An Experimental Study of External Memory Algorithms for Connected Components

Gerth Stølting Brodal
Aarhus University, Denmark

Rolf Fagerberg
University of Southern Denmark, Odense, Denmark

David Hammer
Goethe Universität Frankfurt, Germany
University of Southern Denmark, Odense, Denmark

Ulrich Meyer
Goethe Universität Frankfurt, Germany

Manuel Penschuck
Goethe Universität Frankfurt, Germany

Hung Tran
Goethe Universität Frankfurt, Germany

Abstract

We empirically investigate algorithms for solving Connected Components in the external memory model. In particular, we study whether the randomized $O(\mathrm{Sort}(E))$ algorithm by Karger, Klein, and Tarjan can be implemented to compete with practically promising and simpler algorithms having only slightly worse theoretical cost, namely Borůvka's algorithm and the algorithm by Sibeyn and collaborators. For all algorithms, we develop and test a number of tuning options. Our experiments are executed on a large set of different graph classes including random graphs, grids, geometric graphs, and hyperbolic graphs. Among our findings are: The Sibeyn algorithm is a very strong contender due to its simplicity and due to an added degree of freedom in its internal workings when used in the Connected Components setting. With the right tunings, the Karger-Klein-Tarjan algorithm can be implemented to be competitive in many cases. Higher graph density seems to benefit Karger-Klein-Tarjan relative to Sibeyn. Borůvka's algorithm is not competitive with the two others.

2012 ACM Subject Classification Mathematics of computing → Paths and connectivity problems; Theory of computation → Graph algorithms analysis

Keywords and phrases Connected Components, Experimental Evaluation, External Memory, Graph Algorithms, Randomization

Digital Object Identifier 10.4230/LIPIcs.SEA.2021.23

Supplementary Material *Software (Source Code)*: `https://ae.cs.uni-frankfurt.de/r/p/em-cc` archived at `swh:1:dir:a11f45c7dd725022c9898f14f1b2a9afe1136b9a`

Funding Based upon work partially performed while attending AlgoPARC Workshop on Parallel Algorithms and Data Structures at the University of Hawaii at Manoa, organized by Nodari Sitchinava and in part supported by the National Science Foundation under Grant No. CCF-1930579. Gerth Stølting Brodal and Rolf Fagerberg have been supported by the Independent Research Fund Denmark under grants 9131-00113B and DFF-7014-00041, respectively. David Hammer, Ulrich Meyer, Manuel Penschuck, and Hung Tran have been supported by the Deutsche Forschungsgemeinschaft (DFG) under grants ME 2088/4-2 (SPP 1736 Algorithms for Big Data) and ME 2088/5-1.

Acknowledgements Extensive calculations on the Goethe-HLR high-performance computer of the Goethe University Frankfurt were conducted for this research. The authors would like to acknowledge the CSC team for their support. We would also like to thank Peter Sanders for valuable feedback on an earlier draft of this paper.

1 Introduction

The Connected Components (CC) problem is a fundamental algorithmic task on undirected graphs and has a large number of applications including web graph analysis, communication network design, image analysis, and clustering in computational biology. CC may be viewed as a smaller sibling of the Minimum Spanning Forest (MSF) problem defined on weighted, undirected graphs – any algorithm solving MSF and able to return the trees of the forest one by one can be used to solve CC by first assigning arbitrary edge weights.

In internal memory, CC is simple to solve in linear time by DFS or BFS. A long-standing open problem is whether MSF can also be solved deterministically in linear time. The large body of work devoted to the question (see e.g. the references in [22]) indicates that in internal memory, MSF is harder to tackle than CC, at least in terms of the algorithmic sophistication needed (and potentially also in terms of the asymptotic complexity of the problem).

In external memory (see Section 2 for the definition of the model and its parameters), the I/O-complexity of CC and MSF is bounded from below by $\Omega(E/V \cdot \mathrm{Sort}(V))$ [18] and a number of algorithms come within at most a logarithmic factor of $O(\mathrm{Sort}(E))$. No deterministic algorithm is known to match the lower bound, but a randomized algorithm with $O(\mathrm{Sort}(E))$[1] expected cost exists [15, 8]. Unlike in internal memory, the known external memory CC algorithms are essentially the same as the known algorithms for MSF, either exactly or as close variants. The largest discrepancy between the two settings is for the randomized $O(\mathrm{Sort}(E))$ algorithm, where a fairly involved subroutine in its MSF variant becomes straight-forward for CC.

It seems that the randomized $O(\mathrm{Sort}(E))$ external memory algorithm was never empirically investigated. One aim of this paper is to carry out such an investigation in the CC setting where the discrepancy mentioned above gives the algorithm the largest opportunity of being competitive in practice. Due to the large size of internal memory in most current computer systems, it is not clear whether a small asymptotic advantage of at most a logarithmic factor will materialize in practice for graphs of very large, but still plausible, sizes. In more detail, we want to investigate implementations and tuning options for the randomized $O(\mathrm{Sort}(E))$ CC algorithm, as well as for the practically most promising of the remaining (asymptotically slightly worse, but often simpler) external CC algorithms, and then compare the best implementations of each algorithm on a broad range of graph classes. More generally, the aim of this paper is to investigate the best algorithmic choices for solving the CC problem in external memory.

Previous work. In the semi-external case, where $V \leq M$, scanning the edges and maintaining the components via a Union-Find data structure in internal memory will solve CC in $O(\mathrm{Scan}(E))$ I/Os. The classic Borůvka MSF algorithm was externalized by Chiang et al. [8] by showing how to implement a Borůvka step in $O(\mathrm{Sort}(E))$ I/Os, leading to $O(\log(V/M) \cdot \mathrm{Sort}(E))$ I/Os for the entire algorithm. A simpler method for implementing a Borůvka step in $O(\mathrm{Sort}(E))$ I/Os was later given by Arge et al. [4]. Munagala and Ranade [18] gave a CC algorithm using $O(\log\log(VB/E) \cdot \mathrm{Sort}(E))$ I/Os and also proved the above-mentioned lower bound. The algorithm was generalized to MSF by Arge et al. [4], keeping the I/O bound. The algorithm of [4] was further developed by Bhushan and Gopalan [7], slightly improving the I/O bound.

[1] Using sparsification, the algorithm can be implemented to use $O(E/V \cdot \mathrm{Sort}(V))$ I/Os [8], matching the lower bound exactly. In this paper, we will consider its $O(\mathrm{Sort}(E))$ version as the two bounds are very close and in practice their difference is unlikely to outweigh the added algorithmic complication.

Karger, Klein, and Tarjan [15] gave an internal MSF algorithm with expected $O(E)$ running time using a linear time MSF verification algorithm as its central subroutine. The algorithm can be externalized to use expected $O(\mathrm{Sort}(E))$ I/Os [8] by using external Borůvka steps and the external MSF verification algorithm by Chiang et al. [8]. For CC, it is an easy observation (already made by [2]) that the MSF verification can be substituted by a contraction step, which simplifies the implementation considerably. To the best of our knowledge, neither the CC nor the MSF variant of this external memory algorithm has been studied empirically.

A very simple randomized MSF algorithm using expected $O(\log(V/M) \cdot \mathrm{Sort}(E))$ I/Os was developed by Sibeyn and Meyer. It was first reported by Schultes [24], and further described and empirically tested by Dementiev et al. [26]. A CC variant was theoretically and empirically studied by Sibeyn [27] (and to a lesser extent by Schultes [25]). Due to its simplicity, the algorithm is likely to have very competitive constants in its I/O bound, which is argued theoretically in [27] and substantiated by the experiments in [24, 25, 26, 27]; however, none of these experiments include comparisons to other external memory algorithms.

Our contribution. We implement the CC version of the $O(\mathrm{Sort}(E))$ randomized and external algorithm by Karger, Klein, and Tarjan [15] and develop and investigate a number of tuning options. We then compare it to tuned versions of what we consider the practically most promising other algorithms for the CC, namely external Borůvka and the algorithm by Sibeyn et al. [24, 25, 26, 27]. Our experiments are executed on numerous graph classes, including $\mathcal{G}(n, p)$ graphs, grids, geometric graphs, and hyperbolic graphs (see Section 6).

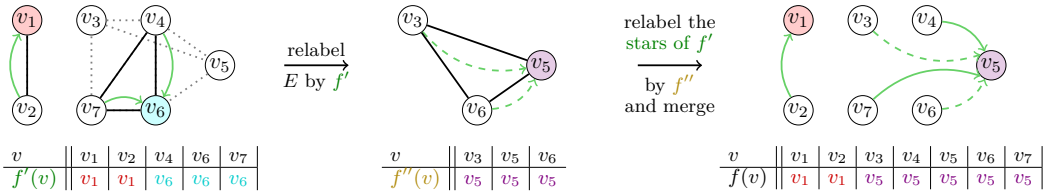
Among our findings are: Sibeyn's algorithm is a very strong contender due to its simplicity and due to an added degree of freedom in its internal workings when used in the CC setting. With the right tunings, the Karger-Klein-Tarjan algorithm can be implemented to be competitive. Higher graph density seems to benefit Karger-Klein-Tarjan relative to Sibeyn, as does larger graph sizes. The latter observation is in line with its better (expected) asymptotic I/O bound. Borůvka's algorithm is not competitive compared to its contenders.

2 Definitions

The Connected Components (CC) problem on an undirected graph $G = (V, E)$ is to partition V such that two nodes are in the same subset iff they are connected by a path in G. We overload the symbols V and E: depending on the context, V may represent either the *set* or the *number* of nodes, and E may similarly represent either the set or the number of edges.

We analyze the cost of algorithms in the I/O-model of Aggarwal and Vitter [1] where M denotes the size of the internal memory, B denotes the block size, and $\mathrm{Scan}(N) = \Theta(N/B)$ and $\mathrm{Sort}(N) = \Theta(N/B \log_{M/B}(N/M))$ denote the costs of scanning and sorting N elements.

As input, we assume the standard external memory representation of a graph as a list of its edges. This means that isolated nodes cannot be represented and should be handled separately by the user, which is straight-forward as they constitute their own connected components. We denote by $V(E)$ the set of nodes contained in an edge set E. Hence, an input is formally a graph $G = (V(E), E)$, but for simplicity we denote it just by E. We require the input E to be given in lexicographical order, as all our algorithms need this. We thereby avoid an initial sorting step in the algorithms, which would only make their relative differences in running times less clear. Unless otherwise stated, we also assume that each unordered edge $\{u, v\}$ is stored only once in its *normalized* form $(\min(u, v), \max(u, v))$.



v	v_1	v_2	v_4	v_6	v_7
$f'(v)$	v_1	v_1	v_6	v_6	v_6

(a) Obtain f' by solving CC on E'

v	v_3	v_5	v_6
$f''(v)$	v_5	v_5	v_5

(b) Obtain f'' by solving CC on E relabeled by f'

v	v_1	v_2	v_3	v_4	v_5	v_6	v_7
$f(v)$	v_1	v_1	v_5	v_5	v_5	v_5	v_5

(c) Obtain f by merging the star of f'' and the star of f' relabeled with f''

Figure 1 Relabeling and contraction. The input E and the subset $E' \subseteq E$ are illustrated in **(a)** where E' corresponds to solid black edges and $E - E'$ to dotted lines. Solving CC on E' yields f' which represents the two CCs $\{v_1, v_2\}$ and $\{v_4, v_6, v_7\}$ by v_1 and v_6, respectively. This corresponds to the two stars indicated by the directed green edges. Since v_3 and v_5 are not covered by E', they are also not included in f'. The result of the contraction E/E' is shown in **(b)** with solid lines and is obtained by relabeling E by f'. Solving CC on E/E' yields f'' indicated by the dashed directed edges. In **(c)**, we merge the stars of f' relabeled with f'' (solid) together with the stars of f'' (dashed) and obtain the final result f. Observe that the star of f may contain edges (e.g. (v_7, v_5)) that were not part of the original input E.

As output, we require a mapping $f\colon V(E) \to V(E)$ where $f(v) = f(u)$ iff u and v are in the same connected component. In other words, for each connected component one node is chosen as its representative. Concretely, the mapping shall be returned as the list of pairs $\{(v, f(v)) \mid v \in V(E)\}$, except that all identities (v, v) are omitted. Note that we can interpret this output as the edge list of a directed graph composed of disjoint *stars*, where a star is a set of nodes pointing to a common center node. Each star represents a connected component in E.

A *relabeling* of a graph E by a mapping $f\colon V(E) \to V(E)$ means applying f to all edge endpoints and then removing parallel edges and self-loops in the resulting edge list. If f is given by a graph of oriented stars as described above, a relabeling can be implemented in $O(\mathrm{Sort}(E))$ I/Os by $O(1)$ sorting and scanning steps on E. A *contraction* E/E' of a graph E by a subset $E' \subseteq E$ of its edges means solving CC on E' and then relabeling E by the returned mapping f'. The concepts of relabeling and contraction are illustrated in Figure 1 **(a)** and **(b)**, respectively.

Note that if we next obtain a mapping f'' by solving CC on the contracted graph E/E', we can solve CC on the original graph E as follows: use the mapping f'' to relabel the graph of stars representing f' (only the target of each star edge is affected by the relabeling) and then return the union of those relabeled edges and the edges of the graph of stars representing f''. The process is illustrated in Figure 1 **(c)**. It is easy to verify that it will produce a graph of stars representing the solution f to CC on the original graph E. All recursive algorithms in the current paper use this process as their framework.

3 Algorithms

In this section, we describe the basic versions of the implemented algorithms.

Union-Find. In the semi-external case, where $V(E) \leq M$, scanning the edges once while maintaining a Union-Find data structure on $V(E)$ in internal memory solves CC in $O(\mathrm{Scan}(E))$ I/Os and $O(E\alpha(E, V(E)))$ time [28], where α is the inverse Ackermann function. We use this as a base case.

Borůvka. A Borůvka step in the MSF setting means letting each node choose an incident edge of minimum weight and then contracting the graph by the set E' of chosen edges. In E', each node is in a connected component of size at least two, so the number of nodes is at least halved in the step. As a Borůvka step requires $O(\mathrm{Sort}(E))$ I/Os (see below), this leads to a recursive algorithm which will use $O(\log_2(V/M)\mathrm{Sort}(E))$ I/Os before the semi-external base case is reached. This constitutes Borůvka's algorithm.

The first part of a Borůvka step finds E' with $O(\mathrm{Sort}(E))$ I/Os as follows: double E during a scan to make it contain both directions of each undirected edge. Then for all nodes choose an incident edge of minimum weight via a single sort and scan of this version of E.

To implement the remainder of a Borůvka step, one can exploit that E' is a graph where each connected component has exactly one cycle, as seen by repeatedly following paths of chosen edges until all nodes have been visited. Assuming that all edge weights are unique (otherwise, use node IDs as tie-breakers), the weights along any such path are strictly decreasing, except when traversing the lightest undirected edge $\{u, v\}$ of the component in two directions $(u, v), (v, u)$, implying that the cycle is a two-cycle. Both directions have the same normalized representation, hence can be identified and de-duplicated by sorting E', after which the connected component corresponds to a tree rooted in v. This can be done for all such pairs in the same sorting step, making the edges E' form a forest where each tree coincides with a connected component. We select the roots as the components' IDs.

In order to return the star graph of the mapping f', we have to inform each node of its tree's root. Early external methods [8, 2] used algorithms for Euler tours of trees based on list ranking. We use a simpler method described in [4]. It requires $O(\mathrm{Sort}(V(E)))$ I/Os and is based on the fact that edge weights are strictly increasing on root-to-leaf paths in the trees, i.e., if we address messages to nodes by the weight of their incoming edge, parents will be processed before their children. This allows edge weights to be used as a "time line" in a general technique known as time-forward processing [17]. The propagation is done for all trees simultaneously by maintaining a set of signals in an external priority queue. The data structure is initialized by inserting signals for all children of all roots. Using sorting steps, we also create a list L of tree edges not incident to a root. In L, all child edges of a node v are grouped together, and the order between groups is determined by the weight of the parent edge of v. We then repeatedly remove the signal with smallest key from the priority queue and forward the information contained to the next block of children from L.

In the CC setting, the above algorithm for a Borůvka step can be implemented by (formally) assigning to all edges their unique normalized identity as their weight. Note that in the first part of the step, this is equivalent to each node simply choosing the edge to the neighbor with the lowest ID.

Karger-Klein-Tarjan. The CC version [2] of the $O(\mathrm{Sort}(E))$ randomized, external algorithm based on Karger, Klein, and Tarjan [15] has the following recursive structure:

1. Perform three Borůvka steps on the input graph. Let the result be E.
2. Let E' contain each edge of E independently with probability $1/2$.
3. Compute the connected components of E' recursively.
4. Form the contraction $E'' = E/E'$.
5. Compute the connected components of E'' recursively.
6. Relabel the result of step 3 by the result of step 5 and merge with the result of step 5, as detailed in Section 2.
7. Perform the relabelings and merges corresponding to the contraction in each of the initial Borůvka steps (as detailed in Section 2) and return the result.

In step 4, only the edges in $E - E'$ need to be processed as contraction by E' eliminates all edges in E'. The crux of the Karger-Klein-Tarjan algorithm is that the number of edges in E'' is $O(V(E))$ in expectation. The argument for this is as follows (adapted from [15] to the CC setting).

Consider building a spanning tree F for E' by the standard Union-Find based algorithm *while* performing the sampling. That is, consider each edge e of E sequentially and include it in F *iff* it is sampled *and* it does not form a cycle with edges already in F. Case 1: e forms a cycle. Then e will not appear in E'' due to the contraction. Case 2a: e does not form a cycle, and is sampled. Then e will not appear in E'' due to the contraction (as it is included in F). Case 2b: e does not form a cycle, and is not sampled. Then e may appear in E''. Since the final F is a spanning tree of E', we have $|F| \leq V(E') - 1$ and hence $|F| < V(E)$. Thus, the number of Case 2b edges is a stochastic variable upper-bounded by a negative binomial distribution with $p = 1/2$ and $r = V(E)$ (the number of tails before $V(E)$ heads have appeared when flipping a fair coin). Therefore the expected number of Case 2b edges is at most $V(E)$, implying the same for the expected number of edges in E''.

This statement is analogous to Lemma 2.1 of [15] for the MSF version. The rest of the argument in [15] for the expected cost carries over[2] almost verbatim, with $O(E)$ time substituted by $O(\mathrm{Sort}(E))$ I/Os.

Sibeyn. The MSF algorithm presented in [26] is a surprisingly simple I/O-efficient algorithm. It works by repeatedly letting some node select its minimum incident edge and contracting that edge. These contractions are done in a lazy fashion using the time-forward processing method with node IDs as the "time" dimension. The original algorithm is described in two versions: one using buckets and the other using a priority queue.

We here describe the version based on priority queues. The algorithm represents the undirected edges only in their normalized form (oriented from lower to higher ID). All edges are initially inserted into a priority queue (PQ) which is ordered by source first and edge weight second. This ordering allows the algorithm to perform node contractions by repeatedly extracting the minimum edge in the PQ. When the extracted edge (u, v, w) has a new source u compared to the previous extracted edge, $\{u, v\}$ is the lightest edge incident to u (after the contractions done so far) and is output as an MSF edge. The edge $\{u, v\}$ is then contracted and u's remaining edges are forwarded to (i.e., taken over by) v. In detail, all subsequent edges (u, v', w') with source u extracted from the PQ become $\{v, v'\}$ by inserting $(\min\{v, v'\}, \max\{v, v'\}, w')$ into the PQ, except that edges with $v' = v$ (i.e., self-loops) are skipped. In this MSF version of the algorithm, forwarded edges need to be annotated with the original node IDs of their endpoints, in order for the output to be a correct MSF. When the number V' of source IDs remaining in the PQ can fit in internal memory, i.e., when $V' \leq M$, the rest of the edges in the PQ are extracted and a semi-external version of Kruskal's algorithm is run on them. If using randomized node IDs, the algorithm requires expected $O(E \log(V/V'))$ priority queue operations to contract the original node set V to a smaller node set V' (i.e. for contracting $V - V'$ nodes) [26]. This implies a total cost of $O(\log(V/M) \cdot \mathrm{Sort}(E))$ I/Os.

In our setting, the goal is to compute connected components. This allows the algorithm to be simplified in a number of ways (some described in [25]). The tree that the algorithm outputs should only capture connectivity, hence its edges need not be edges from the original

[2] The argument in [15] allows for using only two initial Borůvka steps. We here follow the description of the CC algorithm in [2], which uses three.

input E, so there is no need to annotate forwarded edges with original node IDs. Additionally, one can choose an *arbitrary* edge out of the "current" source u as the new target to forward edges to. A natural heuristic is to send the information as far forward in time as possible. This is achieved by simply ordering the PQ by source in increasing order and by target in decreasing order as the first edge out of each new source will then go to the furthest neighbor (or known reachable node due to forwarded edges) immediately.

As the final CC information should be represented as a set of stars, some post-processing has to be done on the rooted trees output by the modified node contraction algorithm. As node IDs give a topological ordering of the tree edges, one can simply reverse the tree edges and use time-forward processing in the opposite direction relative to the node contraction phase. This post-processing only incurs $O(\mathrm{Sort}(V))$ additional I/Os.

The bucket version of the algorithm replaces the priority queue with a set of unsorted buckets. Two variants are described in the CC setting in [27, Section 3.4]: one which processes each bucket in internal memory and one which uses the semi-external Union-Find algorithm on each bucket. Choosing bucket sizes ahead of time for the former variant is non-trivial as the density tends to increase during computation. We therefore focus on the latter variant in this paper.

Randomized Borůvka. A standard Borůvka step has a first part where each node selects an incident edge, and a second part where the connected components of this edge set E' are found via time-forward processing and returned as a mapping represented by a star graph.

We now describe a novel randomized method for the second part which is simpler than time-forward processing, at the cost of a worse bound on the contraction factor. In Section 7, we empirically investigate whether this trade-off is beneficial for the overall I/O cost when using Borůvka steps (as part of Borůvka's or Karger-Klein-Tarjan's algorithm).

We consider the selected edge of a node as an outgoing oriented edge. The method is simple: 1) Let each node keep its selected edge with probability p, resulting in the edge set E''. 2) Mark all edges (u, v) in E'' for which E'' contains an edge (w, u), then remove all marked edges to give the final edge set E'''. Step 1) can be done during the edge selection process at no cost, and step 2) can be done in one additional sort and scan step. No (oriented) path in E''' has length more than one, hence E''' is a star graph itself (it represents its own connected components) and can just be returned. Note that while the star-graph computation discussed for the original Borůvka algorithm requires the cycle of a connected component to be a two-cycle, and therefore requires nodes to choose minimum incident edges according to some assigned unique edge weights, this is not the case for our randomized variant.

▶ **Lemma 1.** *E''' has expected size of at least $p(1-p)V(E)$.*

Proof. E' has size $V(E)$, so the expected size of E'' is $pV(E)$. If we for each edge (w, u) in E' count a mark whenever (w, u) was kept *and* (u, v) was kept (where (u, v) is u's chosen edge), then we have an upper bound on the total number of marks (it is an upper bound, as (u, v) could also be counted as marked via another edge (w', u), but (u, v) can only hold one mark). Hence, the expected number of edges removed from E'' to E''' is less than $p^2V(E)$. Thus, the expected size of E''' is at least $p(1-p)V(E)$, which is maximized for $p = 1/2$. ◀

When contracting using the star graph E''', each edge of E''' will remove at least one node, so at least $1/2(1-1/2)V(E) = V(E)/4$ nodes are removed in expectation. Thus, the expected contraction factor is at least $1/(1-1/4) = 4/3$. The contraction for a given graph may be larger than this (just as for standard Borůvka steps and its lower bound of two on the contraction factor). In Section 7, we empirically study contraction factors.

4 Tuning Options

We suggest and experimentally evaluate several variations of the algorithms with potential for impact on their practical running times and I/O costs.

Pipelining. Pipelining is the concept of one algorithmic sub-routine handing its output directly to another sub-routine without storing the intermediate data on disk. Applying this where possible can save I/Os, and our implementation platform STXXL offers tools for this type of programming. Before settling on using it, however, we want to investigate its impact.

Contraction sub-routine. In Borůvka's algorithm, and in the first step of the Karger-Klein-Tarjan algorithm, nodes are contracted. We investigate if time-forward processing based Borůvka steps or the proposed randomized version will be the fastest. The general form of the I/O cost argument in [26, 24] states that if Sibeyn's algorithm is run until the number of nodes has been contracted from V to V', it uses expected $O(\log(V/V') \cdot \mathrm{Sort}(E))$ I/Os. Thus, another possible contraction sub-routine in Karger-Klein-Tarjan is to use Sibeyn.

Omitting node contractions at the root in Karger-Klein-Tarjan. From the details of the cost analysis of Karger-Klein-Tarjan [15], it seems likely that the initial contraction in the root node of its recursion tree will dominate the running time in practice. The asymptotic result of expected $O(\mathrm{Sort}(E))$ cost still holds if this contraction (but not the contractions in other nodes of the recursion tree) is omitted. Then the algorithm will simply start with a scan of the input edge list when sampling edges before the first recursive call. If the returned mapping happens to contract nodes and edges well, the second recursive call will not contribute much to the total I/O cost, either. In this case, the dominating part will be the contraction after the first recursive call, which comprises two sorting steps and two scannings steps on E (if we enter the base case in the second recursive call, we can even save one of the sorting steps, because the edges do not need to be sorted before making the call).

Sampling parameter in Karger-Klein-Tarjan. The original sampling probability for edges before the first recursive call in Karger-Klein-Tarjan was set to $p = 1/2$, but other values are possible. Lowering p makes the first recursive call cheaper, and for denser graphs, we may still have a good effect of the contraction before the second recursive call, because a sparser subset of edges may still span large portions of the connected components. If this turns out to be true, one could make p depend on the density (lower p when the density is higher).

Approximate counting algorithms for size estimation. In the recursive algorithms, there is a need to estimate V in order to know when the semi-external base case can be entered. One idea is to use approximate counting algorithms [3, 5, 10] from the streaming community to determine an estimate on the number of unique nodes in the edge list. In the streaming model this problem is referred to as the *Distinct Elements* problem and most solutions only provide a (δ, ε) guarantee, meaning that the estimate is within a $(1 + \varepsilon)$-multiplicative error with probability at least $(1 - \delta)$. As smaller values of ε and δ require more internal work (mostly in the form of more evaluations of independent hash-functions), we investigate if we can benefit from these methods while staying I/O-bound.

Which neighbor to contract in Sibeyn. In each step of Sibeyn, the MSF version of the algorithm must choose to contract the current node and its neighbor given by its incident edge of minimum weight. In the CC version, it is free to choose any neighbor. As argued

in [27], it may be beneficial to choose the neighbor with largest node ID. We investigate what is the best choice and the gains possible, and we empirically compare choosing a neighbor with largest node ID, a neighbor with smallest node ID, and a random neighbor (which corresponds to the MSF version).

Minimizing the PQ in Sibeyn. When running the PQ version of Sibeyn, we may exploit that the input edges are sorted. This allows us to skip the initial insertion of all edges into the PQ: while running the algorithm, the list of original edges can just be merged with the output of the PQ, which then only needs to contain reinserted edges, not original edges.

Influence of relinking in Sibeyn with buckets. In the bucket version of Sibeyn's algorithm, the connected components for a bucket are computed and signals are sent to later buckets. Sibeyn [27] introduces a *relinking* variant which restructures the signals before sending them to reduce the number of signals between buckets.

5 Implementation

All algorithms are implemented in C++ using the STXXL library [9], which offers highly tuned external memory versions of fundamental algorithmic building blocks like sorting and priority queues. It also supports pipelining, as described in Section 4. The external priority queue of STXXL, which we use in several places in the algorithms implemented, is based on [23].

In order to accommodate different contraction schemes in the contraction of the recursive Karger-Klein-Tarjan algorithm, we implemented a generic framework for performing the sampling, contraction, relabeling and merging during the algorithm's execution. The supported contraction schemes are Sibeyn, Karger-Klein-Tarjan and randomized Borůvka contractions. The framework comes in two flavors: a purely vector-based and a pipelined stream-based implementation. This allows us to evaluate to what degree pipelining is beneficial.

Edge representation. In our implementation we store each undirected edge by its ordered pair (u, v) where $u < v$. For sorted edges we additionally employ a more I/O-efficient data structure: consecutive edges with the same source u are compressed to a single entry u followed by all its adjacent nodes and a delimiter.

Data structures. The pipelined implementations make use of several STXXL data structures. In these, generated data is not saved in an explicit vector but fed to a container which then functions as a data stream with read-only access. An example of this is STXXL's *sorter*: in the first phase, items are pushed into the write-only sorter in an arbitrary order by some algorithm. After an explicit switch, the filled data structure becomes read-only and the elements are provided as a sorted stream which can be rewound at any time. While a sorter is functionally equivalent to filling, sorting and reading back an external memory vector, the restricted access model reduces constant factors in the running time and I/O cost [6].

Semi-external base case. While we assume that the number of nodes in the original input is known exactly, this is not necessarily true for recursive calls. As we aim to switch to a semi-external base case algorithm, keeping track of the number of remaining nodes is essential. For the Karger-Klein-Tarjan algorithm, the node contraction step contracts a

specified number of nodes and as such, a good estimate for the number of nodes remaining after initial contraction is simply the original node count minus the number contracted[3]. The same holds for the contracted edge set passed on to the second recursive call: the number of connected components returned from the first recursive call is known and corresponds to the number additional nodes contracted. This leaves the first recursive call operating on the sampled edges E'. The number of nodes here is trivially bounded both by the bound known before sampling and by $2|E'|$. The latter bound can be improved somewhat; as edges are kept sorted, the number of unique sources can be counted while sampling and only $|E'|$ is added for an upper bound. Taking the best of these bounds at different stages, we maintain an upper bound on the node count. By using these upper bounds we can save I/Os in the relabeling as relabeled edges may immediately be piped into the semi-external base case without the otherwise required final sorting step. Note that while computing the exact number of nodes requires only a few scanning and sorting steps, this is too costly in practice for competitive results.

6 Graph classes

For our experiments, we use a variety of different synthetic graph models. We consider four types: the Gilbert type classic random graphs, random geometric and random hyperbolic graphs, both belonging to the class of spatial network models, and finally deterministically generated grid graphs. Using scalable graph generators, we generate fully external ($M < V$) graphs with a range of different parameters. For a recent overview of such generators, see [21].

Gilbert graphs. In the $\mathcal{G}(n, p)$ model of Gilbert [12], each edge is present independently with probability p. The $\mathcal{G}(n, p)$ model can generate graphs with a varying number of connected components for sufficiently small p. It is widely used in empirical work, but its degree distribution is often considered atypical compared to real-world instances.

Random Geometric graphs. Random Geometric graphs (RGGs) [13, 19] are a simple case of spatial networks where graphs are projected onto Euclidean space. In RGGs n points are placed uniformly at random into a d-dimensional unit-cube $[0, 1)^d$ where any two points are connected if their Euclidian distance is below a given threshold r. To generate graphs in this model, we use the generator available in KaGen [11].

Random Hyperbolic graphs. Random Hyperbolic graphs (RHGs) [16, 14] are a special case of spatial networks where graphs are projected onto hyperbolic space. We describe the threshold model, the simplest RHG variant [14]. The points are randomly placed onto a two-dimensional disk in hyperbolic space where the radial probability density function increases exponentially towards the border. The angular coordinate is sampled uniformly at random from $[0, 2\pi)$ and points are connected if their hyperbolic distance is less than a given threshold R. The density of points near the center is controllable by setting a dispersion parameter α. One interesting feature of RHGs is that the node degrees follow a power law distribution which is often found in real-world graph instances, in particular when generated via human activities and choices. In the threshold model the exponent is $\gamma = 1 + 2\alpha$ with high probability [14]. To generate graphs in this model, we use the HyperGen generator [20].

[3] This gives an exact count except when a connected component is contracted to a singleton – at which point it will not appear in the edge list.

Grid graphs. We consider two different types of square grid graphs. In both versions, the nodes are seen as points in a two-dimensional grid; (x, y) for $1 \leq x \leq w$ and $1 \leq y \leq h$. For the simpler version, nodes are connected horizontally and vertically to their neighbors. All nodes except for boundary nodes thus have degree 4. To achieve higher degree, we additionally consider generalized grids in which nodes are connected to all nodes within distance d under the infinity norm. That is, node (x, y) has edges to nodes $(x + i, y + j)$ where $-d \leq i \leq d$ and $-d \leq j \leq d$, except where this exceeds the grid boundary. Internal nodes in these graphs have degree $4d(d + 1)$. To investigate the effects of increasing the number of components, we additionally generate graphs which we refer to as *cubes* consisting of multiple disjoint layers, each of which is a generalized grid graph.

7 Experiments

Our experiments were carried out in two phases. In the first phase, we investigated the impact of the various algorithmic variants and proposals for tuning described in Sections 3–5. This was done on subsets of the test graphs of Section 6 and selected other test cases. The aim of this phase was to develop a set of well-engineered implementations of the most promising contenders. In the second phase, we then compared those on a large set of test graphs of Section 6 – the compute time of this phase alone comprised one third of a year. Below, we describe our experimental setup and our learnings from each of the two phases. For space reasons, we mainly include plots for the second phase. The full set of plots are in Appendix A (in the plots, the numbers V and E are denoted by n and m, respectively).

7.1 Experimental setup

The experiments were run on individual nodes of the Goethe-HLR cluster at Goethe University Frankfurt, as this allowed us to run many experiments simultaneously (note that our algorithms all are sequential, parallel algorithms for CC are beyond the scope of this paper). The nodes each have Intel Xeon Skylake Gold 6148 CPUs and 192 GB of RAM. Each node has a HGST Ultrastar HUS726020ALA610 hard drive which was used for the STXXL disk file. The code was compiled using GCC version 8.3.1 with the optimization parameters `O3` and `march=native`.

In each run, the input graph was first loaded onto the local hard drive in the appropriate STXXL data structure: an edge stream for the stream-based implementations and an STXXL vector for the vector-based implementations. The threshold for switching to the semi-external base case was for all the algorithms set to 33,554,432 nodes which corresponds to 256 MiB of node IDs. To capture wall-time and I/O volume, we used the `iostats` module provided by the FOXXLL library (a component of STXXL). The main timing plots in Figures 10–17 show the wall time (bars) and total I/O volume (bytes read plus bytes written during the execution of the algorithm) reported by the `iostats`.

To keep the combined compute time of the experiments from becoming infeasible (even when executing experiments in parallel on a cluster), we reduced the RAM used by the CPUs to a few GB, which allowed us values of V/M up to 80 and graph densities E/V up to 20 (although not both maximal values at the same time) while keeping individual experiments under half a day of compute time. Our hypothesis was that if the algorithms are I/O-bound, the relative running times of the algorithms would stay approximately the same even if moving to larger sizes of RAM and from the hard disks of the cluster nodes to solid-state disks. With the set of final contenders, we conducted experiments on selected graph classes

on a single machine having 16 GB of RAM and a RAID with six solid-state disks of 480 GB each. Those experiments confirmed our hypothesis, as the relative running times changed less than 20% in almost all cases tested.

To limit the amount of memory used on the cluster nodes, we limited the internal memory allowed for STXXL primitives used (sorting streams and priority queues were limited to 1 GB of RAM each). With the base case threshold (accounting for overhead), and the above limits, the implementations should be able to run with approximately 2 GB of memory. We did not have a mechanism to enforce a strict bound on the memory actually allocated, but monitored the amount of RAM actually used, which was in the range of 2 GB to 5 GB. To force disk accesses rather than additional buffering, the `direct` flag was used for the STXXL disk file.

7.2 Phase 1 – Initial Findings

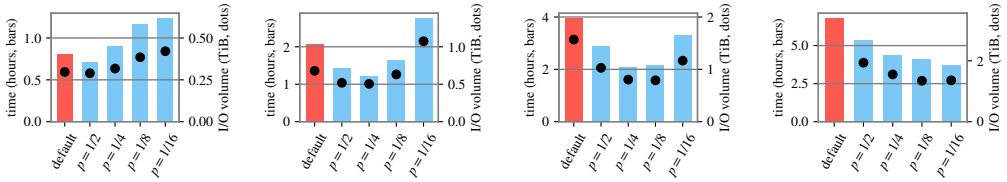
We now describe our main findings in phase one of our experiments. Unless otherwise mentioned, the measure compared is wall clock time.

Randomized Borůvka and Borůvka

For our suggestion for randomized Borůvka steps, we first investigated the impact on the observed contraction ratio of a number of different edge representations and of various sampling parameters. On most graph classes, sampling parameters much closer to one than to 1/2 gave better contraction ratios (see Figure 5), in line with Lemma 1 only giving a lower bound. There was correlation among the graph classes between increased contraction efficiency of the randomized Borůvka steps and increased contraction efficiency of standard Borůvka (past the lower bound of two on the ratio). However, the ratio was consistently worse for the randomized version, and its simpler code did not make up for this when considering the total time of Borůvka's algorithm. Additionally, both of the two versions of Borůvka's algorithm were clearly worse than Sibeyn's algorithm based on PQs, both before and after adding pipelining. For instance, when doing node contraction until the base case is reached, we found that Sibeyn's algorithm was approximately 59% faster than ordinary Borůvka and we likewise found that one variant of our Karger-Klein-Tarjan implementation using Sibeyn's algorithm for node contraction was around 58% faster than one using the randomized Borůvka steps (results vary across graphs, numbers given here are averages). We therefore left Borůvka's algorithm out of the final race.

Pipelining

Adding pipelining in STXXL turned out to improve our implementations of Sibeyn and of Karger-Klein-Tarjan. Introducing pipelining (including compressed edge streams) reduced the running time of one of our Karger-Klein-Tarjan variants approximately 73% on average over a simple version based on STXXL vectors. Even for our Sibeyn implementation (where even a simple implementation incurs much less copying), introducing pipelining improved the running time by approximately 10% on average. The tunings to the PQ based version of Sibeyn suggested in Section 4, e.g. reducing the processed volume of edges in the PQ, turned out to be beneficial, lowering running time by an additional approximately 29% on average.



(a) $m/n = 2.04$ **(b)** $m/n = 5.00$ **(c)** $m/n = 10.00$ **(d)** $m/n = 20.00$

Figure 2 (Subset of Figure 10) Running times and I/O volumes for $\mathcal{G}(n, p)$ graphs with a node set size of 5GiB and varying density. The *default* variant always contracts and has a sampling probability of $p = 1/2$. The remaining variants skip contraction in the root and have fixed sampling probabilites. For $m/n = 20$, the *default* variant exceeded the local hard disk's capacity leading to a halt in the algorithm's execution. We thus only report the elapsed wall time up until that point.

Approximate counting

For the estimation of V using approximate counting algorithms, we tested the FM algorithm by Flajolet and Martin [10]. In essence, the FM algorithm computes for a given input stream an estimate of its number of distinct elements. For this, every input element is mapped by a hash function and incorporated into a later modified and returned proxy value. Due to the output variance being intolerably large, standard median-of-means techniques are employed which in turn, however, require more independent hash functions.

For several graph classes, we employed the FM algorithm in the sampling step of the Karger-Klein-Tarjan algorithm with an increasing number of hash functions. To accurately assess the returned estimates we separately ran the Karger-Klein-Tarjan algorithm with the same seed and explicitly counted the correct number of nodes in each sampling step.

We found that the number of hash functions needed in order to get a useful precision in the estimate was so high that it impacted the running time. Additionally, the errors in the estimation are two-sided, which fits badly with the fact that invoking a Union-Find based base case when not actually being semi-external will have disastrous effects on the running time. Combined, this made us decide not to include this method in the final experiments.

Karger-Klein-Tarjan

For the contraction steps in the Karger-Klein-Tarjan algorithm we tried both standard and randomized Borůvka steps, as well as the PQ based version of Sibeyn, and the latter proved to be the better option.

When varying the sampling parameter p in Karger-Klein-Tarjan, we observed a rather clear correlation (see Figure 2 and Figures 10–16): the best choice for both I/O volume and running time seems to be p equal to the inverse density V/E of the input graph, likely for the reasons conjectured in Section 4: a sampled subset of edges containing around V nodes will often by itself contract the node set considerably, while the left recursion will be cheap if this is achieved for small p, which may happen more often for high densities.

Also, when visualizing the recursion trees, a clear pattern was a balanced tree for this value of p, whereas quite strongly left-leaning and right-leaning trees appeared for larger and smaller values, respectively. Profiling of the distribution of time spent in the nodes of the recursion trees showed the root to be dominating, which is aligned with the analysis in [15]. Often, the contraction step was dominating (as can also be seen in Figure 2). There was also a small tendency for Karger-Klein-Tarjan to improve relative to the other algorithms when V grew compared to M (for fixed density and graph class). These observations (which are

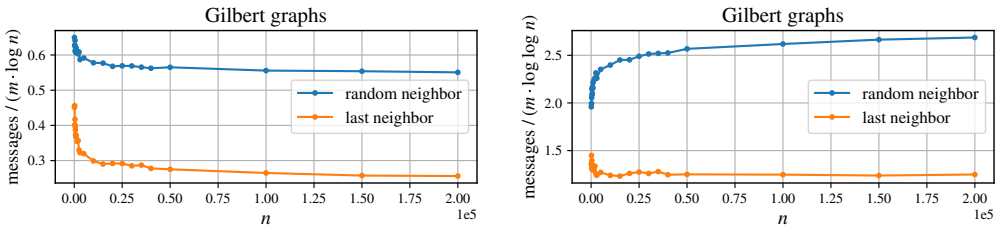


Figure 3 (Copy of Figure 7) Number of forwarded messages divided by $m \log n$ (left) or $m \log \log n$ (right) for Gilbert graphs for increasing values of n. The value p is chosen s.t. a density of five is fixed. In (left) we observe that the total number of produced messages is dominated by $m \log n$ whereas in (right) we see that the volume asymptotically matches with $m \log \log n$ if messages are forwarded to the last neighbor.

visible in the plots in Figures 10–17) inspired us to implement variants of the Karger-Klein-Tarjan which do not use contraction at the root, and adaptive variants which in all recursion tree nodes choose contractions only for low (estimated) density and also choose a sampling parameter close to V/E.

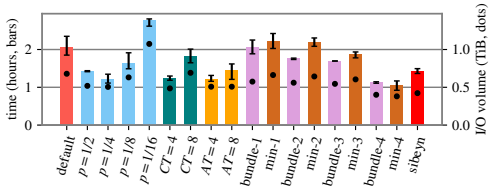
Choice of contraction target in Sibeyn

Perhaps our most interesting observation in phase one was the influence on Sibeyn of the choice of which neighbor to contract (see Section 4). We tried the choices of nearest, random, and farthest in node-ID order (i.e., the "timeline" in the time-forward processing by the PQ). Of these, the random choice intuitively can be expected to behave like the MSF variant of the algorithm (where each node must choose the neighbor of its lightest incident edge, and where the node IDs are randomly permuted). As exemplified by the first plot of Figure 6, where a message is a PQ entry (i.e., an edge in its original form or a later replaced form) each inducing $O(1)$ PQ operations, the choice of nearest is by far the worst and was not considered again. On the other hand, farthest is always better than random (see rest of plots in Figure 6).

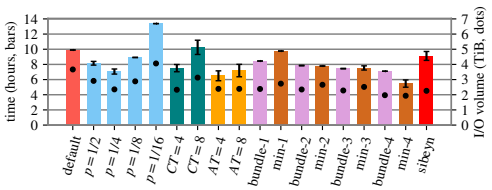
This effect was first studied in [27], where an expected bound of $O(E \log \log(V))$ messages was claimed (but the proof omitted) for Gilbert graphs. As seen in Figure 3, we here verify that claim empirically, and also demonstrate that it does not hold for the random choice. Even more interesting, for random grid graphs and random hyperbolic graphs, the empirical evidence even suggests a better bound of expected $O(E)$ (Figures 8 and 9). These findings suggest that Sibeyn in practice is strictly faster for CC than for MSF, and that it for the former may often run in cost $O(\mathrm{Sort}(E))$. Additionally, the bulk of the messages seem concentrated very late in the time-forward process, which in the external version is preempted by entering the semi-external Union-Find case, which in turn has lower overhead per edge/message than a PQ. Combined with the general simplicity of Sibeyn, these findings indicate that it may be very hard to surpass. For our final Sibeyn implementations, we naturally used the farthest neighbor choice.

7.3 Phase 2 – Final Algorithms

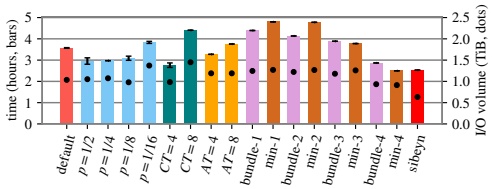
For the second and final phase of experiments we selected the following algorithms (with implementation choices fixed as described above).



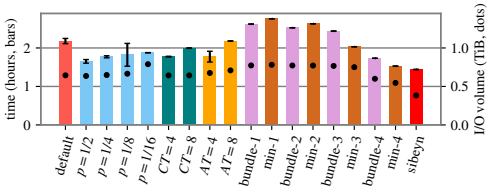
(a) Gilbert graph, smallest node set, $m/n = 5$



(b) Gilbert graph, largest node set, $m/n = 5$



(c) Random Geometric graph



(d) Random Hyperbolic graph

Figure 4 (Copies of Figure 10b, Figure 13b, Figure 16d and Figure 17c)
Running times and I/O volumes for two Gilbert graphs and the two largest generated RGG and RGH instances.

- Karger-Klein-Tarjan in several versions: One with contractions in all recursion tree nodes and fixed sampling parameter $p = 1/2$. Four versions omitting contraction at the root of the recursion tree and having fixed sampling parameters of 1/2, 1/4, 1/8, and 1/16, respectively. Two adaptive versions which in each node of the recursion tree choose (among the values above) the sampling parameter closest to the estimated inverse density V/E of the input graph of the node, and also omit contraction if the estimated density E/V is below a fixed threshold of 4 or 8, respectively. Two similar adaptive versions where instead the threshold is 4 or 8, respectively, when the estimated V is close to the base case, but tends to 2 for growing V. These nine algorithms are denoted *default*, $p = 1/2$, $p = 1/4$, $p = 1/8$, $p = 1/16$, $CT = 4$, $CT = 8$, $AT = 4$, and $AT = 8$, respectively.
- Sibeyn's algorithm based on buckets, using Union-Find for solving CC in buckets, as described in [27] (where buckets are called bundles). We tried four increasing bucket sizes, all without and with relinking to minimize edges straddling buckets (Section 4 and [27]). These eight algorithms are denoted *bundle-x* and *min-x* for $x = 1, 2, 3, 4$.
- The basic Sibeyn using a PQ. This algorithm is denoted *sibeyn*.

Comparing Karger-Klein-Tarjan variants

We find based on Figures 10–17 that among the Karger-Klein-Tarjan variants the adaptive ones are either winning or performing close to the best variant. This can be observed for all considered graph classes (see Figure 4 for an overview). In almost all cases, fixed contraction thresholds tend to perform better than adaptive ones. Further, setting the threshold to a small value seems preferable. This behaviour is consistent when increasing the number of nodes while keeping the density fixed (see for instance Figure 10b, Figure 11b, Figure 12b and Figure 13b) where it is clear that relative performances remain unchanged.

The good performance of these adaptive variants and the comparatively generally weak performance of the *default* variant support our claim that contractions can be intolerably costly.

Comparing Sibeyn variants

We find that both versions of Sibeyn's algorithm are strong contenders. While the PQ based Sibeyn algorithm generally performs better on low density graphs (see Figure 4c and Figure 4d), its relative performance gets worse with increasing V (see Figure 4a, Figure 4b and Figures 10–13). Additionally, while the overall I/O volume may be near optimal (see Figure 4d), the achieved wall clock time does not always reflect this, indicating that the I/Os incurred by the PQ may be more costly than those for sorting.

In comparison, the bucket based Sibeyn algorithm performs consistently among the studied graph classes (see Figure 4 and Figures 10–17). We notice two clear trends, larger buckets generally perform better and adding relinking typically improves performance for graphs with higher densities.

8 Conclusion

The results of our experiments in phase two on the above set of algorithms can be seen in Figures 10–17. Sibeyn's algorithm is a strong contender. One reason is that it is very simple, using essentially only a priority queue (or repeated Union-Find in the bucket version). A tuned implementation of external priority queues can be highly efficient: our measurements on STXXL show that sorting by its priority queue is less than a factor of 2.5 slower than its sorting routine. Another reason is that for its CC variant, the choice of farthest neighbors seems to lower the number of messages generated to essentially linear (with the exact observed bound depending on the graph class) in E, which translates into a similar number of priority queue operations. Very few sorting and scanning steps on the input edge list can be performed by a competing algorithm before it will lose to Sibeyn.

Still, with the right tunings, the Karger-Klein-Tarjan algorithm can be implemented to be competitive in many cases. The best Karger-Klein-Tarjan variant often either wins over PQ based Sibeyn, but not bucket based Sibeyn, or vice versa. If nothing is known about the graph type and density, an adaptive variant such as $CT = 4$ may be a robust choice. In general, higher graph density seems to benefit Karger-Klein-Tarjan relative to Sibeyn. If choosing the bucket based Sibeyn variant, using the largest bucket size is clearly preferable (and often the min variant has a slight advantage). Borůvka's algorithm was not able to compete with neither Sibeyn nor Karger-Klein-Tarjan.

Natural future work suggested by this work include: 1) To investigate theoretically the observed positive effects on Sibeyn of the farthest neighbors choice. As demonstrated in Figures 7–9, different results seem plausible for different graph classes. 2) To compare empirically also the MSF versions of the algorithms.

References

1 Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988. doi:10.1145/48529.48535.

2 Susanne Albers, Andreas Crauser, and Kurt Mehlhorn. Lecture notes on algorithms for very large data sets. https://web.archive.org/web/19970816002522/http://www.mpi-sb.mpg.de/~crauser/Plan.ps.gz, 1997.

3 Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999. doi:10.1006/jcss.1997.1545.

4 Lars Arge, Gerth Stølting Brodal, and Laura Toma. On external-memory MST, SSSP and multi-way planar graph separation. *J. Algorithms*, 53(2):186–206, 2004. doi:10.1016/j.jalgor.2004.04.001.

5 Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In José D. P. Rolim and Salil P. Vadhan, editors, *Randomization and Approximation Techniques, 6th International Workshop, RANDOM 2002, Cambridge, MA, USA, September 13-15, 2002, Proceedings*, volume 2483 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berling Heidelberg, 2002. doi:10.1007/3-540-45726-7_1.

6 Andreas Beckmann, Roman Dementiev, and Johannes Singler. Building a parallel pipelined external memory algorithm library. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pages 1–10. IEEE Computer Society, 2009. doi:10.1109/IPDPS.2009.5161001.

7 Alka Bhushan and Sajith Gopalan. An I/O efficient algorithm for minimum spanning trees. In Zaixin Lu, Donghyun Kim, Weili Wu, Wei Li, and Ding-Zhu Du, editors, *Combinatorial Optimization and Applications - 9th International Conference, COCOA 2015, Houston, TX, USA, December 18-20, 2015, Proceedings*, volume 9486 of *Lecture Notes in Computer Science*, pages 499–509. Springer, 2015. doi:10.1007/978-3-319-26626-8_36.

8 Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey S. Vitter. External-memory graph algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1995. San Francisco, California, USA*, pages 139–149. SIAM, Philadelphia, PA, USA, 1995. URL: http://dl.acm.org/doi/10.5555/313651.313681.

9 Roman Dementiev, Lutz Kettner, and Peter Sanders. STXXL: standard template library for XXL data sets. *Softw. Pract. Exp.*, 38(6):589–637, 2008. doi:10.1002/spe.844.

10 Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985. doi:10.1016/0022-0000(85)90041-8.

11 Daniel Funke, Sebastian Lamm, Ulrich Meyer, Manuel Penschuck, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free massively distributed graph generation. *J. Parallel Distributed Comput.*, 131:200–217, 2019. doi:10.1016/j.jpdc.2019.03.011.

12 Edgar N. Gilbert. Random graphs. *Ann. Math. Statist.*, 30(4):1141–1144, December 1959. doi:10.1214/aoms/1177706098.

13 Edgar N. Gilbert. Random plane networks. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):533–543, 1961. doi:10.1137/0109045.

14 Luca Gugelmann, Konstantinos Panagiotou, and Ueli Peter. Random hyperbolic graphs: Degree sequence and clustering. In *Proceedings of the 39th International Colloquium Conference on Automata, Languages, and Programming - Volume Part II, ICALP 2012, Warwick, UK, July 9-13, 2012*, page 573–585. Springer Berlin Heidelberg, 2012. doi:10.1007/978-3-642-31585-5_51.

15 David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, March 1995. doi:10.1145/201019.201022.

16 Dmitri Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguñá. Hyperbolic geometry of complex networks. *Phys. Rev. E*, 82:036106, September 2010. doi:10.1103/PhysRevE.82.036106.

17 Anil Maheshwari and Norbert Zeh. A survey of techniques for designing I/O-efficient algorithms. In Ulrich Meyer, Peter Sanders, and Jop F. Sibeyn, editors, *Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar, March 10-14, 2002]*, volume 2625 of *Lecture Notes in Computer Science*, pages 36–61. Springer, 2002. doi:10.1007/3-540-36574-5_3.

18 Kamesh Munagala and Abhiram G. Ranade. I/O-complexity of graph algorithms. In Robert Endre Tarjan and Tandy J. Warnow, editors, *Proceedings of the 10th Annual ACM-SIAM*

Symposium on Discrete Algorithms, 17-19 January 1999, Baltimore, Maryland, USA, pages 687–694. ACM/SIAM, 1999. URL: https://dl.acm.org/doi/10.5555/314500.314891.

19 Mathew D. Penrose. *Random Geometric Graphs*. Oxford University Press, 2003. doi:10.1093/acprof:oso/9780198506263.001.0001.

20 Manuel Penschuck. Generating practical random hyperbolic graphs in near-linear time and with sub-linear memory. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors, *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*, volume 75 of *LIPIcs*, pages 26:1–26:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.SEA.2017.26.

21 Manuel Penschuck, Ulrik Brandes, Michael Hamann, Sebastian Lamm, Ulrich Meyer, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in scalable network generation. *CoRR*, abs/2003.00736, 2020. arXiv:2003.00736.

22 Seth Pettie and Vijaya Ramachandran. An optimal minimum spanning tree algorithm. *J. ACM*, 49(1):16–34, 2002. doi:10.1145/505241.505243.

23 Peter Sanders. Fast priority queues for cached memory. *ACM J. Exp. Algorithmics*, 5:7, 2000. doi:10.1145/351827.384249.

24 Dominik Schultes. *External Memory Minimum Spanning Trees*. Bachelor thesis, Universität des Saarlandes, 2003. URL: http://algo2.iti.kit.edu/schultes/emmst/emmst_short.pdf.

25 Dominik Schultes. External memory spanning forests and connected components, 2003. URL: http://algo2.iti.kit.edu/dementiev/files/cc.pdf.

26 Jop Sibeyn, Roman Dementiev, Peter Sanders, and Dominik Schultes. Engineering an external memory minimum spanning tree algorithm. In Jean-Jacques Levy, John C. Mitchell, and Ernst W. Mayr, editors, *Exploring New Frontiers of Theoretical Informatics*, volume 155, pages 195–208. Kluwer Academic Publishers, Boston, 2004. doi:10.1007/1-4020-8141-3_17.

27 Jop F. Sibeyn. External connected components. In Torben Hagerup and Jyrki Katajainen, editors, *Algorithm Theory - SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory, Humlebæk, Denmark, July 8-10, 2004*, volume 3111, pages 468–479. Springer Berlin Heidelberg, 2004. doi:10.1007/978-3-540-27810-8_40.

28 Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975. doi:10.1145/321879.321884.

A Plots

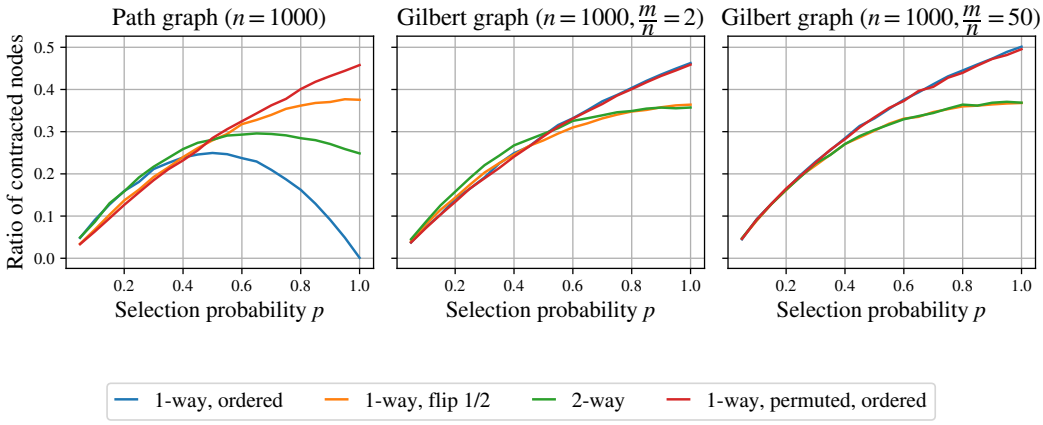


Figure 5 Contraction ratios achieved by variants of a randomized Borůvka step for varying p and varying edge representations. For Gilbert graphs, the contraction ratio increases with increasing selection probability p where the best candidates are the ordered variants. For path graphs, the variants without randomness peak and start to perform worse.

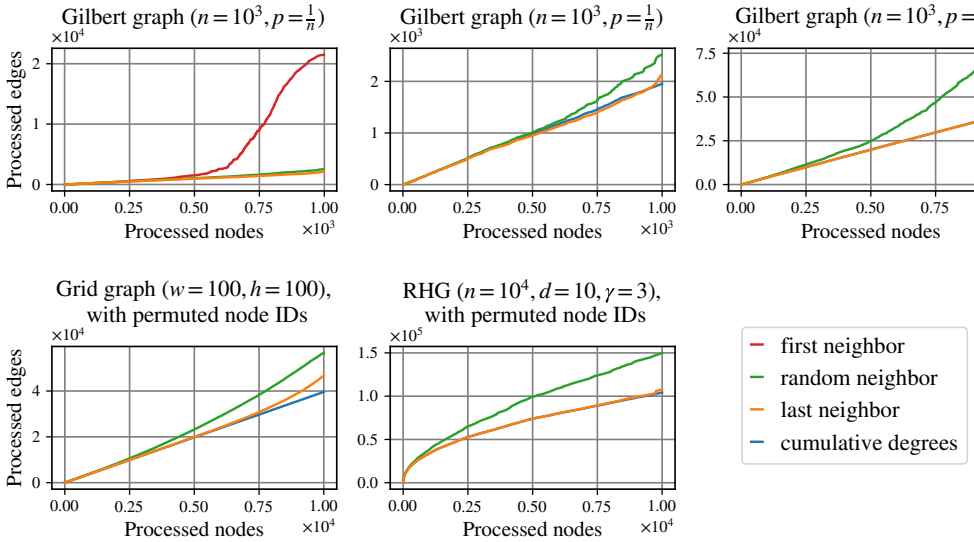


Figure 6 Message volume of forwarded messages for different graphs depending on the contraction strategy. Sibeyn's algorithm processes significantly more edges (priority queue messages) when messages are sent to the first neighbor (see first plot). In comparison, sending messages to the last neighbor produces volumes very close to the baseline (cumulative degrees) and always performs better than sending to a random neighbor.

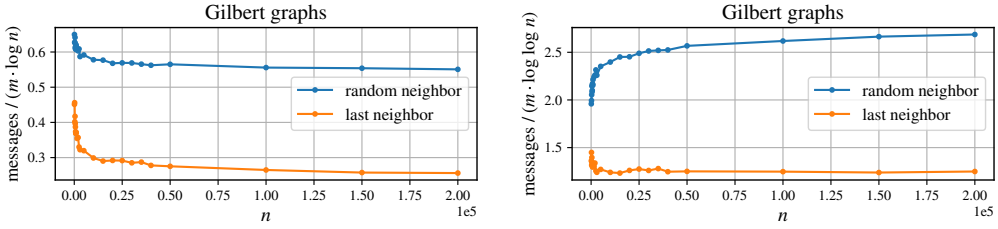


Figure 7 Number of forwarded messages divided by $m \log n$ (left) or $m \log \log n$ (right) for Gilbert graphs for increasing values of n. The value p is chosen s.t. a density of five is fixed. In (left) we observe that the total number of produced messages is dominated by $m \log n$ whereas in (right) we see that the volume asymptotically matches with $m \log \log n$ if messages are forwarded to the last neighbor.

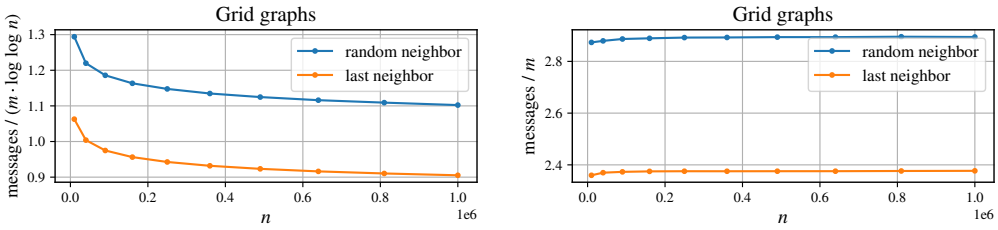


Figure 8 Number of forwarded messages divided by $m \log \log n$ (left) or m (right) for quadratic grid graphs for increasing values of n. By construction, these have density approximately two. In (left) we observe that the total number of produced messages is dominated by $m \log \log n$ whereas in (right) we see that the volume asymptotically matches with m.

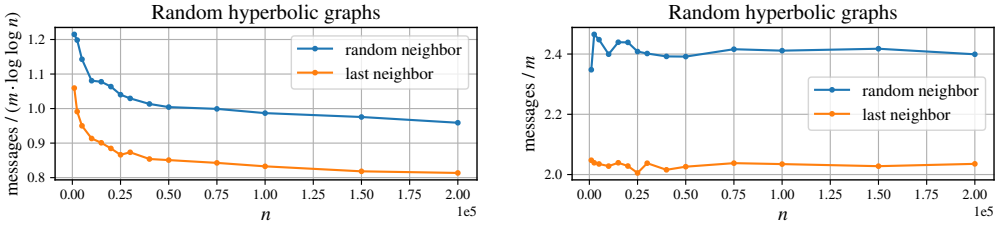
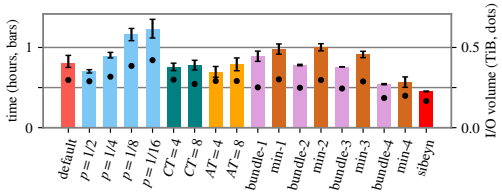
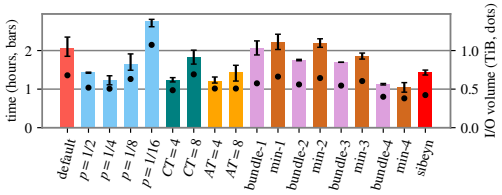


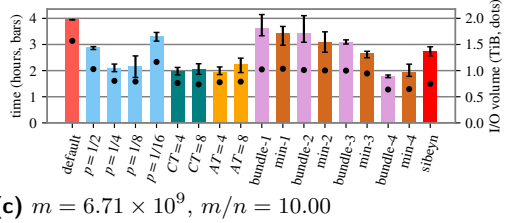
Figure 9 Number of forwarded messages divided by $m \log \log n$ (left) or m (right) for RHGs for increasing values of n. The degree parameter is set to 10 for all of these, yielding an approximate density of five. The degree exponent is set to 3. In (left) we observe that the total number of produced messages is dominated by $m \log \log n$ whereas in (right) we see that the volume asymptotically matches with m.



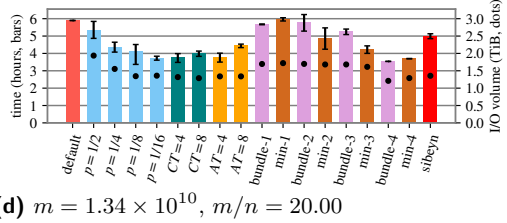
(a) $m = 1.34 \times 10^9$, $m/n = 2.04$



(b) $m = 3.36 \times 10^9$, $m/n = 5.00$



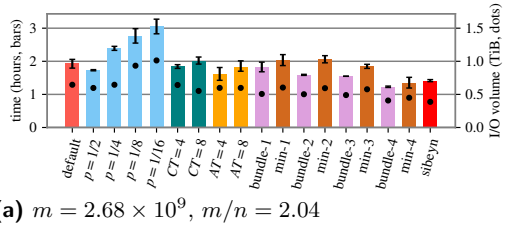
(c) $m = 6.71 \times 10^9$, $m/n = 10.00$



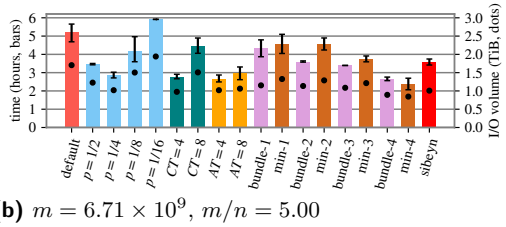
(d) $m = 1.34 \times 10^{10}$, $m/n = 20.00$

Figure 10 Running times and I/O volumes for $\mathcal{G}(n, p)$ graphs with a node set size of 5GiB and varying density. For $m/n = 20$, the *default* variant exceeded the local hard disk's capacity leading to a halt in the algorithm's execution. We thus only report the elapsed wall time up until that point. The considered algorithms are in fixed order from left to right:

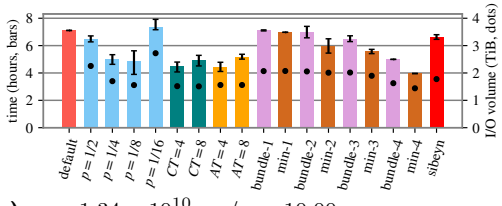
default:	fixed sampling $p = 1/2$, always contract
$p = 1/x$:	fixed sampling $p = 1/x$, always contract except in root
$CT = x$:	adaptive sampling, contract if estimated density below fixed threshold x
$AT = x$:	adaptive sampling, contract if estimated density below adaptive threshold x .
bundle-x:	Sibeyn's algorithm based on buckets, without linking
min-x:	Sibeyn's algorithm based on buckets, with linking
sibeyn:	Sibeyn's algorithm based on priority-queues



(a) $m = 2.68 \times 10^9$, $m/n = 2.04$

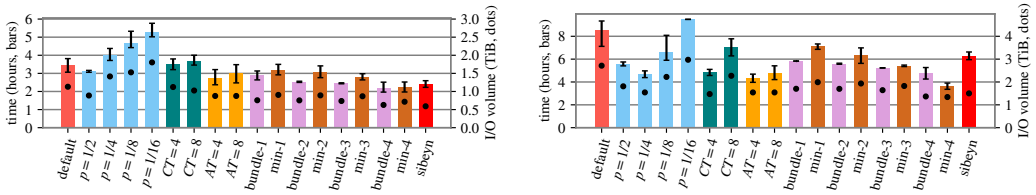


(b) $m = 6.71 \times 10^9$, $m/n = 5.00$



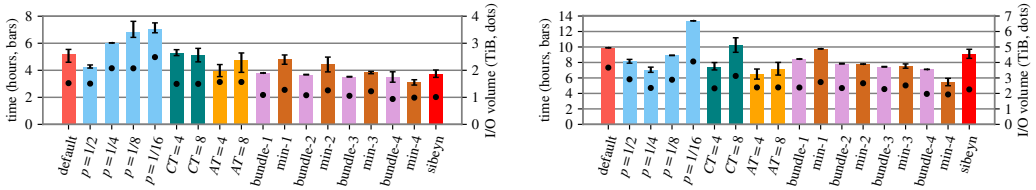
(c) $m = 1.34 \times 10^{10}$, $m/n = 10.00$

Figure 11 Running times and I/O volumes for $\mathcal{G}(n, p)$ graphs with a node set size of 10GiB and varying density. For $m/n = 10$, the *default* variant exceeded the local hard disk's capacity leading to a halt in the algorithm's execution. We thus only report the elapsed wall time up until that point.



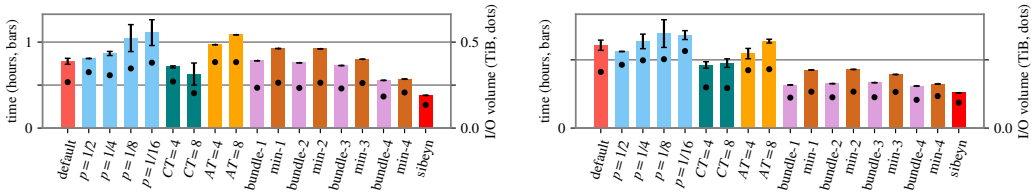
(a) $m = 4.03 \times 10^9$, $m/n = 2.04$ **(b)** $m = 1.01 \times 10^{10}$, $m/n = 5.00$

Figure 12 Running times and I/O volumes for $\mathcal{G}(n, p)$ graphs with a node set size of 15GiB and varying density.



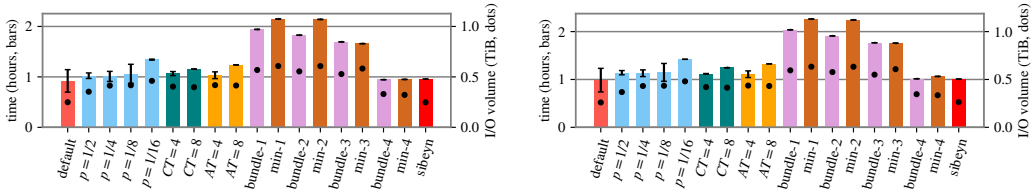
(a) $m = 5.37 \times 10^9$, $m/n = 2.04$ **(b)** $m = 1.34 \times 10^{10}$, $m/n = 5.00$

Figure 13 Running times and I/O volumes for $\mathcal{G}(n, p)$ graphs with a node set size of 20GiB and varying density.



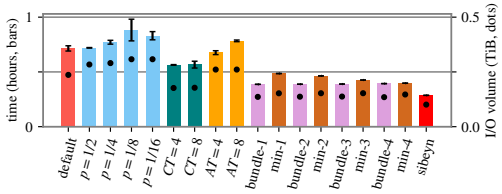
(a) $m = 1.34 \times 10^9$, $m/n = 2.00$ **(b)** $m = 1.34 \times 10^9$, $m/n = 1.00$

Figure 14 Running times and I/O volumes for (a) a grid graph with $(w, h) = (25\,905, 25\,905)$ and (b) a path graph. For both instances the parameters were chosen to generate a 20GiB graph. Node IDs are permuted.

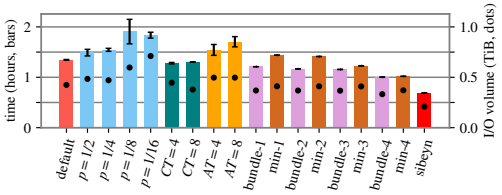


(a) $m = 3.89 \times 10^9$, $m/n = 11.96$ **(b)** $m = 4.05 \times 10^9$, $m/n = 12.00$

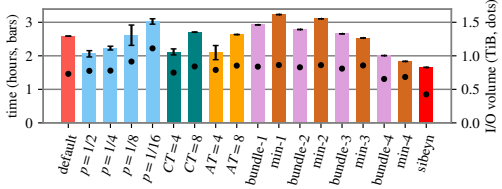
Figure 15 Running times and I/O volumes for cubes with the parameters (a) one layer and $(w, h, d) = (18\,000, 18\,000, 2)$ and (b) 100 layers and $(w, h, d) = (2600, 1300, 2)$.



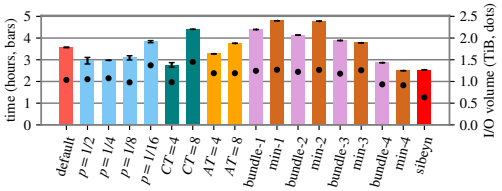
(a) $n = 9.18 \times 10^8$, $m = 1.04 \times 10^9$, $m/n = 1.14$



(b) $n = 1.06 \times 10^9$, $m = 2.62 \times 10^9$, $m/n = 2.46$

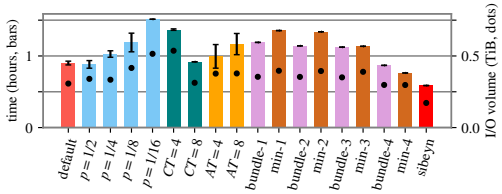


(c) $n = 1.07 \times 10^9$, $m = 5.48 \times 10^9$, $m/n = 5.11$

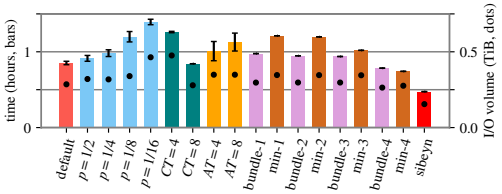


(d) $n = 1.07 \times 10^9$, $m = 8.14 \times 10^9$, $m/n = 7.59$

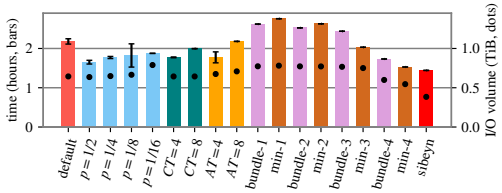
Figure 16 Running times and I/O volumes for RGGs with roughly $n = 2^{30}$ and varying density. Node IDs are permuted.



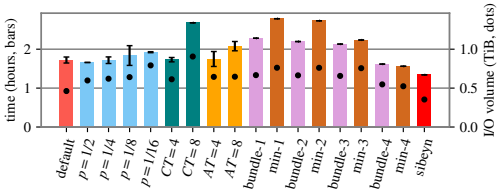
(a) $n = 6.54 \times 10^8$, $m = 2.61 \times 10^9$, $m/n = 3.99$, $\gamma = 3$



(b) $n = 6.46 \times 10^8$, $m = 2.36 \times 10^9$, $m/n = 3.65$, $\gamma = 4$



(c) $n = 6.70 \times 10^8$, $m = 5.57 \times 10^9$, $m/n = 8.30$, $\gamma = 3$



(d) $n = 6.70 \times 10^8$, $m = 5.22 \times 10^9$, $m/n = 7.80$, $\gamma = 4$

Figure 17 Running times and I/O volumes for RHGs with roughly $n = 2^{30}$, degree exponent $\gamma \in \{3, 4\}$ and varying density. Node IDs are permuted.