A Simple and Practical Linear-Work Parallel Algorithm for Connectivity

Julian Shun
Carnegie Mellon University
jshun@cs.cmu.edu

Laxman Dhulipala
Carnegie Mellon University
ldhulipa@andrew.cmu.edu

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

ABSTRACT

Graph connectivity is a fundamental problem in computer science with many important applications. Sequentially, connectivity can be done in linear work easily using breadth-first search or depth-first search. There have been many parallel algorithms for connectivity, however the simpler parallel algorithms require super-linear work, and the linear-work polylogarithmic-depth parallel algorithms are very complicated and not amenable to implementation. In this work, we address this gap by describing a simple and practical expected linear-work, polylogarithmic depth parallel algorithm for graph connectivity.

Our algorithm is based on a recent parallel algorithm for generating low-diameter graph decompositions by Miller et al. [44], which uses parallel breadth-first searches. We discuss a (modest) variant of their decomposition algorithm which preserves the theoretical complexity while leading to simpler and faster implementations. We experimentally compare the connectivity algorithms using both the original decomposition algorithm and our modified decomposition algorithm. We also experimentally compare against the fastest existing parallel connectivity implementations (which are not theoretically linear-work and polylogarithmic-depth) and show that our implementations are competitive for various input graphs. In addition, we compare our implementations to sequential connectivity algorithms and show that on 40 cores we achieve good speedup relative to the sequential implementations for many input graphs. We discuss the various optimizations used in our implementations and present an extensive experimental analysis of the performance. Our algorithm is the first parallel connectivity algorithm that is both theoretically and practically efficient.

Categories and Subject Descriptors: F.2 [Analysis of Algorithms and Problem Complexity]: General

Keywords: Parallel Algorithms, Graph Connectivity, Experiments

1. INTRODUCTION

Finding the connected components of a graph is a fundamental problem in computer science that has been well-studied. The problem takes as input an undirected graph with n vertices and m edges, and assigns each vertex a label such that vertices in the same connected component have the same label, and vertices in different connected components have different labels. Graph connectivity has many important applications, such as in VLSI design and image analysis for computer vision.

Sequentially, connectivity can be easily implemented in linear work using breadth-first search (BFS) or depth-first search, or nearly linear work with union-find. On the other hand, computing connected components and spanning forests[1] in parallel has been a long studied problem [1, 2, 15, 16, 18, 26, 28, 30, 31, 33, 34, 36, 41, 45, 47, 49, 50, 52, 53, 60]. Some of the parallel algorithms developed are relatively simple, but require super-linear work. The algorithms of Shiloach and Vishkin [53] and Awerbuch and Shiloach [2] work by combining the vertices into trees such that at the end of the algorithm vertices in the same component will belong to the same tree. These algorithms guarantee that the number of trees decreases by a constant factor in each iteration, but do not guarantee that a constant fraction of the edges are removed, and thus require $O(m \log n)$ work. The random mate algorithms of Reif [52] and Phillips [50] work by contracting vertices in the same component together and guarantee that a constant fraction of the vertices decrease in expectation per iteration, but again do not guarantee that a constant fraction of the edges are removed. Therefore, these algorithms also require $O(m \log n)$ expected work and are not work-efficient.

Work-efficient polylogarithmic-depth parallel connectivity algorithms have been designed in theory [17, 19, 23, 24, 49, 51]. These algorithms are based on random edge sampling [19, 23, 24] or linear-work minimum spanning forest algorithms, which also involve sampling and filtering edges [17, 49, 51]. However, these algorithms are complicated and unlikely to be practical (there are no implementations of these algorithms available).

There has also been significant experimental work on parallel connectivity algorithms in the past. Hambrusch and TeWinkel [25] implement connected component algorithms on the Massively Parallel Processor (MPP). Greiner [22] implements and compares parallel connectivity algorithms using NESL [9]. Goddard et al. [21], Hsu et al. [29], Bader et al. [3, 4], Patwary et al. [48], Shun et al. [57], Slota et al. [58], and the Galois system [46] implement algorithms for shared-memory CPUs. Bus and Tvrdik [12], Krishnamurthy et al. [35], Bader and JaJa [5] and Caceres et al. [13] implement connected components algorithms for distributed-memory machines. There has been some recent work on designing connectivity algorithms for GPUs [27, 59, 6]. There have also been connectivity algorithms that require time proportional to the diameter of the graph in recent graph processing packages [32, 37, 38,

[1] A spanning forest algorithm can be used to compute connected components.

SPAA'14, June 23–25, 2014, Prague, Czech Republic.

http://dx.doi.org/10.1145/2612669.2612692 .

54]. None of the previous parallel algorithms implemented are theoretically work-efficient.

We note that a parallel BFS can be performed to visit the components of the graph one-by-one. While this approach is linear-work, the depth is proportional to the sum of the diameters of the connected components. Therefore this approach is not efficient as a general-purpose parallel connectivity algorithm, although it works well for low-diameter graphs with few connected components.

In this paper, we present a simple linear-work algorithm for connectivity requiring polylogarithmic depth, and experimentally show that it rivals the best existing parallel implementations for connectivity. Our algorithm is the first work-efficient parallel graph connectivity algorithm with an implementation, and furthermore the implementation also performs well in practice.

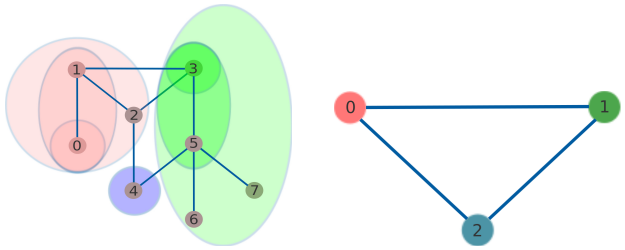
Our algorithm is based on a simple parallel algorithm for generating low-diameter decompositions of graphs by Miller et al. [44], which is an improvement of an algorithm by Blelloch et al. [10]. A low-diameter decomposition of a graph partitions the vertices, such that the diameter of each partition is small, and the number of edges between partitions is small [42]. Such decompositions have many uses in computer science, including in linear system solvers [10] and in metric embeddings [7]. The algorithm of Miller et al. partitions a graph such that the diameter of each partition is $O(\log n/\beta)$ and the number of edges between components is $O(\beta m)$ for $0 < \beta < 1$. It runs in linear work and $O(\log^2 n/\beta)$ depth with high probability[2]. Their algorithm is based on performing breadth-first searches from different starting vertices in parallel with start times drawn from an exponential distribution. Due to properties of the exponential distribution, the algorithm only needs to run the multiple breadth-first searches for at most $O(\log n/\beta)$ iterations before visiting all vertices.

We observe that this decomposition algorithm can be used to generate the connected components labeling of a graph. Our algorithm simply calls the decomposition algorithm recursively with β set to a constant fraction, and after each call contracts each partition into a single vertex, and relabels the vertices and edges between partitions. Since the number of edges decreases by a constant fraction in expectation in each recursive call, the algorithm terminates after $O(\log n)$ calls with high probability. Hence we obtain an algorithm for connected components labeling that runs in linear work and $O(\log^3 n)$ depth with high probability. An illustration of this algorithm is shown in Figure 1. Our implementation is based on parallel breadth-first searches and some simple parallel routines.

We also present a slight modification of the decomposition algorithm of Miller et al. which relaxes the relative ordering among vertices due to different breadth-first search start times. We show that this modification does not affect the asymptotic complexity of the decomposition algorithm, while leading to a simpler and faster implementation. We use this decomposition algorithm for connectivity and apply various optimizations to our implementations.

We experimentally compare our algorithm against the fastest existing parallel connectivity implementations (which are not theoretically linear-work and polylogarithmic-depth) [57, 48, 54, 58] on a variety of input graphs and show that our algorithm is competitive. On 40 cores, our parallel implementations achieve 18–39 times speedup over the same implementation run on a single thread, and achieve good speedups over the sequential implementations on many graphs. We show that on most graphs, the number of edges decreases by significantly more than predicted by the theoretical bounds due to duplicate edges between components. In addition, we study how the performance of our algorithms varies with different settings of β in the decomposition algorithms.

[2]We use "with high probability" (w.h.p.) to mean probability at least $1 - 1/n^c$ for any constant $c > 0$.



(a) graph decomposition (b) contracted graph

Figure 1. Illustration of our decomposition-based connectivity algorithm. (a) At $t = 0$, vertex 0 starts a BFS (red ball), and at $t = 1$, vertices 3 (green ball) and 4 (blue ball) start BFS's. In this illustration, when there are ties (multiple BFS's visiting the same unvisited neighbor), the BFS center with the lowest ID wins. The balls represent the resulting partitions and the rings around the balls represent each level of the corresponding BFS. (b) Each ball is contracted into a single vertex, and the decomposition is applied recursively.

Contributions. The main contributions of this paper are as follows. Firstly, we describe a simple linear-work and polylogarithmic-depth parallel algorithm for connectivity. This is the first practical parallel connectivity algorithm with a linear-work guarantee. Secondly, we describe a (modest) variation of the parallel decomposition algorithm by Miller et al. that leads to a faster implementation and prove that it has the same theoretical guarantees as the original algorithm. Next, we present highly-optimized implementations of our algorithm. Finally, we present experimental results showing that our algorithm is competitive with the best previously available parallel implementations of graph connectivity.

2. NOTATION AND PRELIMINARIES

In this paper, we use the concurrent-read concurrent-write (CRCW) parallel random access machine model (PRAM). We state our results in the work-depth model, where ***work*** is equal to the number of operations required (equivalently, the product of the time and the number of processors) and ***depth*** is equal to the number of time steps required.

We use the atomic compare-and-swap and writeMin operations in our implementations. A ***compare-and-swap*** (CAS) is an atomic instruction that takes three arguments—a memory location (*loc*), an old value (*oldV*) and a new value (*newV*); if the value stored at *loc* is equal to *oldV* it atomically stores *newV* at *loc* and returns *true*, and otherwise it does not modify *loc* and returns *false*. A ***writeMin*** is an atomic instruction that takes three arguments—a memory location (*loc*), a value (*val*), and a comparison function $<$, and atomically updates the value at *loc* to be the minimum of the stored value and *val* according to $<$. It returns *true* if the value at *loc* was changed, and *false* otherwise. writeMin can be implemented with a loop, which reads the value at *loc* and applies a CAS if *val* is less than the value read according to $<$. The loop terminates when either a CAS is successful or when the read value is smaller than *val* according to $<$. The reader may refer to [56] for details.

A ***connected component*** in an undirected graph contains vertices such that any two vertices can reach one another through a path. The ***connected components labeling*** problem takes an undirected graph $G = (V, E)$, and returns a labeling L such that for two vertices u and v, $L(u) = L(v)$ if u and v belong in the same connected component, and $L(u) \neq L(v)$ otherwise.

A ***breadth-first search (BFS)*** algorithm takes an unweighted graph $G = (V, E)$ and a source vertex $r \in V$, and visits the vertices reachable from r in breadth-first order, i.e. for all reachable vertices $u, v \in V$, if $\text{dist}(r, u) < \text{dist}(r, v)$ then u will be visited before v, where $\text{dist}(x, y)$ is the length of the shortest path between x and y. A simple parallel algorithm processes each level of the BFS in parallel [11].

The ***exponential distribution*** with parameter λ is defined by the probability density function:

$$f(x, \lambda) = \begin{cases} \lambda e^{-\lambda x} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

The mean of the exponential distribution is $1/\lambda$.

A (β, d)***-decomposition*** ($0 < \beta < 1$) of an undirected graph $G = (V, E)$ is a partition of V into subsets $V_1, \ldots, V_k$ such that (1) the shortest path between any two vertices in each V_i using only vertices in V_i is at most d, and (2) the number of edges $(u, v) \in E$ such that $u \in V_i$, $v \in V_j$, $i \neq j$ is at most βm.

Miller et al. present a parallel decomposition algorithm based on parallel BFS's [44], which we call DECOMP. They prove that for a value β, DECOMP generates a $(\beta, O(\frac{\log n}{\beta}))$ decomposition in $O(m)$ work and $O(\frac{\log^2 n}{\beta})$ depth with high probability on a CRCW PRAM. The algorithm works by assigning each vertex v a *shift value* δ_v drawn from an exponential distribution with parameter β (mean $1/\beta$). Miller et al. show that the maximum shift value is $O(\frac{\log n}{\beta})$ w.h.p. Each vertex v is then assigned to the partition S_u that minimizes the *shifted distance* $\text{dist}_{-\delta}(u, v) = \text{dist}(u, v) - \delta_u$. This can be implemented by performing multiple BFS's in parallel. Each iteration of the implementation explores one level of each BFS and at iteration t (starting with $t = 0$) breadth-first searches are started from the unvisited vertices v such that $\delta_v \in [t, t + 1)$. If multiple BFS's reach the same unvisited vertex w in the same time step, then w is assigned to the partition corresponding to the origin of the BFS with the smaller fractional portion of the shift value (equivalently, w is assigned to the partition whose origin has the smallest shifted distance to w). Since the maximum shift value is $O(\frac{\log n}{\beta})$, the algorithm terminates in $O(\frac{\log n}{\beta})$ iterations. Each iteration requires $O(\log n)$ depth for packing the frontiers of the BFS's, leading to an overall depth of $O(\frac{\log^2 n}{\beta})$ w.h.p. The BFS's are work-efficient, so the total work is $O(m)$.

3. LINEAR-WORK CONNECTIVITY

In this section, we describe our simple linear-work parallel algorithm for connectivity. As a subroutine, it uses the parallel decomposition algorithm DECOMP described in Section 2. By the definition of a decomposition, the number of inter-component edges remaining after a call to DECOMP starting with m edges is at most βm in expectation. We can contract each component into a single vertex and recurse on the remaining graph, whose edge count has decreased by at least a constant factor in expectation. This leads to a linear-work parallel connectivity algorithm, assuming the contraction and relabeling can be done efficiently.

The pseudo-code for our connected components algorithm (CC) is shown in Algorithm 1. The input to DECOMP is a graph $G(V, E)$ and a value β, and the output is a labeling L of the vertices in V, such that vertices in the same partition will have the same label. CONTRACT takes a graph $G(V, E)$ and a labeling L as input, and returns a new graph $G'(V', E')$ such that vertices with the same label in V according to L are contracted into a single vertex, forming the vertex set V', and the inter-component edges in E are relabeled according to L and form the edge set E'. RELABELUP takes as input labelings L and L' and returns a new labeling L'' such that $L''[i] = L'[L[i]]$. RELABELUP is necessary because the original labels L must be updated with the labels L' returned by the recursive call to CC.

Algorithm 1 Parallel decomposition-based algorithm for connected components labeling

```
1:  β = some constant fraction in (0, 1)
2:  procedure CC(G(V, E))
3:      L = DECOMP(G(V, E), β)
4:                          ▷ L contains the labels returned by DECOMP
5:      G'(V', E') = CONTRACT(G(V, E), L)
6:      if |E'| = 0 then
7:          return L
8:      else
9:          L' = CC(G'(V', E'))
10:         L'' = RELABELUP(L, L')
11:         return L''
```

THEOREM 1. *Algorithm 1 runs in $O(m)$ expected work and $O(\log^3 n)$ depth w.h.p. on a CRCW PRAM.*

PROOF. The algorithm sets β to a constant between 0 and 1. Since the number of edges decreases to at most βm in expectation after each recursive call, and the rate of reduction is independent across iterations, the total number of calls is $O(\log_{\frac{1}{\beta}} m)$ w.h.p. Each recursive call requires $O(\frac{\log^2 n}{\beta})$ depth w.h.p. and $O(m')$ work where m' is the number of remaining edges for DECOMP. Hence the total contribution of DECOMP to the depth of CC is $O(\log_{\frac{1}{\beta}} m \frac{\log^2 n}{\beta}) = O(\log^3 n)$ w.h.p. and the total contribution to the work of CC is upper bounded by $\sum_{i=0}^{\infty} \beta^i cm$ for some constant c, which is $O(m)$ in expectation.

We now discuss an implementation of DECOMP that allows us to do contraction and relabeling within the same complexity bounds. Recall that DECOMP performs multiple breadth-first searches in parallel, with each BFS corresponding to one of the components (partitions) of the graph. We can maintain all BFS's using a single frontier array, where vertices belonging to the same component are in consecutive positions in the frontier. In each iteration, vertices that need to start their own BFS are added to the end of this frontier array in parallel. We store all of the frontiers created throughout one call to DECOMP, and there are $O(\frac{\log n}{\beta})$ such frontiers w.h.p. Each individual BFS stores the starting and ending position of its component's vertices on each frontier, as well as the total number of edges for these vertices. Using this information, we can compute appropriate offsets into shared arrays for each component using prefix sums over all the $O(\frac{\log n}{\beta})$ frontiers for each BFS. For each iteration of CC, the work for computing offsets is $O(m')$ where m' is the number of edges at the beginning of the iteration, and the depth is $O(\frac{\log n'}{\beta})$.

As a vertex visits other vertices during the BFS's, if it encounters an edge to a vertex belonging to the same component (an intra-component edge), it will mark that edge as deleted (using some special value). These edges will be packed out at the end of DECOMP, which can be done in $O(m')$ total work and $O(\log m')$ depth, where m' is the number of edges at the beginning of the iteration. The rest of the edges will be inter-component edges and hence need to be kept for the next iteration. Each component will become a single vertex in the next iteration, with all of the edges of the component vertices merged. Since the vertices of each component are stored consecutively on the frontiers, we can create a new edge array and have the original vertices copy their edges in,

guaranteeing that the resulting array will store each component's edges consecutively (we can compute each vertex's offset into this array with a prefix sum). We can remove duplicate edges within the complexity bounds of an iteration using parallel hashing [43, 20], although the number of edges decreases by a constant factor in expectation even if we do not remove duplicates.

To relabel the new vertices, we first compute the total number of components k and assign each original label with a new label in the range $[0, \ldots, k-1]$, which can be done using prefix sums. Singleton vertices are then removed, but their labels are kept. For the k' non-singleton vertices remaining, we relabel them to the range $[0, \ldots, k'-1]$ and recursively call CC. After the recursive call, the original labels are relabeled according to the result of CC. This can all be done using prefix sums in linear work in the number of remaining vertices and $O(\log n)$ depth per iteration.

We summarize the proof of this theorem. For a constant fraction β, there are $O(\log n)$ calls to DECOMP w.h.p., each of which does $O(\log n)$ iterations of BFS. Each iteration of BFS requires $O(\log n)$ depth for packing. The depth for contraction and relabeling is absorbed by the depth of DECOMP. This gives an overall depth of $O(\log^3 n)$ w.h.p. DECOMP, contraction and relabeling can be done work-efficiently, and each call to DECOMP decreases the number of edges by a constant fraction in expectation, leading to $O(m)$ expected work overall. □

We note that theoretically the depth of DECOMP could be improved to $O(\log n \log^* n)$ by using approximate compaction [20] (which is linear-work) for packing the frontiers of the BFS's. This would give us an algorithm with expected linear-work algorithm and $O(\log^2 n \log^* n)$ depth w.h.p.

We consider a slight variation of DECOMP which breaks ties arbitrarily among frontier vertices visiting the same unvisited neighbor in a given iteration of the BFS's. This modification simplifies our implementation and leads to improved performance as we discuss later in the paper. This variation is equivalent to rounding down all the δ_v values to the nearest integer and again assigning each vertex v to the partition S_u that minimizes $\text{dist}_{-\delta}(u, v) = \text{dist}(u, v) - \delta_u$, but breaking ties arbitrarily. We call this version ***Decomp-Arb*** and show that this modified version has the same theoretical guarantees (within a constant factor). In particular, we show that the number of inter-component edges in the decomposition is at most $2\beta m$ in expectation (the original bound was βm).

THEOREM 2. *Decomp-Arb generates a* $O(2\beta, O(\frac{\log n}{\beta}))$ *decomposition in* $O(m)$ *expected work and* $O(\frac{\log^2 n}{\beta})$ *depth w.h.p.*

PROOF. Since we are still picking values from an exponential distribution, the diameter of each component is $O(\frac{\log n}{\beta})$ w.h.p. as shown in [44]. Hence the depth of the algorithm is the same as the original algorithm, namely $O(\frac{\log^2 n}{\beta})$ w.h.p. The work is still $O(m)$ in expectation, since the BFS's are work-efficient. Hence we only need to show that the number of inter-component edges is at most $2\beta m$ in expectation.

As in [44], consider the midpoint w of an edge (u, v). Lemma 4.3 of [44] states that if u and v belong to different components, then $dist_{-\delta}(u', w)$ and $dist_{-\delta}(v', w)$ are within 1 of the minimum shifted distance to w. Decomp-Arb rounds all shifted distances down to the nearest integer. Hence when comparing two rounded shift distances, their difference is at most 1 if and only if the two original shift distances were within 2 of each other. In other words, suppose the two distances we are comparing are d_1 and d_2. Then $||\lfloor d_2 \rfloor - \lfloor d_1 \rfloor|| \leq 1$ if and only if $|d_2 - d_1| < 2$. Hence we can modify Lemma 4.3 of [44] to state that if u and v belong to different components, then $dist_{-\delta}(u', w)$ and $dist_{-\delta}(v', w)$ (using the original shift distances) are within 2 of the minimum shifted distance to w.

Lemma 4.4 of [44] uses properties of the exponential distribution to show that the probability that the smallest and second smallest shifted distance to w (corresponding to the first two BFS's that arrive at w) has a difference of less than c is at most βc. Here we have $c = 2$, so the probability that an edge is an inter-component edge is at most 2β. By linearity of expectations, the expected total number of inter-component edges is at most $2\beta m$. □

We can plug in Decomp-Arb into the proof of Theorem 1 and obtain a linear-work connectivity algorithm for $0 < \beta < 1/2$.

4. IMPLEMENTATION DETAILS

A naive implementation of Algorithm 1 would probably only require tens of lines of code. However to obtain the best performance in practice, an implementation must take into account constant factors, cache performance, and the synchronization primitives used. Therefore, in this section we describe our algorithmic engineering efforts to obtain a fast implementation of Algorithm 1. We describe three versions of DECOMP, referring to the original algorithm as Decomp-Min, the version which breaks ties arbitrarily as Decomp-Arb, and a variant of Decomp-Arb that we discuss later as Decomp-Arb-Hybrid.

We represent our graph using the adjacency array format, where we have an array of vertex offsets V into an array of edges E. The targets of the outgoing edges of vertex i are then stored in $E[V[i]], \ldots, E[V[i+1]] - 1$ (to deal with the edge case, we set $V[n] = m$). Our graph is undirected so each edge is stored in both directions. We also maintain an array D, where $D[i]$ stores the degree of the i'th vertex. Initially $D[i]$ is set to $V[i+1] - V[i]$.

As suggested in [44], in our implementations we simulate the assignment of values from the exponential distribution to vertices by generating a random permutation (in parallel), and in each round adding chunks of vertices starting from the beginning of the permutation as start centers for new BFS's, where the chunk size grows exponentially. If a vertex in a chunk has already been visited, then it is not added as a start center. Each vertex also draws a random integer from a large enough range to simulate the fractional part of its shift value (denoted by δ'_v for vertex v), used to break ties if multiple BFS's visit the same unvisited neighbor. We maintain the active frontier of the BFS's using a single array. New BFS centers are simply added to the end of this array in parallel. We note that parallel BFS can also be implemented using Cilk reducers [40] with similar performance.

Since we do not need to keep around the inter-component edges in recursive calls to CC, we pack out inter-component edges as we encounter them. Therefore as we explore vertices, we determine on-the-fly whether the incident edge to the explored vertex is an inter-component edge or an intra-component edge.

In contrast to the description in the proof of Theorem 1, in our implementations we do not store the frontiers of the BFS's and offsets of each BFS into the frontiers. Therefore the vertices of the same component will not be able to be accessed contiguously in memory. Instead, in the contraction phase we use an integer sort to collect all the vertices of the same component together. We found this to be more efficient than the method described in the proof of Theorem 1 because the amount of bookkeeping is reduced and the integer sort is only performed over the remaining inter-component edges, which is usually much fewer than the number of original

edges. We use the linear-work and $O(m^\epsilon)$ depth ($0 < \epsilon < 1$) integer sort algorithm from the Problem Based Benchmark Suite [57].

Decomp-Min is split into two phases over the frontier vertices (pseudo-code shown in Algorithm 2). In our implementation, we use an array C to to store both the component ID's of the vertices and to store the values that vertices write to resolve conflicts. In particular, the array C stores pairs (c_1, c_2) where for a vertex v, c_1 is used for markings from frontier vertices competing to visit v, and c_2 stores the component ID of vertex v. We will use $C_1[v]$ and $C_2[v]$ to refer to the first and second value of the pair $C[v]$, respectively. Decomp-Min uses the writeMin operation (described in Section 2) on integer pairs, where the comparison function (not shown in the pseudo-code) uses integer comparison on the first value of pair. Note that instead of keeping pairs in C we could keep two arrays, one to store the component IDs and the other to resolve conflicts, but this leads to an additional cache miss per vertex visit.

The entries of C are initialized to (∞, ∞) on Line 1. The ∞ in the second value of the pair indicates that the vertex has not yet been visited, and the first value of the pair is the identity value for the writeMin function. When a vertex v is added to the BFS on Lines 5–6 (i.e. it starts a new BFS), $C[v]$ is set to $(-1, v)$—the value -1 in $C_1[v]$ indicates that v has been visited, and the value v in $C_2[v]$ indicates that the component ID of v is its own vertex ID. In our implementation, inter-component edges are kept while intra-component edges are deleted on-the-fly. We overwrite the edge array E as we loop over the edges (Lines 17–18 and 21–22) using a counter k indicating the current position in the array (Line 11). In the first phase, frontier vertices mark unvisited neighbors with the writeMin primitive (Lines 14–16) with the fractional part of its BFS center's shift value, $\delta'_{C_2[v]}$ (the BFS center's ID is equal to $C_2[v]$, the component ID of v). We assume there are no ties as the numbers can be drawn from a large enough range to guarantee this w.h.p. Also, as long as for a neighbor w, $C_1[w] \neq -1$, this means the neighbor has not been visited in a *previous iteration*. In this case, we need to keep the edge (Lines 17–18) as we currently do not know whether it is an intra- or inter-component edge (this can only be determined once all other frontier vertices finish doing their writeMin's). Otherwise, the neighbor w has been visited in a previous iteration and we can determine the status of the edge to w—if w has component label different from v, it keeps the edge as it is an inter-component edge (Lines 20–22). It labels the endpoint of the edge with its new component ID (so that it does not have to be relabeled later) but sets the sign bit of the value (negates it and subtracts 1) to indicate that this edge need not be considered again in the second phase. Otherwise, the edge is an intra-component edge and is deleted. We set the degree of v to be the number of edges kept in this phase (Line 23).

In the second phase, the remaining edges incident on v are looped over and for edges which have a non-negative value (an edge whose status has not yet been determined from the first phase), we determine whether $\delta'_{C_2[v]}$ is stored on the neighbor w. If so, then v uses a compare-and-swap (CAS) to attempt to atomically set $C_1[w]$ to -1 (so that future writeMin's will not mark it again) and if successful adds w to the next frontier (Lines 30–31) and does not keep the edge (it is an intra-component edge). A CAS is required here since there could be multiple vertices from the same component exploring the same neighbor w (they all have the same $\delta'_{C_2[v]}$ value), and we want w to be added only once to the next frontier. If the condition on Line 30 does not hold, we check whether the component ID of w matches that of v, and if they differ, then the edge is an inter-component edge and we keep it (Lines 32–35). We set the sign bit of the value of its component ID and store it in E (Lines 34–35). If $C_2[w] = C_2[v]$, then (v, w) is an intra-component edge and we do not keep it. If the edge has a negative value, then it was already processed in the first phase, and we just keep it (Lines 36–38). We set the degree of v to be the number of inter-component edges incident on v (Line 39). After the BFS's are finished, we unset the sign bit of the remaining (inter-component) edges, so that they can be properly processed during the relabeling phase after the call to DECOMP by the connected components algorithm.

Note that for high-degree vertices (e.g. degree greater than $k \log n$ for some constant k), the inner sequential for-loops over the neighbors of a vertex can be replaced with a parallel for-loop, marking the deleted edges with a special value and packing the edges with a parallel prefix sums after the for-loop.

Algorithm 2 Decomp-Min

```
1:  C = {(∞,∞),...,(∞,∞)}
2:  Frontier = {}
3:  numVisited = 0
4:  while (numVisited < n) do
5:      add to Frontier unvisited vertices v with δ_v < round + 1
6:          and set C[v] = (−1, v)                          ▷ new BFS centers
7:      numVisited = numVisited + size(Frontier)
8:      NextFrontier = {}
9:      parfor v ∈ Frontier do
10:         start = V[v]                                    ▷ start index of edges in E
11:         k = 0
12:         for i = 0 to D[v] − 1 do
13:             w = E[start + i]
14:             if C_1[w] ≠ −1 then
15:                 if C_1[w] > δ'_{C_2[v]} then
16:                     writeMin(C[w],(δ'_{C_2[v]}, C_2[v]))
17:                 E[start + k] = w
18:                 k = k + 1
19:             else
20:                 if C_2[w] ≠ C_2[v] then
21:                     E[start + k] = −C_2[w] − 1
22:                     k = k + 1
23:         D[v] = k
24:     parfor v ∈ Frontier do
25:         start = V[v]                                    ▷ start index of edges in E
26:         k = 0
27:         for i = 0 to D[v] − 1 do
28:             w = E[start + i]
29:             if w ≥ 0 then
30:                 if C_1[w] = δ'_{C_2[v]} and CAS(C_1[w], δ'_{C_2[v]}, −1) then
31:                     add w to NextFrontier               ▷ v won on w
32:                 else
33:                     if C_2[w] ≠ C_2[v] then
34:                         E[start + k] = −C_2[w] − 1
35:                         k = k + 1
36:             else
37:                 E[start + k] = w
38:                 k = k + 1
39:         D[v] = k
40:     NextFrontier = Frontier
```

Decomp-Min is split into two phases because we need all the vertices to apply the writeMin on their unvisited neighbors before we can determine a winner. Hence, a synchronization point is needed between the writeMin's and the checks to see if a vertex successfully visits a neighbor.

In contrast to Decomp-Min, Decomp-Arb only requires one phase over the edges of the frontier vertices and their outgoing edges (pseudo-code shown in Algorithm 3). Here C stores only a single integer value, indicating the component ID's of the vertices. Each entry is initialized to ∞ (Line 1) to indicate that the vertex has not yet been visited. The code of Decomp-Arb is similar to that of Decomp-Min, except that there is only a single phase over the edges of each frontier. Instead of using a writeMin as in Decomp-

Min, Decomp-Arb uses a CAS to mark an unvisited neighbor (Line 14) with the component ID of the frontier vertex. A vertex that successfully marks a neighbor can delete its edge to that neighbor since it is guaranteed to be an intra-component edge. That vertex is also responsible for adding the neighbor to the next frontier (Line 15). Otherwise, the vertex checks the component ID of its neighbor and if it differs from its own, it keeps the edge as an inter-component edge (Lines 17–19). It also marks the endpoint of the edge with its component ID so that it doesn't have to be relabeled later (Line 18). Note that although the pseudo-code shown does not make use of the fact that the degree is set to the number of inter-component edges on Line 20, we make use of it during the relabeling phase (not shown in the pseudo-code). Unlike in Decomp-Min, Decomp-Arb does not need to use the fractional part of the shift values (the δ'_v values) because an arbitrary BFS can mark an unvisited neighbor.

Decomp-Arb only requires a single phase over the edges of the frontier vertices because once a vertex w is visited by some vertex v and its component ID is set to the component ID of v, it can no longer be visited again by another vertex. At that point we know that the edge from v to w is an intra-component edge and can delete it, and any other neighbor of w with a different component ID than w that fails to mark w with the CAS has an inter-component edge to w which is kept.

Algorithm 3 Decomp-Arb

```
1:  C = {∞, ..., ∞}
2:  Frontier = {}
3:  numVisited = 0
4:  while (numVisited < n) do
5:      add to Frontier unvisited vertices v with δ_v < round + 1
6:        and set C[v] = v                          ▷ new BFS centers
7:      numVisited = numVisited + size(Frontier)
8:      NextFrontier = {}
9:      parfor v ∈ Frontier do
10:         start = V[v]                            ▷ start index of edges in E
11:         k = 0
12:         for i = 0 to D[v] − 1 do
13:             w = E[start + i]
14:             if C[w] = ∞ and CAS(C[w], ∞, C[v]) then
15:                 add w to NextFrontier
16:             else
17:                 if C[w] ≠ C[v] then             ▷ inter-component edge
18:                     E[start + k] = C[w]
19:                     k = k + 1
20:         D[v] = k
21:     NextFrontier = Frontier
```

During the relabeling phase, we only need to relabel the source endpoint of each remaining edge, as the target endpoint was already relabeled during DECOMP. After relabeling, we use a parallel hash table [55] to remove duplicate edges between components. On the way back up from the recursive call to CC, we simply index into the labeling returned by CC with a parallel for-loop to relabel the original labels appropriately (corresponding to RELABELUP of Algorithm 1).

As we show experimentally in Section 5, Decomp-Arb performs better than Decomp-Min due to only requiring one pass over the edges of each frontier during the BFS's, and needing less bookkeeping overall.

We considered the *direction-optimizing* (hybrid) BFS idea first described by Beamer et al. [8] and later implemented for general graph traversal algorithms in Ligra [54]. In BFS, the idea is that when the frontier is large, it is cheaper to have all unvisited vertices read their incoming neighbors and once a vertex finds a neighbor on the frontier, it chooses it as its parent and quits (subsequent incoming edges to this vertex do not need to be examined). If a large number of vertices' neighbors are on the frontier, then this possibly saves many edge traversals.

In contrast to a standard BFS, our connectivity algorithm requires all edges to be inspected, since we must decide whether the edge is an inter-component or an intra-component edge for the recursive call. Therefore, if we apply the optimization, we must introduce a post-processing step that inspects the edges determining whether or not they should be kept, so the total number of edges inspected is not reduced. We apply this optimization to Decomp-Arb, as it allows a vertex to select an arbitrary neighbor's component ID, and thus can exit the loop over the neighbors early. One modification is that edges that are relabeled on-the-fly during the write-based computation (e.g. Line 19 of Algorithm 3) must be marked that they have been relabeled, so that we do not process them again during the post-processing phase (we use the sign bit in the label for this purpose). Our experiments show that even though no edge traversals are saved, switching to the read-based computation when the frontier is large (the fraction of vertices on the frontier is greater than 20%) helps for some graphs, as the read-based computation is more cache-friendly, and does not require using an atomic operation, in contrast to the original Decomp-Arb which uses compare-and-swaps to resolve conflicts. We refer to the direction-optimizing version of Decomp-Arb as ***Decomp-Arb-Hybrid***.

5. EXPERIMENTS

We compare our three implementations of the connectivity algorithm to the fastest available parallel connectivity algorithms that we are aware of [57, 48, 54, 58]. We refer to our algorithm using Decomp-Min as ***decomp-min-CC***, Decomp-Arb as ***decomp-arb-CC*** and Decomp-Arb-Hybrid as ***decomp-arb-hybrid-CC***. We also tried parallelizing over the edges for the high-degree vertices in our implementations (as discussed in Section 4), but due to the modest core count of our machine, we did not find a performance improvement. Patwary et al. [48] describe two parallel spanning forest implementations—a lock-based one and a verification-based one. We use their lock-based implementation (***parallel-SF-PRM***) since we found that the verification-based one sometimes fails to terminate. Furthermore, they found that their lock-based implementation usually outperforms their verification-based one. We also compare with the parallel spanning forest implementation in the Problem Based Benchmark Suite (PBBS) [57] (***parallel-SF-PBBS***). We note that these existing spanning forest-based parallel implementations are not theoretically work-efficient. As for connectivity based on BFS, we compare with the direction-optimizing BFS [8] available as part of Ligra [54], performed on each component of the graph. We refer to this implementation as ***hybrid-BFS-CC***. This approach is work-efficient but the depth can be linear in the worst case. Very recently and independently of our work, Slota et al. [58] describe a connected components algorithm which combines direction-optimizing BFS with label propagation (***multistep-CC***). In label propagation, each vertex starts with a unique ID and in each iteration every vertex updates its ID to be the minimum of its own ID and all of its neighbors IDs; the label propagation terminates when no IDs change in an iteration. In the worst case, the algorithm of Slota et al. requires quadratic work and linear depth. We compare all of the parallel implementations to a simple sequential spanning forest-based connectivity algorithm using union-find (***serial-SF***) from the PBBS. The single-thread times for hybrid-BFS-CC and multistep-CC are sometimes better than serial-SF, and can also be used as a sequential baseline. For the spanning forest-based connectivity algorithms, we include in the timings a post-processing step that finds the ID of the root of the tree for each vertex (done in parallel for the parallel implementations).

We run our experiments on a 40-core (with hyper-threading) machine with 4 × 2.4GHz Intel 10-core E7-8870 Xeon processors (with a 1066MHz bus and 30MB L3 cache) and 256GB of main memory. We run all parallel experiments with two-way hyper-threading enabled, for a total of 80 hyper-threads. We compiled our code with `g++` version 4.8.0 with the `-O2` flag. The parallel codes use Cilk Plus [39] to express parallelism, which is supported by the `g++` compiler that we use. In particular, the parallel for-loops are written using the `cilk_for` construct. Divide-and-conquer parallelism, which is required by the parallel integer sort, is written using the `cilk_spawn` construct. When running in parallel, we use the command `numactl -i all` to evenly distribute the allocated memory among the processors.

We use a variety of synthetic graphs, the first three of which are taken from the PBBS [57], and a real-world graph. ***random*** is a random graph where every vertex has five edges to neighbors chosen randomly. The ***rMat*** graph [14] is a graph with a power-law degree distribution. ***rMat2*** uses the same generator as rMat, but with a higher edge-to-vertex ratio, giving a denser graph. ***3D-grid*** is a grid graph in 3-dimensional space where every vertex has six edges, each connecting it to its 2 neighbors in each dimension. ***line*** is a path of length $n - 1$ (i.e. each vertex has two neighbors except for the first and the last vertex in the path). This is a degenerate graph with diameter $n - 1$. ***com-Orkut*** is a social network graph downloaded from the Stanford Network Analysis Project (SNAP), available at `http://snap.stanford.edu`. For the synthetic graphs, the vertex labels are randomly assigned. The sizes of the graphs are shown in Table 1. For our decomposition-based algorithms we store an edge in each direction, so we use twice the number of edges than as noted in Table 1, while for the spanning forest-based algorithms, edges only need to be stored in one direction.

Input Graph	Num. Vertices	Num. Edges
random	10^8	5×10^8
rMat	2^{27}	5×10^8
rMat2	2^{20}	4.2×10^8
3D-grid	10^8	3×10^8
line	5×10^8	5×10^8
com-Orkut	3,072,627	117,185,083

Table 1. Input graphs

The serial and parallel running times of the implementations on the various inputs are summarized in Table 2. The times that we report are based on a median of three trials. We see that decomp-arb-CC and decomp-arb-hybrid-CC usually outperform decomp-min-CC (by up to 2.3 times). This is because (1) decomp-arb-CC and decomp-arb-hybrid-CC require only one pass over the edges of the frontier instead of two passes in decomp-min-CC and (2) the vertices store less data when computing the labeling. Decomp-arb-hybrid-CC is faster than decomp-arb-CC for most of the graphs, especially for the graphs whose frontier grows very large (e.g. about 2x faster for rMat2 and com-Orkut), as these graphs benefit more from the optimization of using a read-based computation for the large frontiers. For 3D-grid and line, the times are about the same for decomp-arb-CC and decomp-arb-hybrid-CC, since in decomp-arb-hybrid-CC the frontier never grows large enough to switch to the read-based computation. Among the two spanning forest-based parallel implementations, parallel-SF-PRM is faster than parallel-SF-PBBS in parallel. Compared to parallel-SF-PRM, decomp-arb-hybrid-CC is at most 70% slower in parallel, and faster sequentially. On 40 cores with hyper-threading, our parallel implementations achieve a self-relative speedup of between 18 and 39.

We observe that the implementations based on a single direction-optimizing BFS (hybrid-BFS-CC and multistep-CC) work well for dense graphs with low-diameter, such as random, rMat2 and com-Orkut, outperforming the other implementations both sequentially and in parallel on these graphs. For the dense rMat2 graph, which requires only 5 levels of BFS to completely traverse, even the sequential times of these implementations are competitive with the parallel times of the other implementations. This is because the read-based optimization of direction-optimizing BFS significantly reduces the number of edges traversed. For graphs with many components (i.e. rMat with over 13 million components), hybrid-BFS-CC does poorly in parallel since it visits the components one-by-one, while multistep-CC does better because it uses parallel BFS to compute only one component, and then switches to label propagation to compute the rest. For the line graph, both implementations perform poorly and get no speedup due to the large diameter of the graph. Our fastest parallel implementation (decomp-arb-hybrid-CC) is faster than hybrid-BFS-CC and multistep-CC for the line graph, competitive for the rMat and 3D-grid graphs, and slower for the random, rMat2 and com-Orkut graphs. For graphs with only one component (random, rMat2, 3D-grid and line), multistep-CC and hybrid-BFS-CC both perform exactly one BFS, and the differences in running times are due to the choice of when to switch to the read-based computation, starting vertex of the BFS, and slight implementation differences. Note that on a single thread, multistep-CC outperforms serial-SF for four of the graphs, since the read-based optimization allows it to traverse many fewer edges for these graphs.

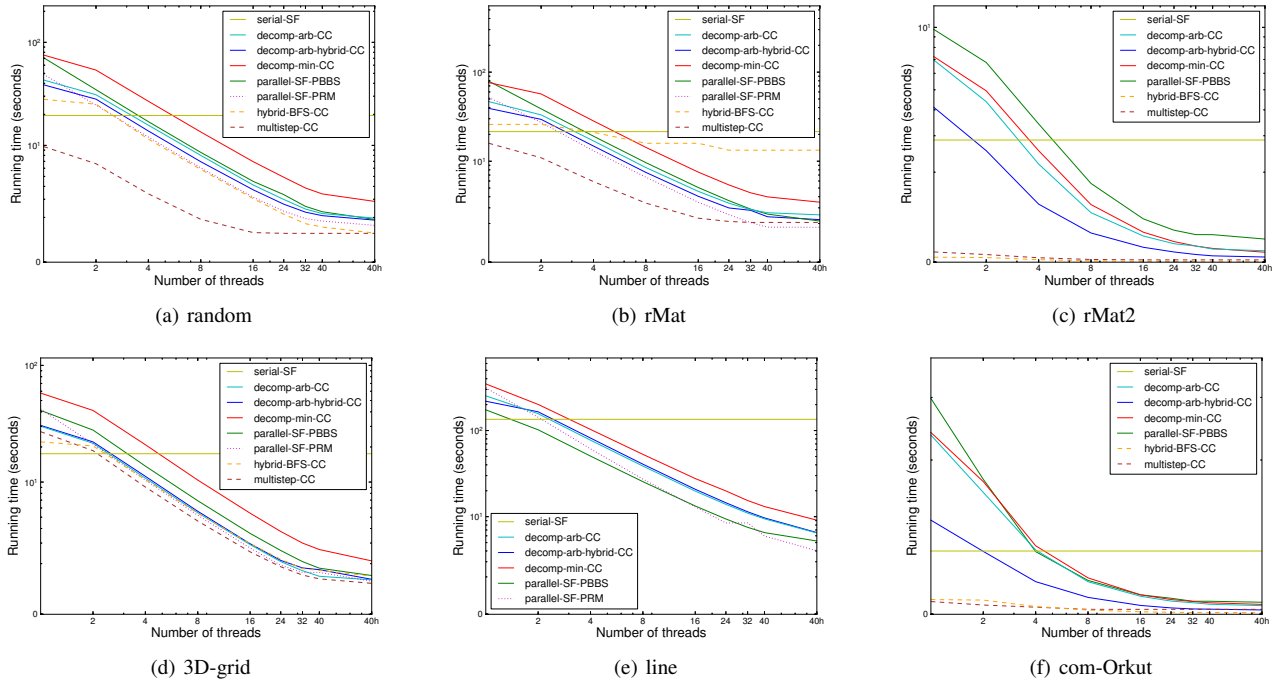
Compared to the best single-thread times among serial-SF, hybrid-BFS-CC and multistep-CC, on 40 cores our fastest implementation achieves up to a 13 times speedup. For the dense rMat2 graph, on 40 cores our parallel implementation is actually slower than hybrid-BFS-CC run on a single thread, but this is a special case on which the direction-optimizing BFS approach works particularly well.

Figure 2 shows the running time versus the number of threads for the different implementations on the input graphs. For the line graph, we do not plot hybrid-BFS-CC and multistep-CC as they perform very poorly and get no speedup. We see that our parallel implementations get good speedup, and except for rMat2 and com-Orkut, outperform the best sequential time with a modest number of threads. Our parallel implementations (decomp-arb-CC, decomp-arb-hybrid-CC and decomp-min-CC) perform reasonably well and are competitive with the other parallel implementations implementations, which are not theoretically linear-work and polylogarithmic-depth guarantee, for all graphs except rMat2 and com-Orkut, on which the direction-optimizing BFS implementations perform exceptionally well. While our parallel implementations do not achieve the fastest performance for any particular graph, due to their theoretical guarantees, they perform reasonable well across all inputs and do not suffer from poor performance on any "worst-case" inputs.

Figure 3 shows the 40-core running time of decomp-arb-CC, decomp-arb-hybrid-CC and decomp-min-CC as a function of the parameter β for several graphs. We see that the trends for the implementations are similar, and the β leading to the fastest running times is between 0.05 and 0.2. Figure 4 shows the number of edges remaining per iteration for decomp-arb-hybrid-CC as a function of β. As expected, the number of edges drops more quickly for smaller β, leading to fewer phases until reaching the base case. Furthermore, the upper bound of a 2β-fraction of edges being removed (or β-fraction for decomp-min-CC) per iteration does not account for the removal of duplicate edges between contracted components. For all our graphs except the line graph, there are (many) duplicate edges between components that are removed, leading to a much sharper decrease (up to an order of magnitude more than predicted by the upper bound) in the number of remaining edges per iteration.

Implementation	random		rMat		rMat2		3D-grid		line		com-Orkut	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serial-SF	19.5	–	21.5	–	2.86*	–	17.5	–	68.6	–	0.82*	–
decomp-arb-CC	43.1	1.97	46.7	2.5	6.95	0.256	30.1	1.36	254	6.49	2.35	0.115
decomp-arb-hybrid-CC	38.7	1.89	39.8	2.22	4.11	0.116	30.6	1.39	247	6.5	1.22	0.058
decomp-min-CC	74.8	2.86	76.3	3.49	7.22	0.221	57.9	2.11	348	9.11	2.39	0.132
parallel-SF-PBBS	70.9	1.91	79.2	2.13	9.79	0.515	41.1	1.53	174	5.22	2.98	0.156
parallel-SF-PRM	48.8	1.64	42.2	1.3	4.51	0.1	30.3	1.33	313	4.02	1.25	0.04
hybrid-BFS-CC	28	1.3	25.9	13.3	0.111	0.009	22.1	1.51	304	304†	0.191	0.021
multistep-CC	9.74	1.29	15.9	2.06	0.23	0.05	27.0	1.22	343	343†	0.16	0.06

Table 2. Times (seconds) for connected components labeling. (40h) indicates 40 cores with hyper-threading. *We use the timing for the sequential spanning forest code from Patwary et al. [48] as we found it to be faster than the PBBS implementation. †We use the sequential time as the parallel time was higher due to overheads of parallel execution.



(a) random (b) rMat (c) rMat2

(d) 3D-grid (e) line (f) com-Orkut

Figure 2. Times versus number of threads on a 40-core machine with hyper-threading. (40h) indicates 80 hyper-threads.

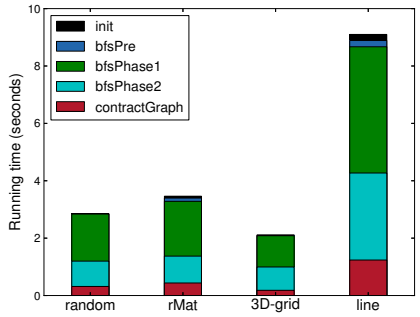


Figure 5. Breakdown of timings on 40 cores with hyper-threading for decomp-min-CC.

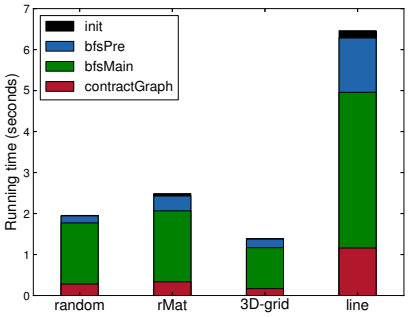
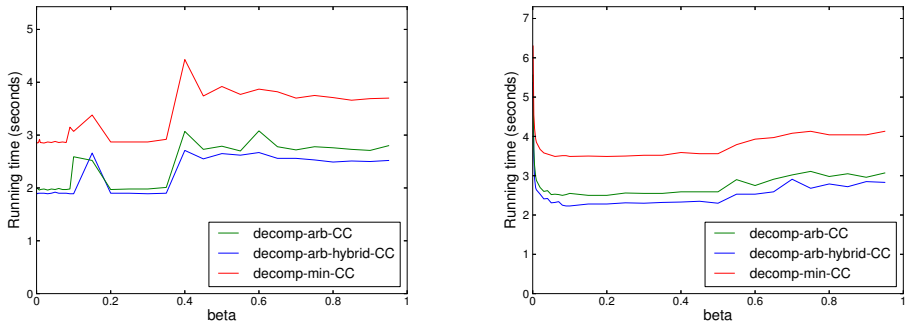


Figure 6. Breakdown of timings on 40 cores with hyper-threading for decomp-arb-CC.

Figure 5 shows the breakdown of the 40-core running time for decomp-min-CC on several graphs. In the figure, "init" refers to the time for generating random permutations and initializing arrays, "bfsPre" refers to adding new vertices to the BFS frontier and computing offsets into shared arrays for the frontier vertices, "bfsPhase1" refers to the first phase (Lines 9–23 of Algorithm 2), "bfsPhase2" refers to the second phase (Lines 24–39 of Algorithm 2), and "contractGraph" includes the time for removing duplicate edges, renumbering vertices and edges, creating the contracted graph for the recursive call, and relabeling after the recursive call. We see that 80–90% of the time is spent in the two BFS phases, with the first phase being the more expensive of the two.

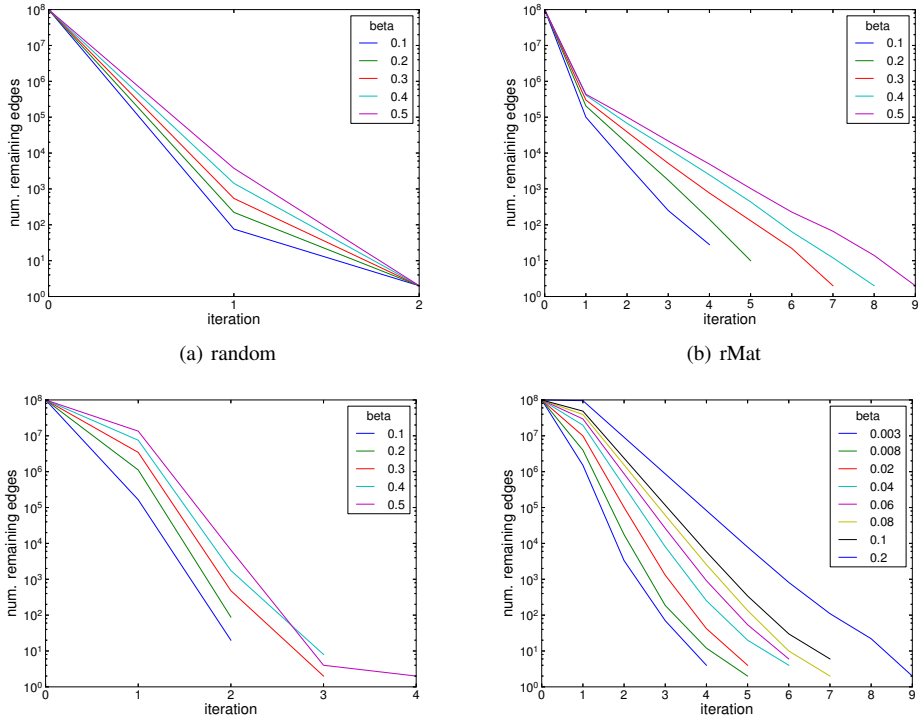
Figure 6 shows the breakdown of the running time for decomp-arb-CC on 40 cores on several inputs. "bfsMain" refers to the single phase of the BFS iteration (Lines 9–20 of Algorithm 3), and the other sub-timings have the same meaning as in the previous paragraph. The majority of the time (55–75%) is spent in the main BFS phase. Compared to decomp-min-CC, the savings in running time



(a) random (b) rMat

(c) 3D-grid (d) line

Figure 3. Running time versus β on various input graphs on a 40-core machine using 80 hyper-threads.



(a) random (b) rMat

(c) 3D-grid (d) line

Figure 4. Number of remaining edges per iteration versus β of decomp-arb-hybrid-CC on various graphs.

of decomp-arb-CC comes from this part of the computation due to requiring only one pass over the edges.

Figure 7 shows the breakdown of the 40-core running time for decomp-arb-hybrid-CC. "bfsSparse" refers to the time spent in the main phase of the BFS when performing the write-based computation for sparse frontiers, and "bfsDense" refers to the time spent in the main phase performing the read-based computation on the dense frontiers. As noted in Section 4, a post-processing step to filter out the intra-component edges is required, and "filterEdges" refers to this phase. We see that for 3D-grid and line, the frontier

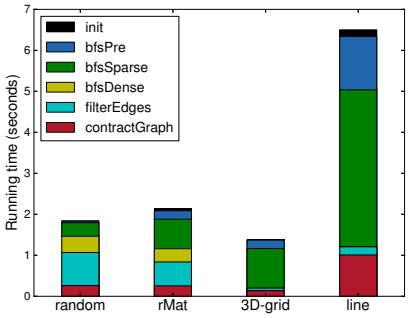


Figure 7. Breakdown of timings on 40 cores with hyper-threading for decomp-arb-hybrid-CC.

never becomes dense enough to switch to the read-based computation, hence all of the BFS time is captured by bfsSparse. On the other hand, random and rMat do have BFS frontiers that become dense enough where the read-based computation is invoked. Since they switch to the read-based computation, some edges do not get inspected and hence the filterEdges phase performs more work to filter out the intra-component edges. For random and rMat, about 40% of the time is spent in the main BFS phase.

Figure 8 shows the running time of decomp-arb-hybrid-CC on 80 hyper-threads as a function of graph size for random graphs with sizes from $m = 5 \times 10^7$ to 5×10^8, and $n = m/5$. The running time increases almost linearly as we increase the graph size.

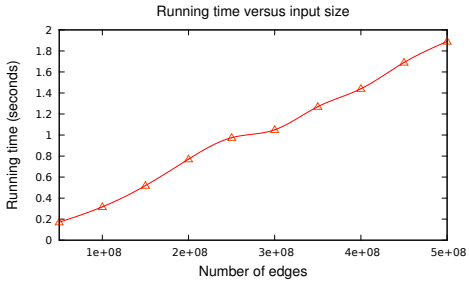


Figure 8. Running time of decomp-arb-hybrid-CC vs. problem size for random graphs on 40 cores with hyper-threading.

Besides PBBS and the implementations by Patwary et al., Bader and Cong describe a parallel spanning tree implementation based on parallel depth-first search [3]. However, Patwary et al. [48] show that their implementations are faster than Bader and Cong's implementation. Galois [46] also contains implementations of connected components based on union-find, but we found them to be slower than the implementation by Patwary et al, decomp-arb-hybrid-CC, and decomp-arb-CC for all of the input graphs. Several graph processing systems [54, 32, 37, 38] have connected components implementations based on label propagation, but the depth of the algorithm is proportional to the diameter of the graph and the algorithm is not work-efficient. As noted in [54], this algorithm usually does not perform as well as linear or near-linear work algorithms.

6. CONCLUSION

We have presented a simple linear-work parallel algorithm for finding the connected components of a graph. Our algorithm is the first practical work-efficient parallel algorithm with polylogarithmic depth for this problem. We present implementations of our algorithm and experimentally show that it is competitive with the fastest existing parallel algorithms for finding the connected components of a graph.

Acknowledgements. This work is supported by the National Science Foundation under grant number CCF-1314590, the Intel Labs Academic Research Office for the Parallel Algorithms for Non-Numeric Computing Program, the Intel Science and Technology Center for Cloud Computing (ISTC-CC) and a Facebook Graduate Fellowship. We thank Jeremy Fineman, Phillip Gibbons, Gary Miller and Shen Chen Xu for helpful discussions.

References

[1] A. Agrawal, L. Nekludova, and W. Lim. A parallel $O(\log N)$ algorithm for finding connected components in planar images. In *ICPP*, pages 783–786, 1987.

[2] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for Ultracomputer and PRAM. In *ICPP*, pages 177–187, 1983.

[3] D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). *Journal of Parallel and Distrib. Comput.*, 65(9):994–1006, 2005.

[4] D. A. Bader, G. Cong, and J. Feo. On the architectural requirements for efficient execution of graph algorithms. In *ICPP*, pages 547–556, 2005.

[5] D. A. Bader and J. JaJa. Parallel algorithms for image histogramming and connected components with an experimental study. *J. Parallel Distrib. Comput.*, 35(2):173–190, 1996.

[6] D. S. Banerjee and K. Kothapalli. Hybrid algorithms for list ranking and graph connected components. In *High Performance Computing*, pages 1–10, 2011.

[7] Y. Bartal. Graph decomposition lemmas and their role in metric embedding methods. In *ESA*, pages 89–97. 2004.

[8] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *Supercomputing*, pages 12:1–12:10, 2012.

[9] G. E. Blelloch. NESL. In *Encyclopedia of Parallel Computing*, pages 1278–1283. 2011.

[10] G. E. Blelloch, A. Gupta, I. Koutis, G. L. Miller, R. Peng, and K. Tangwongsan. Near linear-work parallel SDD solvers, low-diameter decomposition, and low-stretch subgraphs. In *SPAA*, pages 13–22, 2011.

[11] G. E. Blelloch and B. M. Maggs. Parallel algorithms. In *The Computer Science and Engineering Handbook*, pages 277–315. 1997.

[12] L. Bus and P. Tvrdik. A parallel algorithm for connected components on distributed memory machines. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 280–287. 2001.

[13] E. Caceres, H. Mongelli, C. Nishibe, and S. W. Song. Experimental results of a coarse-grained parallel algorithm for spanning tree and connected components. In *High Performance Computing & Simulation*, pages 631–637, 2010.

[14] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *SDM*, pages 442–446, 2004.

[15] F. Y. Chin, J. Lam, and I.-N. Chen. Efficient parallel algorithms for some graph problems. *Commun. ACM*, pages 659–665, 1982.

[16] K. Chong and T. Lam. Finding connected components in $O(\log n \log \log n)$ time on the EREW PRAM. *Journal of Algorithms*, 18(3):378–402, 1995.

[17] R. Cole, P. N. Klein, and R. E. Tarjan. Finding minimum spanning forests in logarithmic time and linear work using random sampling. In *SPAA*, pages 243–250, 1996.

[18] R. Cole and U. Vishkin. Approximate parallel scheduling. II. applications to logarithmic-time optimal parallel graph algorithms. *Information and Computation*, 92(1):1–47, 1991.

[19] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. Comput.*, 20(6):1046–1067, Dec. 1991.

[20] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *FOCS*, pages 698–710, 1991.

[21] S. Goddard, S. Kumar, and J. F. Prins. Connected components algorithms for mesh-connected parallel computers. In *Parallel Algorithms: 3rd DIMACS Implementation Challenge*, pages 43–58, 1995.

[22] J. Greiner. A comparison of parallel algorithms for connected components. In *SPAA*, pages 16–25, 1994.

[23] S. Halperin and U. Zwick. An optimal randomized logarithmic time connectivity algorithm for the EREW PRAM. *J. Comput. Syst. Sci.*, 53(3):395–416, 1996.

[24] S. Halperin and U. Zwick. Optimal randomized EREW PRAM algorithms for finding spanning forests. In *J. Algorithms*, pages 1740–1759, 2000.

[25] S. Hambrusch and L. TeWinkel. A study of connected component labeling algorithms on the MPP. In *Supercomputing*, pages 477–483, 1988.

[26] Y. Han and R. A. Wagner. An efficient and fast parallel-connected component algorithm. *J. ACM*, 37(3):626–642, July 1990.

[27] K. A. Hawick, A. Leist, and D. P. Playne. Parallel graph component labelling with GPUs and CUDA. *Parallel Comput.*, 36(12):655–678, Dec. 2010.

[28] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Commun. ACM*, 22(8):461–464, Aug. 1979.

[29] T.-S. Hsu, V. Ramachandran, and N. Dean. Parallel implementation of algorithms for finding connected components in graphs, 1997.

[30] K. Iwama and Y. Kambayashi. A simpler parallel algorithm for graph connectivity. *J. Algorithms*, 16(2):190–217, Mar. 1994.

[31] D. B. Johnson and P. Metaxas. Connected components in $O(\log^{3/2} n)$ parallel time for the CREW PRAM. *Journal of Computer and System Sciences*, 54(2):227–242, 1997.

[32] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: mining peta-scale graphs. *Knowl. Inf. Syst.*, 27(2):303–325, 2011.

[33] D. R. Karger, N. Nisan, and M. Parnas. Fast connected components algorithms for the EREW PRAM. *SIAM J. Comput.*, 28(3):1021–1034, Feb. 1999.

[34] V. Koubek and J. Krsnakova. Parallel algorithms for connected components in a graph. In *Fundamentals of Computation Theory*, pages 208–217. 1985.

[35] A. Krishnamurthy, S. S. Lumetta, D. E. Culler, and K. Yelick. Connected components on distributed memory machines. In *Parallel Algorithms: 3rd DIMACS Implementation Challenge*, pages 1–21, 1994.

[36] C. Kruskal, L. Rudolph, and M. Snir. Efficient parallel algorithms for graph problems. *Algorithmica*, 5(1-4), 1990.

[37] A. Kyrola, G. E. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *Operating System Design and Implementation*, pages 31–46, 2012.

[38] A. Kyrola, J. Shun, and G. E. Blelloch. Beyond synchronous computation: New techniques for external memory graph algorithms. In *Symposium on Experimental Algorithms*, 2014.

[39] C. E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[40] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *SPAA*, pages 303–314, 2010.

[41] W. Lim, A. Agrawal, and L. Nekludova. A fast parallel algorithm for labeling connected components in image arrays. In *Tech. Report NA86-2, Thinking Machines Corporation*, 1986.

[42] N. Linial and M. Saks. Low diameter graph decompositions. *Combinatorica*, 13(4):441–454, 1993.

[43] Y. Matias and U. Vishkin. On parallel hashing and integer sorting. *Journal of Algorithms*, 12(4):573–606, 1991.

[44] G. L. Miller, R. Peng, and S. C. Xu. Parallel graph decomposition using random shifts. In *SPAA*, pages 196–203, 2013.

[45] D. Nath and S. N. Maheshwari. Parallel algorithms for the connected components and minimal spanning tree problems. *Inf. Process. Lett.*, 14(1):7–11, 1982.

[46] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Symposium on Operating Systems Principles*, pages 456–471, 2013.

[47] N. Nisan, E. Szemeredi, and A. Wigderson. Undirected connectivity in $O(\log^{1.5} n)$ space. In *FOCS*, pages 24–29, 1992.

[48] M. Patwary, P. Refsnes, and F. Manne. Multi-core spanning forest algorithms using the disjoint-set data structure. In *IPDPS*, pages 827–835, 2012.

[49] S. Pettie and V. Ramachandran. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM J. Comput.*, 31(6):1879–1895, 2002.

[50] C. A. Phillips. Parallel graph contraction. In *SPAA*, pages 148–157, 1989.

[51] C. K. Poon and V. Ramachandran. A randomized linear work EREW PRAM algorithm to find a minimum spanning forest. In *ISAAC*, pages 212–222, 1997.

[52] J. Reif. Optimal parallel algorithms for integer sorting and graph connectivity. *TR-08-85, Harvard University*, 1985.

[53] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982.

[54] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Principles and Practice of Parallel Programming*, pages 135–146, 2013.

[55] J. Shun and G. E. Blelloch. Phase-concurrent hash tables for determinism. In *SPAA*, 2014.

[56] J. Shun, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Reducing contention through priority updates. In *SPAA*, pages 152–163, 2013.

[57] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *SPAA*, pages 68–70, 2012.

[58] G. M. Slota, S. Rajamanickam, and K. Madduri. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *IPDPS*, 2014.

[59] J. Soman, K. Kishore, and P. J. Narayanan. A fast GPU algorithm for graph connectivity. In *IPDPS*, pages 1–8, 2010.

[60] U. Vishkin. An optimal parallel connectivity algorithm. *Discrete Applied Mathematics*, 9(2):197–207, 1984.