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 applications. Sequentially, connectivity can be done easily using a simple breadth-first search or depth-first search in linear work. There have been many parallel algorithms for connectivity. However the simpler parallel algorithms require superlinear 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. [42], which uses parallel breadth-first searches. We discuss a (modest) variant of their decomposition algorithm which preserves the theoretical complexity of the algorithm while leading to a simpler and faster implementation. 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 and show that we are competitive for large input graphs (0.88–1.41 times faster on a 40-core machine). In addition, we compare our algorithms to the fastest sequential connectivity algorithm, and show that we achieve 9–19 times speedup relative to the sequential implementation on 40 cores. We discuss the various optimizations used in our algorithms and present extensive experimental analysis of the performance of our algorithms. 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

General Terms: Theory, Experiments

Keywords: Parallel Algorithms, Graph Connectivity, Experiments

1. INTRODUCTION

Finding the connected components of a graph is a fundamental problem in computer science. The problem takes as input an undirected graph for 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. 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. However, computing connected components and spanning forests[1] in parallel has been a long studied problem [28, 34, 39, 56, 48, 50, 26, 2, 1, 51, 16, 31, 47, 33, 18, 30, 15, 43, 36, 45]. Some of the parallel algorithms developed are relatively simple, but require superlinear work. The algorithms of Shiloach and Vishkin [51] 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 in 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 [50] and Phillips [48] work by contracting vertices in the same component together. These algorithms 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 [19, 23, 17, 49, 47, 24]. 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, 47]. However, these algorithms are complicated and unlikely to be practical (there are no implementations of these algorithms).

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 [10]. Goddard et al. [21], Hsu et al. [29], Bader et al. [3, 4] and Patwary et al. [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, 55, 6]. There have also been connectivity algorithms that require time proportional to the diameter of the graph in recent graph processing packages [32, 52, 37]. The fastest publicly available implementation of connected components that we are aware of is based on the spanning forest code of

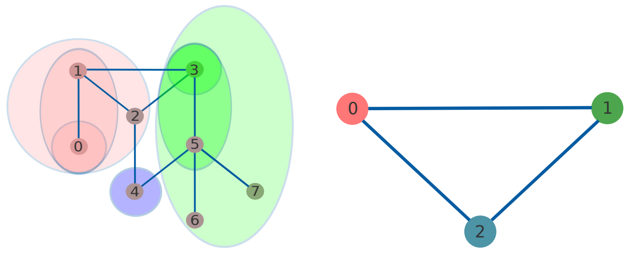
[1]Note that a spanning forest algorithm can be used to compute connected components.

the Problem Based Benchmark Suite [54]. None of the previous parallel algorithms implemented are theoretically work-efficient.

In this paper, we present a 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 connectivity algorithm with an implementation, and the implementation also performs well in practice.

Our algorithm is based on a recent simple parallel algorithm for generating low-diameter decompositions of graphs by Miller et al. [42], which is an improvement of an algorithm by Blelloch et al. [11]. 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 [40]. Such decompositions have many uses in computer science, including in linear system solvers [11] 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 in expectation. An illustration of this algorithm is shown in Figure 1. Our simple implementation is based on parallel breadth-first searches and some simple parallel routines.



(a) graph decomposition (b) contracted graph

Figure 1. Illustration of 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 vertex 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.

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 connected components and apply various optimizations to our implementations.

We experimentally compare our algorithm against the fastest existing parallel connectivity implementations (based on spanning forest) [54, 46] and show that our algorithm is competitive. In particular, on a 40 core machine with hyper-threading we are 0.88–1.41 times faster than the fastest existing parallel implementation on a variety of input graphs. Furthermore we achieve 17–35 times speedup over the parallel implementation run on a single thread and 9–19 times speedup over the fastest sequential implementation. We show that on many 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.

Contributions. In summary, 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 previous 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 use both the arbitrary and priority write versions of the CRCW PRAM, where a priority write here means that the minimum (or maximum) value written concurrently is stored. 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.

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 otherwise $L(u) \neq L(v)$.

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 [9].

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

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

vertices in V_i is at most d, and (2) the number of edges (u, v) 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 [42], 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 $O(\log^2 n/\beta)$ 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 $O(\log n/\beta)$ 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. Since the maximum shift value is $O(\log n/\beta)$, the algorithm terminates in $O(\log n/\beta)$ iterations. Each iteration requires $O(\log n)$ depth for packing the frontiers of the BFS's, leading to an overall depth of $O(\log^2 n/\beta)$ 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 a 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. This leads to a linear-work parallel connectivity algorithm, assuming the contraction and relabeling can be done efficiently.

The pseudocode for this algorithm 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 result of the recursive call to CC.

Algorithm 1 Parallel decomposition-based algorithm for connected components labeling

```
1: procedure CC($G(V, E)$)
2:     $\beta$ = some constant fraction
3:     $L$ = DECOMP($G(V, E)$, $\beta$)
4:                       ▷ $L$ are 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 $O(m)$ work and $O(\log^3 n)$ depth in expectation on a CRCW PRAM.*

PROOF. The algorithm sets β to a constant fraction. Since the number of edges decreases to at most βm in expectation after each recursive call, the total number of calls is $O(\log_{1/\beta} m)$ in expectation. Each recursive call requires $O(\log^2 n)$ depth and $O(m')$ expected work where m' is the number of remaining edges for DECOMP. Hence the total contribution of DECOMP to the depth of CC is $O(\log_{1/\beta} m \frac{\log^2 n}{\beta}) = O(\log^3 n)$ 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 which need to start their own BFS are added to the end of this frontier array. We store all of the frontiers created throughout one call to DECOMP, and there are $O(\log n/\beta)$ 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 the size of each component using prefix sums over all the $O(\log n/\beta)$ frontiers for each BFS. For each iteration of CC, the overall work is $O(n')$ where n' is the number of vertices at the beginning of the iteration, since the total size of the frontiers is $O(n')$, and the depth is $O(\log n'/\beta)$.

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 preserved 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 in $O(m')$ work and $O(\log m')$ depth using hashing [41], 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 logarithmic depth per iteration.

We summarize the proof of this theorem. There are $O(\log n)$ calls to DECOMP, 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, giving overall $O(\log^3 n)$ depth in expectation. 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) in the BFS's. This gives us a linear-work algorithm with $O(\log^2 n \log^* n)$ expected depth.

We consider a slight variation of DECOMP which breaks ties arbitrarily between vertices visiting the same new vertex 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 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 $O(\frac{\log^2 n}{\beta})$ depth and $O(m)$ work in expectation.*

PROOF. Since we are still picking values from an exponential distribution, the diameter of each component is $O(\frac{\log n}{\beta})$ in w.h.p. as shown in [42]. 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 [42], consider the midpoint w of an edge (u,v). Lemma 4.3 of [42] 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 only if the two original shift distances were within 2 of each other. In other words, suppose the two distances we are comparing are shift distances d_1 and d_2. Then $||d_2| - \lfloor d_1 \rfloor| \leq 1$ only if $|d_2 - d_1| < 2$. Hence we can modify Lemma 4.3 of [42] 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 [42] uses properties of the exponential distribution to show that the probability that the sh 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. We have $c = 2$, so the probability that an edge is an inter-component edge is at most 2β. Hence 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 $\beta < 1/2$.

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, caching performance, and synchronization primitives used. Therefore, in this section we describe our algorithmic engineering efforts to obtain a fast implementation of Algorithm 1. We describe the two versions of DECOMP, referring to the original algorithm as ***Decomp-Min*** and the version which breaks ties arbitrarily as Decomp-Arb.

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 [42], 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 ordering 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. 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.

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 [54].

Decomp-Min is split into two phases over the frontier vertices (pseudocode shown in Algorithm 2). In our implementation, we use the array C to to store both the component ID's of the vertices and the ID of the neighboring vertex that successfully reserves it. 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 on pairs: writeMin takes two arguments, a location *loc* storing a pair *val* as the first argument and a pair as the second argument; it atomically updates *loc* with *val* if the first argument of the pair *val* is less than the first argument of the pair stored at *loc*, and otherwise it does nothing. This operation can be implemented as a priority update described by Blelloch et al. [53]. Note that instead of keeping pairs in C we could keep two arrays, one to store the component IDs and the other to store the winning vertex ID, but this leads to an additional cache miss per visit to a vertex.

The pair values of C are initialized to (∞, ∞) in 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 ID. In our implementation, inter-component edges are kept while intra-component edges are deleted. We overwrite the E array 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 shift value δ_v, denoted by $\delta_{v,frac}$. 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 u, $C_1[u] \neq -1$, this means the neighbor has not been visited in a *previous iteration*. In this case we need to keep the edge to u (Lines 17–18) as currently it does not know whether it is an intra- or inter-component edge (this can only be decided once all other frontier vertices finish doing their writeMin's). Otherwise, the neighbor u has been visited in a previous iteration and we can determine the status of the edge to u—if u has component label different from the vertex, it keeps the edge since it is an inter-component edge (Lines 20–22). It labels the endpoint of the edge with its new component ID (so that it doesn't 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 determined from the first phase) we determine whether vertex v successfully marked the neighbor with $\delta_{v,frac}$ (Line 30). If successful, we do not keep the edge (it is an intra-component edge) and also store a value of -1 in the first entry of $C[w]$ so that future writeMin's will not mark it again (Line 31), since the check on Line 14 will fail. In this case vertex v is also responsible for adding w to the next frontier (Line 32). Otherwise another vertex marked the neighbor w so we check whether the component ID of w matches that of v. If they differ, then the edge is an inter-component edge and we keep it (Lines 34–36). We set the sign bit of the value of its component ID and store it in E (Lines 35–36). Otherwise we do not keep the edge. If the edge has a negative value, then it was already processed in the first phase, and we just keep it (Lines 37–39). We set the degree of v to be the number of inter-component edges incident on v (Line 40).

Note that for high-degree vertices (e.g. degree $\Omega(\log n)$), 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. Also, since the inter-component edge targets have the sign bit set, we unset their sign bit during the relabeling phase after the call to DECOMP in the connected components algorithm.

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 (pseudocode 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 the frontiers. Instead of using a writeMin as in Decomp-Min, Decomp-Arb uses a compare-and-swap (CAS) to mark an unvisited neighbor (Line 14). 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 different from its own, it keeps the edge as an inter-component edge (Lines 17–20). It also marks the endpoint of the edge with its component ID so that

Algorithm 2 Decomp-Min

```
 1: $C = \{(\infty,\infty),\ldots,(\infty,\infty)\}$
 2: Frontier = {}
 3: numVisited = 0
 4: while (numVisited $< n$) do
 5:     add to Frontier unvisited vertices $v$ with $\delta_v <$ round + 1
 6:       and set $C[v] = (-1, v)$                     ▷ new BFS centers
 7:     numVisited = numVisited + size(Frontier)
 8:     NextFrontier = {}
 9:     parfor $v \in$ 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:             $w = E[\text{start} + i]$
14:             if $C_1[w] \neq -1$ then
15:                 if $C_1[w] > \delta_{v,frac}$ then
16:                     writeMin($C[w]$,($\delta_{v,frac}, C_2[v]$))
17:                 $E[\text{start} + k] = w$
18:                 $k = k + 1$
19:             else
20:                 if $C_2[w] \neq C_2[v]$ then
21:                     $E[\text{start} + k] = -C_2[w] - 1$
22:                     $k = k + 1$
23:         $D[v] = k$
24:     parfor $v \in$ 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:             $w = E[\text{start} + i]$
29:             if $w \geq 0$ then
30:                 if $C_1[w] = \delta_{v,frac}$ then          ▷ $v$ won on $w$
31:                     $C_1[w] = -1$
32:                     add $w$ to NextFrontier
33:                 else
34:                     if $C_2[w] \neq C_2[v]$ then
35:                         $E[\text{start} + k] = -C_2[w] - 1$
36:                         $k = k + 1$
37:             else
38:                 $E[\text{start} + k] = w$
39:                 $k = k + 1$
40:         $D[v] = k$
41:     NextFrontier = Frontier
```

it doesn't have to be relabeled later (Line 19). Note that although the code shown does not make use of the fact that the degree is set to the number of inter-component edges on Line 21, we make use of it during the relabeling phase (not shown in the pseudocode).

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

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 [54] 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.

As the experiments in Section 5 show, Decomp-Arb performs

Algorithm 3 Decomp-Arb

```
1: $C = \{\infty, \ldots, \infty\}$
2: Frontier = {}
3: numVisited = 0
4: while (numVisited $< n$) do
5:     add to Frontier unvisited vertices $v$ with $\delta_v <$ round $+ 1$
6:         and set $C[v] = v$                    ▷ new BFS centers
7:     numVisited = numVisited + size(Frontier)
8:     NextFrontier = {}
9:     parfor $v \in$ 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:            $w = E[\text{start} + i]$
14:            if $C[w] = \infty$ and CAS$(C[w], \infty, C[v])$ then
15:                add $w$ to NextFrontier
16:            else
17:                if $(C[w] \neq C[v])$ then
18:                                             ▷ inter-component edge
19:                    $E[\text{start} + k] = C[w]$
20:                    $k = k + 1$
21:        $D[v] = k$
22:    NextFrontier = Frontier
```

better than Decomp-Min due to only requiring one pass over the edges of each frontier during the BFS's, and needing less book-keeping overall.

We considered the direction-optimizing BFS idea [52, 8], which reduces the number of edges traversed during a standard BFS. However, we found it not to give us an advantage in our algorithm, since we are required to look at all of the edges anyway to decide whether it is an inter-component edge or an intra-component edge.

5. EXPERIMENTS

We compare our connectivity algorithm using the two versions of the decomposition algorithm described, and also compare it to the fastest available parallel connectivity algorithms we are aware of [54, 46]. We refer to our connectivity algorithm using Decomp-Min as ***decomp-min-CC*** and using Decomp-Arb as ***decomp-arb-CC***. The fastest publicly available implementation is part of the Problem Based Benchmark Suite (PBBS) [54] and is in fact a parallel spanning forest algorithm that uses union-find to identify and link components (***parallel-SF-PBBS***). We also compare with recent parallel implementations by Patwary et al. [46]. They describe two implementations based on union-find in their paper, which we refer to as ***parallel-SF-PRM-lock*** and ***parallel-SF-PRM-verify***. We note that the existing parallel implementations are not theoretically work-efficient. We compare these parallel implementations to a simple sequential spanning forest-based connectivity algorithm using union-find (***serial-SF***), which we obtained from the authors of [46]. We also tried a sequential connectivity algorithm based on breadth-first search, but found it to be slower. For the spanning forest algorithms, we include in the timings a post-processing step that finds the ID of the root of the tree for each vertex.

We run our experiments on a 40-core (with hyper-threading) machine with $4 \times$ 2.4GHZ Intel 10-core E7-8870 Xeon processors, a 1066MHz bus, and 256GB of main memory. We ran all parallel experiments with two-way hyper-threading enabled, for a total of 80 threads. We compiled all of our code with `g++` version 4.8.0 with the `-O2` flag. The parallel codes use Cilk Plus [38] to express parallelism, which is supported by the `g++` compiler that we use. The times that we report are based on a median of three trials.

We used a variety of input graphs, the first three of which are taken from PBBS [54]. ***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. ***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 $n - 1$. For all of the graphs, the vertex labels are randomly assigned. The sizes of the graphs are shown in Table 1. For our decomposition-based algorithms we store the edge in each direction, so we use twice the number of edges than as noted in Table 1.

Input Graph	Num. Vertices	Num. Edges
random	10^8	5×10^8
rMat	2^{27}	5×10^8
3D-grid	10^8	3×10^8
line	5×10^8	5×10^8

Table 1. Graph inputs

The serial and parallel running times of the implementations on the various inputs are summarized in Table 2. We see that decomp-arb-CC outperforms decomp-min-CC for all inputs by 35–50%. This is because (1) decomp-arb-CC requires 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. Among the three existing parallel implementations from PBBS and Patwary et al., parallel-SF-PBBS is the fastest in parallel, although the implementations by Patwary et al. are faster sequentially (we note that parallel-SF-PRM-verify did not run for the rMat and line graphs). We compare the parallel running times of our decomposition-based connectivity algorithms to parallel-SF-PBBS. Compared to parallel-SF-PBBS, decomp-arb-CC is faster on random (by 24%) and 3D-grid (by 41%), about the same on rMat and slower on the degenerate line graph (by 12%). Decomp-min-CC is slower than parallel-SF on all inputs. Compared to the serial implementations, our fastest parallel implementation (decomp-arb-CC) on a single thread is about 30–50% slower. Decomp-arb-CC achieves a self-relative speedup of 17–35 and a speedup of 9–19 relative to the sequential implementation.

Figure 2 shows the running time versus the number of threads for the different implementations. We see that decomp-arb-CC performs the best for all thread numbers. Figure 3 shows the speedups of the parallel implementations relative to serial-SF. We note that the implementations by Patwary et al. perform best up to about 16 threads, after which the other parallel implementations outperform them. At 40 threads the fastest implementation is decomp-arb-CC on random, rMat and 3D-grid, and the fastest implementation is parallel-SF-PBBS for the line graph.

Figure 4 shows the 40-core running time of decomp-arb-CC and decomp-min-CC as a function of the parameter β. We see that the trends for the two implementations are similar, and the β leading to the fastest running times is between 0.1 and 0.3. Figure 5 shows the number of edges remaining per iteration for decomp-arb-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 (an order of magnitude more than predicted by the upper bound) in the number of remaining edges per iteration.

Implementation	random (1)	random (40h)	rMat (1)	rMat (40h)	3D-grid (1)	3D-grid (40h)	line (1)	line (40h)
serial-SF	25.5	–	24.5	–	20	–	135*	–
decomp-arb-CC	43.3	2.2	47	2.74	30.1	1.47	253	7.04
decomp-min-CC	74.8	3.01	76.7	3.82	58.7	2.16	413	10.1
parallel-SF-PBBS	60	2.73	65	2.73	49.4	2.08	197	6.16
parallel-SF-PRM-lock	33.4	3.99	36.6	4.15	30.3	2.96	263	9.04
parallel-SF-PRM-verify	29.8	4.21	–†	–†	25.1	3.43	–†	–†

Table 2. Times (seconds) for connected components labeling. (40h) indicates the time on 40 cores with hyper-threading. †parallel-SF-PRM-verify did not run on the input graph. *For the line graph, we used the timing for the sequential spanning forest code from PBBS [54] as we found it to be faster than the implementation obtained from Patwary et al. [46].

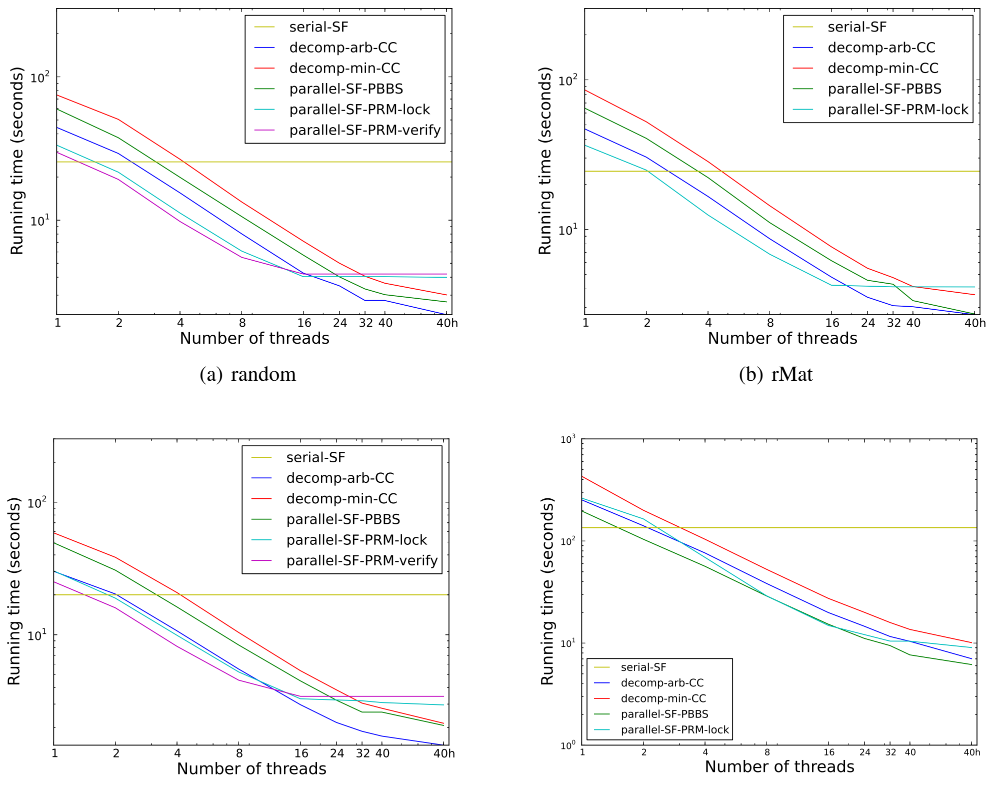


Figure 2. Times versus number of threads on various input graphs on a 40-core machine with hyper-threading. (40h) indicates 80 hyper-threads.

Figure 6 shows the breakdown of the running time for decomp-min-CC and decomp-arb-CC on 40 cores. We see that 80–90% of the time is spent in computing the decompositions (labeled *decomp*). The rest of the time is spent on graph contraction and relabeling (labeled *contract*), and initialization routines (labeled *init*) such as generating the random permutations and initializing arrays.

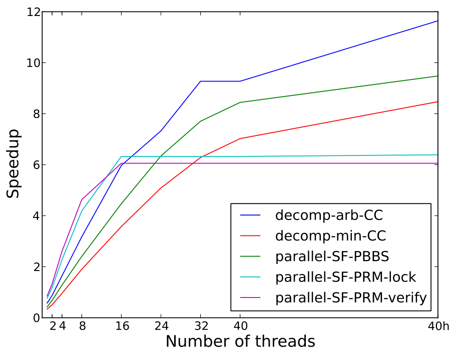
Figure 7 shows the running time of decomp-arb-CC on 80 hyper-threads as a function of graph size for random graphs from $m = 5 \times 10^7$ to 5×10^8, and $n = m/10$. The running time increases almost linearly as we increase the graph size.

Besides PBBS and the implementations by Patwary et al., Bader and Cong describe a parallel spanning tree implementation which is based on parallel depth-first search [3]. However, Patwary et al. [46] show that their implementations are faster than Bader and Cong's. Galois [44] also contains implementations of connected components based on union-find, but we found them to be slower than the PBBS baseline implementation. Except for the line graph, their implementations were also slower than the implementation by Patwary et al. Using 80 hyper-threads, their fastest implementation takes 7.3 seconds for random, 7.7 seconds for rMat, 7.04 seconds for 3D-grid and 7.45 seconds for the line graph.

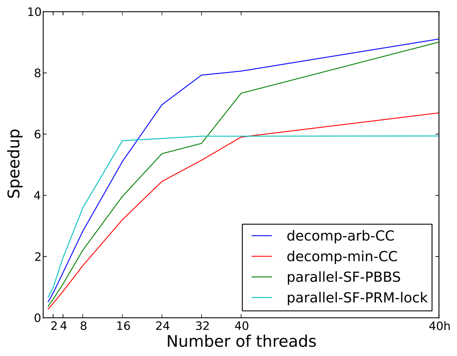
Several graph processing systems [52, 32, 37] have connected components implementations based on vertex ID propagation: each vertex starts with a unique ID and in each iteration every vertex updates its ID to be the minimum of its own ID and all of its neighbors IDs. This algorithm terminates when no IDs change in a round. The depth of the algorithm is proportional to the diameter of the graph and the algorithm is not work-efficient. Ligra [52] implements this algorithm for shared-memory. We ran the Ligra connectivity code on the 40-core machine, and their parallel time on 80 hyper-threads was 9 seconds for random, 8 seconds for rMat, and 43 seconds for 3D-grid. For the line graph, the algorithm performs an order of magnitude worse than the other graphs due to its high diameter (this algorithm was not designed for such graphs). As expected, the Ligra times are much slower than those reported in Table 2, and this algorithm is not competitive with standard connectivity algorithms.

6. CONCLUSION

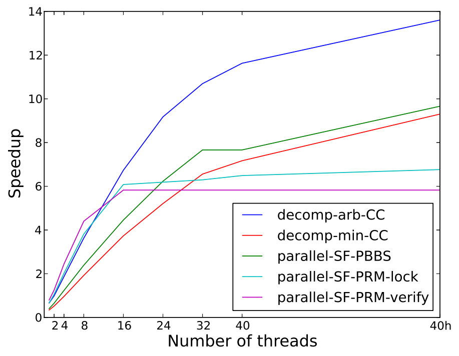
We have presented a simple linear-work parallel algorithm for finding the connected components of a graph. Our algorithm is the



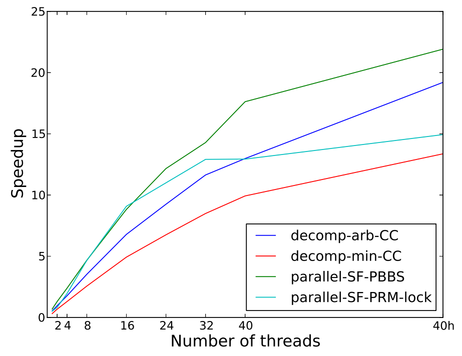
(a) random



(b) rMat

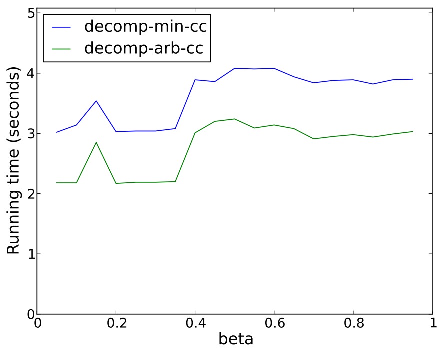


(c) 3D-grid

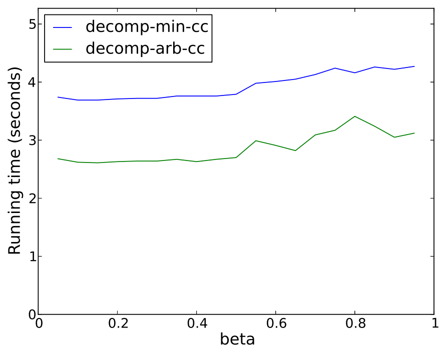


(d) line

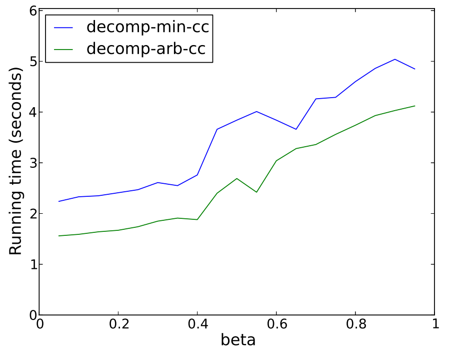
Figure 3. Speedup relative to serial-SF on various input graphs on a 40-core machine with hyper-threading. (40h) indicates 80 hyper-threads.



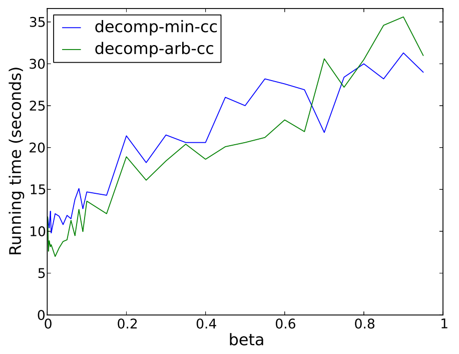
(a) random



(b) rMat

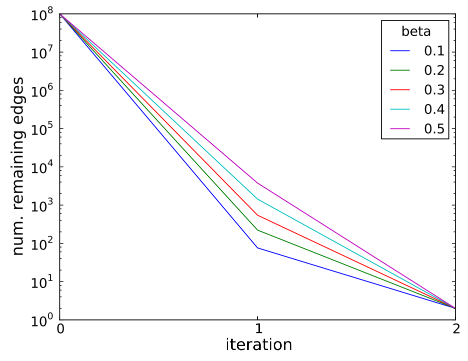


(c) 3D-grid

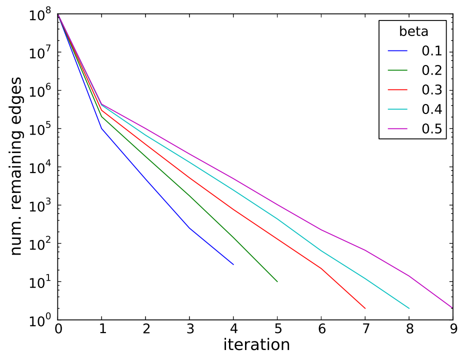


(d) line

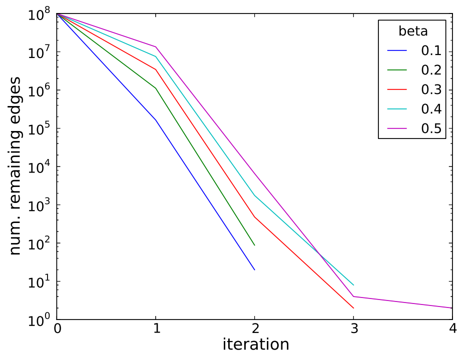
Figure 4. Running time versus β of decomp-arb-CC on various input graphs on a 40-core machine using 80 hyper-threads.



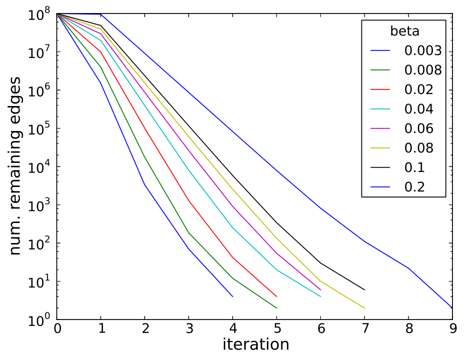
(a) random



(b) rMat



(c) 3D-grid



(d) line

Figure 5. Number of remaining edges per iteration versus β of decomp-arb-CC on various input graphs.

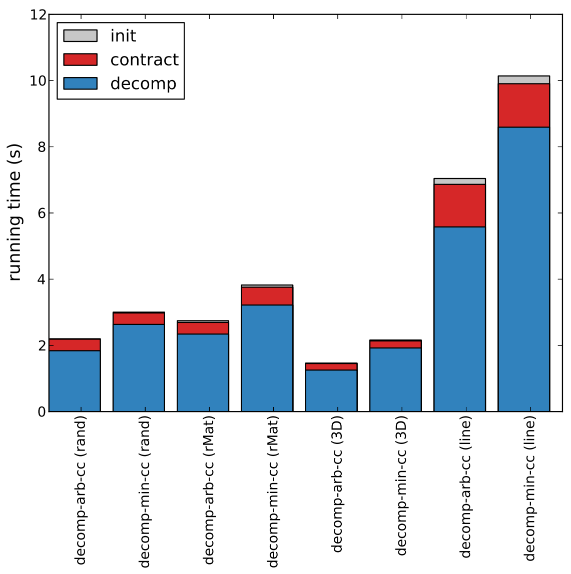


Figure 6. Breakdown of timings on 40 cores with hyper-threading.

first practical work-efficient parallel algorithm for this problem. We present implementations of our algorithm and show that it is competitive with the fastest existing algorithm for finding the connected components of a graph. Improving the depth of our algorithm would be an interesting direction for future work.

References

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

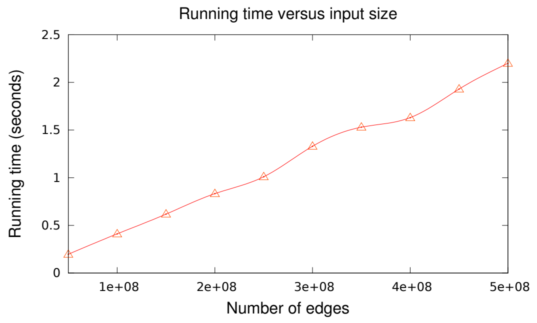


Figure 7. Running time vs. problem size for random graphs on 40 cores with hyper-threading.

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

[3] D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 65(9), 2005.

[4] D. A. Bader, G. Cong, and J. Feo. On the architectural requirements for efficient execution of graph algorithms. In *ICPP*, 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), 1996.

[6] D. S. Banerjee and K. Kothapalli. Hybrid algorithms for list ranking and graph connected components. In *HiPC*, 2011.

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

[8] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *Supercomputing*, 2012.

[9] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39, March 1996.

[10] G. E. Blelloch. NESL. In *Encyclopedia of Parallel Computing*. 2011.

[11] 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*, 2011.

[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*. 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 *HPCS*, 2010.

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

[15] F. Y. Chin, J. Lam, and I.-N. Chen. Efficient parallel algorithms for some graph problems. *Commun. ACM*, 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), 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*, 1996.

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

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

[20] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *FOCS*, 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*, 1995.

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

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

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

[25] S. Hambrusch and L. TeWinkel. A study of connected component labeling algorithms on the MPP. In *SC*, 1988.

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

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

[28] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Commun. ACM*, 22(8), 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), 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), 1997.

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

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

[34] V. Koubek and J. Krsnakova. Parallel algorithms for connected components in a graph. In *Fundamentals of Computation Theory*. 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*, 1994.

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

[37] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, 2012.

[38] C. E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3), 2010.

[39] 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.

[40] N. Linial and M. Saks. Low diameter graph decompositions. *Combinatorica*, 13(4), 1993.

[41] Y. Matias and U. Vishkin. On parallel hashing and integer sorting. *Journal of Algorithms*, 12(4), 1991.

[42] G. L. Miller, R. Peng, and S. C. Xu. Parallel graph decomposition using random shifts. In *SPAA*, 2013.

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

[44] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, 2013.

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

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

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

[48] C. A. Phillips. Parallel graph contraction. In *SPAA*, 1989.

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

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

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

[52] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP*, 2013.

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

[54] 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*, 2012.

[55] J. Soman, K. Kishore, and P. J. Narayanan. A fast GPU algorithm for graph connectivity. In *International Symposium on Parallel Distributed Processing*, 2010.

[56] U. Vishkin. An optimal parallel connectivity algorithm. *Discrete Applied Mathematics*, 9(2), 1984.