an online sequence of update and query operations; see [11, 15] for overviews of classic and recent results. In this paper we consider dynamic BFS for the incremental setting where additional edges are inserted one-by-one. After each edge insertion the updated BFS level decomposition has to be output.

At first sight, dynamic BFS on sparse graphs might not seem interesting since certain edge insertions could require $\Omega(n)$ updates on the resulting BFS levels, implying that the time needed to report the changes is in the same order of magnitude as recomputing the BFS levels from scratch using the standard

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

linear time BFS algorithm. The situation, however, is completely different in the external-memory setting, where the currently best static BFS implementations take much more time to *compute* the BFS levels as compared to *reporting* them, i.e. writing them to disk. Thus, providing a fast dynamic alternative is an important step towards a toolbox for external-memory graph computing. In this paper we report on the engineering of an external-memory dynamic BFS implementation (based on earlier theoretical work [13]). To this end a modified external-memory clustering method tuned to our needs has been developed, too.

2 I/O-Model and Related Work

Computation model. Theoretical results on out-of-core algorithms typically rely on the commonly accepted external-memory (EM) model by Aggarwal and Vitter [1]. It assumes a two level memory hierarchy with fast internal memory having a capacity to store M data items (e.g., vertices or edges of a graph) and a slow disk of infinite size. In an I/O operation, one block of data, which can store B consecutive items, 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, $\text{scan}(N) < \text{sort}(N) \ll N$.

Review of Static and Dynamic EM BFS Algorithms. There has been a significant number of publications on external-memory graph algorithms; see [3, 16] for recent overviews. In the following we shortly review the external memory MM_BFS algorithm by Mehlhorn and Meyer [12] and its dynamic extension [13]. In order to keep the description simple, we concentrate on edge insertions on already connected undirected sparse graphs with n=$|V|$ vertices and $m = |E| = \mathcal{O}(|V|)$ edges.

MM_BFS. The MM_BFS algorithm consists of two phases – a preprocessing phase and a BFS phase. In the preprocessing phase, the algorithm has to produce a clustering. This can be done with an Euler Tour technique based on an arbitrary spanning tree T of the graph G. In case T is not part of the input, it can be obtained using $\mathcal{O}((1 + \log\log(B \cdot n/m)) \cdot \text{sort}(n + m))$ I/Os [6]. Initially, each undirected edge of T is replaced by two oppositely directed edges. Then, in order to construct the Euler tour around this bi-directed tree, each node chooses a cyclic order [7] of its neighbors. The successor of an incoming edge is defined to be the outgoing edge to the next node in the cyclic order. The tour is then broken at a special node (say the root of the tree) and the elements of the resulting list are then stored in consecutive order using an external memory list-ranking algorithm; Chiang et al. [8] showed how to do this in sorting complexity. Thereafter, the Euler tour is chopped into clusters of μ nodes and duplicates are removed such that each node only remains in the *first* cluster it originally occurs; again this requires a couple of sorting steps. By construction, the distance in G between any two vertices belonging to the same cluster is bounded by $\mu - 1$ and there are $\mathcal{O}(n/\mu)$ clusters.

For the BFS phase, the key idea is to load whole preprocessed clusters into some efficient data structure (hot pool) at the expense of few I/Os, since the clusters are stored contiguously on disk and contain vertices in neighboring BFS levels. This way, the neighboring nodes $N(l)$ of some BFS level l can be computed by scanning only the hot pool. The next BFS level is obtained by removing those nodes visited in levels $l-1$ and l from $N(l)$; see [14]. However, as the algorithm proceeds, newly discovered neighbor nodes may belong to so far unvisited clusters. Unstructured I/Os are required to import those clusters into the hot pool, from where they are gradually evicted again once they have been used to create the respective BFS levels. Maintaining the hot pool itself requires $\mathcal{O}(\text{scan}(n+m)\cdot\mu)$ I/Os, whereas importing the clusters into it accounts for $\mathcal{O}(n/\mu + \text{sort}(n+m))$ I/Os. Choosing $\mu = \sqrt{B}$ yields an I/O-complexity of $\mathcal{O}(n/\sqrt{B} + \text{sort}(n))$ for the BFS-phase on sparse graphs.

Dynamic BFS. In the following we review the high-level ideas to computing BFS on general undirected sparse graphs in an incremental setting. Let us consider the insertion of the ith edge (u, v) and refer to the graph (and the shortest path distances from the source in the graph) before and after the insertion of this edge as G_{i-1} (d_{i-1}) and G_i (d_i), respectively.

We run the BFS phase of MM_BFS, with the difference that the adjacency list for v is added to the hot pool H when creating BFS level $\max\{0, d_{i-1}(v) - \alpha\}$ of G_i, for a certain advance $\alpha > 1$. By keeping the adjacency lists sorted according to node distances in G_{i-1} this can be done I/O-efficiently for all nodes v featuring $d_{i-1}(v) - d_i(v) \leq \alpha$. For nodes with $d_{i-1}(v) - d_i(v) > \alpha$, we import whole clusters containing their adjacency lists into H using unstructured I/Os. Each such cluster must comprise the adjacency lists of $\Omega(\alpha)$ nodes whose mutual distances in G_{i-1} are bounded by $\mu = \Theta(\alpha)$, each vertex belongs to exactly one cluster. If the BFS phase for the currently used value of α would require more than $\alpha \cdot n/B$ random cluster accesses, we increase α by a factor of two, compute a new clustering for G_{i-1} with larger chunk size μ and start a new attempt by repeating the whole approach with the increased parameters.

Meyer [13] proved an amortized high-probability bound of $\mathcal{O}(n/B^{2/3} + \text{sort}(n) \cdot \log B)$ I/Os per update under a sequence of $\Theta(n)$ edge insertions. The analysis relies on the fact that there can be only be very few updates in which the BFS levels change significantly for a large number of nodes. If it can be guaranteed that each cluster loaded into the pool actually carries $\Omega(\alpha)$ vertices, most of the updates will require few cluster fetches in early attempts with small advance.

Unfortunately, the standard Euler tour based clustering method described above might produce very unbalanced clusters: in fact $\Omega(n/\mu)$ clusters may contain only a single vertex each. A randomized clustering approach [13] repairs this deficiency as follows:

Each vertex v in the spanning tree T_s is assigned an independent binary random number $r(v)$ with $\mathbf{P}[r(v) = 0] = \mathbf{P}[r(v) = 1] = 1/2$. When removing duplicates from the Euler tour, instead of storing v in the cluster related to the chunk with the *first* occurrence of a vertex v, now we only stick to its first occurrence iff $r(v) = 0$ and otherwise ($r(v) = 1$) store v in the cluster that corresponds to the *last* chunk of the Euler tour v appears in. For chunk size $\mu > 1$ and each but the last chunk, the expected number of kept vertices is at least $\mu/8$.

3 Challenges and New Results

The dynamic BFS approach in [13] applies several clusterings of *different* values for μ (say $\mu = 2^1, 2^2, 2^3, \ldots, \sqrt{B}$) for the *same* input graph. In fact, for incremental dynamic BFS on already connected graphs, these clusterings could be re-used for all subsequent edge insertions. Unfortunately, while the theoretical EM model assume external space to be of unlimited size, this is not true in reality. In fact, due to disk space limitations it may often be impossible to keep even a few different clusterings at the same time. On the other hand, even though it will not harm the theoretical worst-case bounds, re-computing the same clusterings over and over again could actually become the dominating part of the I/O numbers we see in practice: for example in improved static BFS implementations of [4, 12], the preprocessing for graph clustering often takes more time than the actual BFS phase, although the latter comes with a significantly higher asymptotic I/O-bound than the preprocessing in the worst case. In addition, the old clustering method described above crucially relies on randomization. Thus, the improved deterministic clustering we propose in Section 4 features both practical and theoretical advantages.

As already mentioned in Section 2 the theoretical I/O bounds for the dynamic EM BFS algorithm in [13] are amortized over sequences of $\Theta(n)$ edge insertions. Hence, single updates could theoretically become as costly as with the static EM-BFS approach, and the hidden constants might be even worse. On the other hand, edge insertions with little effect on the resulting BFS levels should hopefully be manageable with significantly less I/O. In our practical experiments we consider such extreme cases on several graph classes. While our dynamic BFS implementation was never slower than a factor of 1.25 compared to static BFS, we have also experienced cases where dynamic BFS outperformes the static re-computation by more than a factor of 70.

4 Level-aligned Hierarchical Clustering

The high level idea for our hierarchical clustering is rather easy: we renumber each vertex with a new bit representation $\langle b_r, \ldots, b_{q+1}, b_q, \ldots, b_1 \rangle$ that is interpreted as a combination of prefix $\langle b_r, \ldots, b_{q+1} \rangle$ and suffix $\langle b_q, \ldots, b_1 \rangle$. Different prefixes denote different clusters, and for a concrete prefix (cluster) its suffixes denote vertices within this cluster. Depending on the choice of q we get the whole spectrum between few larger clusters (q big) or many small clusters (q small). In particular we would like the following to hold:

For any $1 \leq \mu = 2^q \leq \sqrt{B}$, (1) there are $\lceil n/\mu \rceil$ clusters, (2) each cluster comprises μ vertices (one cluster may have less vertices), and (3) for any two vertices u and v belonging to the same cluster, their distance in G is $\mathcal{O}(\mu)$. In order to make this work the new vertex numbers will have to be carefully chosen. Additionally, a look-up table is built that allows to find the sequence of disk blocks for adjacency lists of the vertices associated with a concrete cluster using $\mathcal{O}(1)$ I/Os.

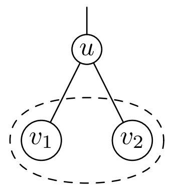

Fig. 1: Sibling-Merge

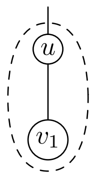

Fig. 2: Parent-Merge

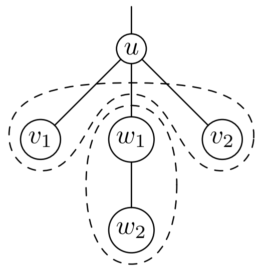

Fig. 3: Complex merge example

In order to group close-by vertices into clusters (such that an appropriate renumbering can take place) we start with an arbitrary spanning tree T_s^0 rooted at source vertex s. Then we work in $p = \left\lceil \log \sqrt{B} \right\rceil$ phases, each of which transforms the current tree T_s^j into a new tree T_s^{j+1} having $\lceil |T_s^j|/2 \rceil$ vertices. (The external-memory BFS algorithms considered here only use clusters up to a size of $\sqrt{B}$ vertices so this construction is stopped after p phases. The hierarchical clustering approach imposes no limitations and can be applied for up to $\lceil \log n \rceil$ phases for other applications.) The tree shrinking is done using time-forward processing [5, 8] from the leaves toward the root (for example by using negated BFS numbers for the vertices T_s^j). Consider the remaining leaves $v_1, \ldots, v_k$ with highest BFS numbers and common parent vertex u in T_s^j. If k is even then v_1 and v_2 will form a cluster (and hence a vertex in T_s^{j+1}), v_3 and v_4 will be combined, v_5 and v_6, etc. (*sibling-merge*, see Figure 1). In case k is odd, v_1 will be combined with u (*parent-merge*, Figure 2) and (if $k \geq 3$) v_2 with v_3, v_4 with v_5, etc. Merged vertices are removed from T_s^j and therefore any vertex is a leaf at the time it is reached by TFP, e.g. node w_1 shown in Figure 3 was already consumed by vertex w_2, so it is no longer available at the time v_1, v_2 get processed. Thus, each vertex of T_s^{j+1} is created out of exactly two vertices from T_s^j, except for the root which may only consist of the root from T_s^j. Note that the original graph vertices kept in a cluster are not necessarily direct neighbors but they do have short paths connecting them in the original graph. The following lemma makes this more formal:

Lemma 1. *The vertices of T_s^j form clusters in the original graph having size* $\text{size}^{(j)} = 2^j$ *(excluding the root vertex which may be smaller), maximum depth* $\overline{\text{depth}}^{(j)} = 2^j - 1$*, and maximum diameter* $\overline{\text{diam}}^{(j)} = 2^{j+1} - 2$*.*

Proof. By induction (obvious for $\text{size}^{(j)}$). The clusters defined by T_s^0 consist of exactly one vertex each and satisfy $\overline{\text{diam}}^{(0)} = 0$ and $\overline{\text{depth}}^{(0)} = 0$. For $j > 0$, three ways of merging a vertex v_1 have to be considered (with a sibling $(\{v_1, v_2\})$, the parent $(\{v_1, u\})$ or not merged at all $(\{v_1\})$), resulting in

$$\begin{aligned}
\overline{\text{depth}}^{(j)} &= \max \left\{ \begin{array}{l} \overline{\text{depth}}^{(j)}(\{v_1, v_2\}), \\ \overline{\text{depth}}^{(j)}(\{v_1, u\}), \\ \overline{\text{depth}}^{(j)}(\{v_1\}) \end{array} \right\} \\
&= \max \left\{ \begin{array}{l} \max\left\{\overline{\text{depth}}^{(j-1)}(v_1), \overline{\text{depth}}^{(j-1)}(v_2)\right\}, \\ \overline{\text{depth}}^{(j-1)}(v_1) + 1 + \overline{\text{depth}}^{(j-1)}(u), \\ \overline{\text{depth}}^{(j-1)}(v_1) \end{array} \right\} \\
&= 2 \cdot \overline{\text{depth}}^{(j-1)} + 1 = 2 \cdot (2^{j-1} - 1) + 1 = 2^j - 1
\end{aligned}$$

$$\begin{aligned}
\overline{\text{diam}}^{(j)} &= \max \left\{ \begin{array}{l} \overline{\text{diam}}^{(j)}(\{v_1, v_2\}), \\ \overline{\text{diam}}^{(j)}(\{v_1, u\}), \\ \overline{\text{diam}}^{(j)}(\{v_1\}) \end{array} \right\} \\
&= \max \left\{ \begin{array}{l} \overline{\text{depth}}^{(j-1)}(v_1) + 1 + \overline{\text{diam}}^{(j-1)}(u) + 1 + \overline{\text{depth}}^{(j-1)}(v_2), \\ \overline{\text{depth}}^{(j-1)}(v_1) + 1 + \overline{\text{diam}}^{(j-1)}(u), \\ \overline{\text{diam}}^{(j-1)}(v_1) \end{array} \right\} \\
&= 2 \cdot \overline{\text{depth}}^{(j-1)} + \overline{\text{diam}}^{(j-1)} + 2 = 2 \cdot (2^{j-1} - 1) + \overline{\text{diam}}^{(j-1)} + 2 \\
&= 2^j + \overline{\text{diam}}^{(j-1)} = 2^j + 2^j - 2 = 2^{j+1} - 2
\end{aligned}$$

Note that while $\overline{\text{diam}}^{(j)}$ and $\overline{\text{depth}}^{(j)}$ denote the *maximum* diameter and depth possible for a cluster with 2^j vertices the actual values may be much smaller. □

The hierarchical approach produces a clustering with (1) $\Theta(n/\mu)$ clusters each having (2) size $\Theta(\mu)$ (excluding the root cluster) and (3) diameter $\mathcal{O}(\mu)$ for each $1 \leq \mu = 2^q \leq \sqrt{B}$.

Details on the construction of T_s^{j+1}. Two types of messages (*connect*(id) and *merged*(id)) are sent during the time-forward processing. When vertices are combined, the vertex visited first sends the ID of the new vertex in T_s^{j+1} to the other one in a *merged*() message. The *connect*() messages are used to generate edges of T_s^{j+1} using the new IDs. The *merged*() message (if any) of a vertex is sorted before the *connect*() messages of that vertex, so checking whether the current minimal entry in the priority queue has received such a message can be done in $\mathcal{O}(1)$.

Renumbering the vertices. The p phases of contracting the spanning tree each contribute one bit of the new vertex number $\langle b_r, \ldots, b_{p+1}, b_p, \ldots, b_1 \rangle$. The construction of T_s^{j+1} defines the bit b_{j+1} for the vertices from T_s^j to be 0 for the left and 1 for the right child in case of a sibling-merge, to be 0 for the parent and 1 for the child in case of a parent-merge, and to be 1 for the root vertex (unless it was already merged with another vertex). After p contraction phases T_s^p with $c = \lceil n/2^p \rceil$ vertices/clusters remain. The remaining $(r-p)$ bits are assigned by computing BFS numbers (starting with $(2^{\lceil \log_2 c \rceil} - c)$ at the root) for the vertices of T_s^p. These BFS numbers are inserted (in their binary representation) as the bits $\langle b_r, \ldots, b_{p+1} \rangle$ of the new labels for the vertices of T_s^p.

To efficiently combine the label bits from different phases and propagate them to the vertices of G again time-forward processing can be applied. The trees $T_s^p, \ldots, T_s^0$ will be revisited in that order and vertices of T_s^j can be processed e.g. in BFS order and each vertex v of T_s^j will combine the label bits received from T_s^{j+1} with the bit assigned during the construction phase and then send messages with its partial label to the vertices in T_s^{j-1} it comprises. The resulting new vertex numbering of G will cover the integers from $[n'-n, n')$ where $n' = 2^{\lceil \log_2 n \rceil}$. There will be a single gap $[0, n'-n)$ (unless n is a power of two) that can be easily excluded from storage by applying appropriate offsets when allocating and accessing arrays. Thereafter the new labeling has to be propagated to adjacency lists of G and the adjacency lists have to be reordered ($\mathcal{O}(\text{sort}(n+m))$ I/Os).

Assuming the adjacency lists are stored as an adjacency array sorted by vertex numbers (two arrays, the first with vertex information, e.g. offsets into the edge information; the second with edge information, e.g. destination vertices), there is no need for an extra index structure to retrieve any cluster in $\mathcal{O}(1 + \frac{x}{B})$ I/Os where x is the number of edges in that particular cluster. This is possible because all vertices of a cluster are numbered contiguously (for all values of $1 \le \mu = 2^q \le \sqrt{B}$) and the numbers of the first and last vertex in a cluster can be computed directly from the cluster number (and cluster size μ).

Lemma 2. *For an undirected connected graph G with n vertices, m edges, and given spanning tree T_s the Hierarchical Bottom-Up Clustering for all cluster sizes $1 \le \mu = 2^q \le n$ can be computed using $\mathcal{O}(\text{sort}(n))$ I/Os for constructing the new vertex labeling and $\mathcal{O}(\text{sort}(n+m))$ I/Os for the rearrangement of the adjacency lists of G.*

5 Implementation details of dynamic BFS

In the theoretical design of dynamic BFS the single parameter α was used to control the following quantities: the size of the clusters, the timer threshold (to avoid that elements are kept in the hot pool during the whole computation after a previous cluster fetch of the same element) and the number of levels that are prefetched into the hot pool. In our implementation we used two parameters α_1 and α_2 instead. The number of levels that are fetched into the hot pool is denoted by α_1 whereas the size of the clusters is controlled by α_2. The timer values are given by a simple approximation of the cluster diameter.

We split the original hot pool into two hot pools – one for the fetched levels (denoted as H) and one for the loaded clusters (denoted as HC). The elements in the two hot pools have different properties (for example the cluster id and the timer are needed in HC but not in H) and we were able to measure random I/Os from cluster fetches and sequential I/Os from feeding H with new levels and removing consumed or outdated entries separately.

For the insertion of an edge (v_1, v_2) into the graph we made the following observation. Let l_1 be the level of the vertex v_1 and l_2 the level of vertex v_2 and w.l.o.g be $l_1 \leq l_2$. The first level f_1 with possible improvements for the BFS levels is given by $f_1 = l_1 + \lfloor \frac{l_2 - l_1}{2} \rfloor + 1$. Hence, we do not need to recompute the level of any vertex in a level $l < f_1$. The distance $l_2 - l_1$ might be arbitrarily huge. The α_1 levels that could be fetched into H will never be required if $l_1 + \alpha_1 < l_2$. Therefore we start prefetching levels into H from level f_1 instead of l_1. HC will load clusters in the local neighborhood of v_2 to assign new BFS levels to adjacent vertices.

In Section 4 we argued for simplicity that the hierarchical clustering can be implemented using the time-forward processing technique and a priority queue. Since we are operating on a tree we can actually omit the priority queue in order to achieve better constant factors: we build triples $(vertex, level, parent)$ for each vertex in our tree and sort them by level and furthermore by parent. Now we scan the triples and merge two adjacent elements if they have the same parent. If there is no such adjacent element it is merged with its parent which is considered in the next level. Hence, we store a message that its parent is already clustered and then the parent is omitted later. For small levels these messages fit in internal memory, for larger levels an external sorted vector is used.

6 Experiments

Configuration. Our external-memory dynamic BFS implementation relies on the STXXL library [10]. For our static EM-BFS results we used the STXXL code from Ajwani et al. [4]. We performed our experiments on a machine with an Intel dual core E6750 processor @ 2.66 GHz, 4GB main memory (3.5 GB free), 3 hard disks with 500 GB each as external memory for STXXL, and a separate disk for the operating system, graph data, log files etc. The operating system was Debian GNU/Linux amd64 'wheezy' (testing) with kernel 3.2. We compiled with GCC 4.7.2 in C++11 mode using optimization level 3.

Graph classes. For our experiments we used four different graph classes: one real-world graph with logarithmic diameter, two synthetic graph classes with diameters $\Theta(\sqrt{n})$ and $\Theta(n)$ and a tree graph class that was designed to elicit poor performance for the static BFS approach with standard Euler-tour clustering.

The real-world graph sk-2005 has around 50 million vertices, about 1.8 billion edges and is based on a web-crawl. It was also used by Crescenzi et al. [9] and has a known diameter of 40. The synthetic x-level graphs are similar to the B-level random graphs in [2]. They consist of x levels, each having $\frac{n}{x}$ vertices (except

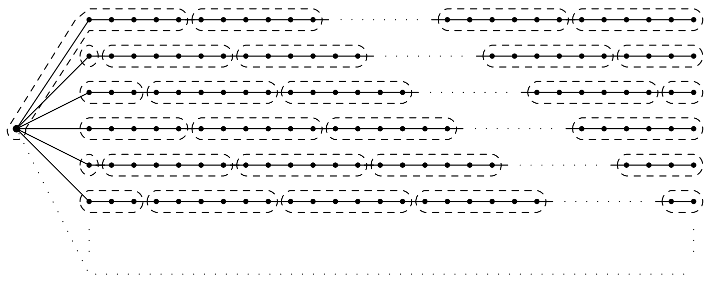

Fig. 4: Our cl_n2_29 graph has the same shape as the graph in this picture but with 1048576 lists of length 511 each. In this picture the result of an Euler tour based clustering with $\mu = 6$ is shown as it is used in MM_BFS.

the level 0 containing only one vertex). The edges are randomly distributed between consecutive levels. The $\sqrt{n}$-level graph graph features $n = 2^{28}$ nodes and $m = 1.1 \cdot 10^9$ edges, for the $\Theta(n)$-level graph we have $n = 2^{28}$ and $m = 0.9 \cdot 10^9$. The fourth graph class represents trees whose special shape are tuned to yield an Euler tour clustering that forces the static MM_BFS algorithm into $\Omega(n/\sqrt{B})$ unstructured I/Os: whenever a new BFS level is reached, many new clusters are encountered for the first time. A schematic depiction is presented in Figure 4. In our concrete case the resulting cl_n2_29 graph features about 2^{29} nodes and 2^{20} lists. The parameters were chosen in a way that prefetching heuristics will not help MM_BFS caching adjacency lists in main memory.

6.1 Results.

In our experiment we inserted new edges (v_1, v_2) into the graph, where we set $v_1 = s$ (the source of the BFS tree) and select the other vertex v_2 from BFS levels $0.1 \cdot d$, $0.2 \cdot d$, ..., d where d denotes the height of the BFS tree. The experiments were executed independently. For each inserted edge the initial BFS tree / graph was the same. The source of the BFS tree was chosen to make the experiments more difficult: two vertices far away from the source might have a small distance to each other and then usually only a small fraction of the whole graph data has to be reassigned to new BFS levels in our graph data. We measured the time for dynamic BFS plus the time to write the result and the number of vertices that have been updated. Experiments during the implementation showed that for a small cluster size α_2, e. g. $\alpha_2 = 64$, its value is algorithmically never increased. This leads to a high number of random I/Os. Thus we set $\alpha_2 = 1024$ which causes a small amount of random cluster fetches. For smaller α_2 our results were slightly better for each graph class on our test machine but not for the $\sqrt{n}$-level graph. The number of elements in the hot pool H is given by the value of α_1. For large α_1 a huge hot pool H has to be scanned for each BFS level computation. We set the initial $\alpha_1 = 4$ to avoid too many sequential I/Os. Table 1 contains the time for computing static EM-BFS for each graph divided into the preprocessing and the BFS computation. The cl_n2_29 graph stands out with a slow BFS computation while the preprocessing is almost as fast as the preprocessing for the other graph classes. Table 2 contains the time for the hierarchical clustering and the time that is needed to reorganize the adjacency lists (add cluster information to edges, sort them, ...). The clustering is slower than the preprocessing of the static BFS because logarithmically many phases e. g. containing Euler Tour and list-ranking computation have to be done instead of one. Nevertheless, the computation time is still independent from the graph class and the computation time only depends on the input size. The gain of the hierarchical clustering is obtained in the dynamic BFS computation of the cl_n2_29 graph (details at the end of this section).

	sk-2005	$\sqrt{n}$-level graph	$\Theta(n)$-level graph	cl_n2_29 graph
Time Preprocessing	0.91	1.35	1.19	1.29
Time BFS-level computation	2.41	3.26	1.36	$>$ 17

Table 1: Running time (in hours) of static EM-BFS with source 0.

	sk-2005	$\sqrt{n}$-level graph	$\Theta(n)$-level graph	cl_n2_29 graph
Compute hierarchical level-aligned clustering	0.39	1.64	1.35	3.01
Reorganization of adjacency lists	1.38	0.94	0.84	0.54

Table 2: Running time (in hours) of level-aligned hierarchical clustering.

Figure 5 contains the results of our dynamic BFS computing the updated BFS levels. Because one vertex of the newly inserted edge is always the source, the results mirror the local hot spots in the graph in which the update of a few BFS levels is expensive. For example, the web graph sk-2005 has many vertices in the levels close to the source 0. Vertices with a larger distance to the source seem to have a list-like path to the source. Hence, an update of the BFS-levels was very fast from vertices with large distance to the source. In our experiments the worst scenario is the $\sqrt{n}$-level graph. It is the only case in which our current implementation loses against EM-BFS for large distance between the two vertices of the new edge.

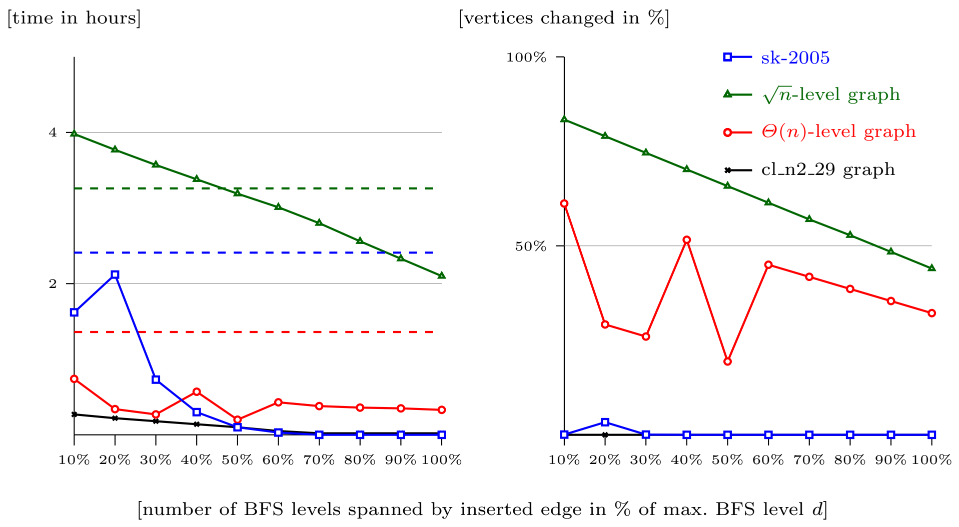

Fig. 5: Results of dynamic BFS. The time of each static BFS is plotted in as a dashed line in the left plot for sk-2005 (2.41 h), $\sqrt{n}$-level graph (3.26 h) and $\Theta(n)$-level graph (1.36 h). The static BFS time of cl_n2_29 graph was not drawn because it is too huge with 17 hours to suit into the plot.

Results of experiments with cl_n2_29 graph: as expected the hot pool of static BFS had to go external and reads/writes Terabytes of data (input data set size: 8 GB). Therefore, static MM_BFS needs more than 17 hours. Each update during the dynamic BFS computation needed at most 0.23 hours.

Our results using hard disks were viable due to comparatively large α_2. In experiments on a similar machine using solid state drives we were able to improve our results. We beat the static BFS for each graph class in each test scenario by using a smaller $\alpha_2 = 256$. For our $\sqrt{n}$-level graph we were able to beat static BFS by a factor of 1.14 in our worst case. This is explained by the fact that for smaller α_2 the work on CPU is much smaller but the I/O-time increases. It seems that with SSDs the I/O time increases slower than the CPU-time decreases. We plan to present more details in a full version.

7 Conclusion

We have given initial results of the first external-memory dynamic BFS implementation using a new deterministic level-aligned hierarchical clustering. Even though we applied rather hard edge insertion scenarios our implementation was usually faster, and for some graph classes much faster, than the static BFS implementation. We investigated the interaction of the different parameters that influence the performance of our dynamic BFS in more detail.

Acknowledgments

We want to thank Asmaa Edres for her work on the implementation of the time-forward processing hierarchical clustering using a priority queue in her master's thesis.

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. *Traversing large graphs in realistic setting*. PhD thesis, Saarland University, 2008.
3. D. Ajwani and U. Meyer. Design and engineering of external memory traversal algorithms for general graphs. In *Algorithmics of Large and Complex Networks*, volume 5515 of *LNCS*, pages 1–33. Springer, 2009.
4. D. Ajwani, U. Meyer, and V. Osipov. Improved external memory BFS implementation. In *Proc. 9th ALENEX*, pages 3–12, 2007.
5. L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.
6. L. Arge, G. Brodal, and L. Toma. On external-memory MST, SSSP and multi-way planar graph separation. *J. Algorithms*, 53(2):186–206, 2004.
7. M. Atallah and U. Vishkin. Finding euler tours in parallel. *Journal of Computer and System Sciences, 29(30)*, pages 330–337, 1984.
8. Y. J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamasia, D. E. Vengroff, and J. S. Vitter. External memory graph algorithms. In *Proceedings of the 6th annual Symposium on Discrete Algorithms (SODA)*, pages 139–149. ACM-SIAM, 1995.
9. P. Crescenzi, R. Grossi, C. Imbrenda, L. Lanzi, and A. Marino. Finding the diameter in real-world graphs – experimentally turning a lower bound into an upper bound. In *Proc. 18th ESA*, volume 6346 of *LNCS*, pages 302–313. Springer, 2010.
10. R. Dementiev and P. Sanders. Asynchronous parallel disk sorting. In *Proc. 15th SPAA*, pages 138–148. ACM, 2003.
11. D. Eppstein, Z. Galil, and G. Italiano. Dynamic graph algorithms. In Mikhail J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 8. CRC Press, 1999.
12. K. Mehlhorn and U. Meyer. External-memory Breadth-First Search with sublinear I/O. In *Proc. 10th ESA*, volume 2461 of *LNCS*, pages 723–735. Springer, 2002.
13. U. Meyer. On dynamic Breadth-First Search in external-memory. In *25th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 551–560, 2008.
14. K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proceedings of the 10th Annual Symposium on Discrete Algorithms (SODA)*, pages 687–694. ACM-SIAM, 1999.
15. L. Roditty. *Dynamic and static algorithms for path problems in graphs*. PhD thesis, Tel Aviv University, 2006.
16. J. S. Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2006.