Parallel Computing 44 (2015) 53–68

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Evaluation of connected-component labeling algorithms for distributed-memory systems

J. Iverson [a,b,*], C. Kamath [b], G. Karypis [a]

[a] University of Minnesota, Minneapolis, MN 55455, USA
[b] Lawrence Livermore National Laboratory, Livermore, CA 94550, USA

ARTICLE INFO

Article history:
Received 29 September 2014
Received in revised form 20 January 2015
Accepted 16 February 2015
Available online 2 March 2015

Keywords:
Distributed-memory
Connected component
Scalability

ABSTRACT

Connected component labeling is a key step in a wide-range of applications, such as community detection in social networks and coherent structure identification in massively-parallel scientific simulations. There have been several distributed-memory connected component algorithms described in literature; however, little has been done regarding their scalability analysis. Theoretical and experimental results are presented for five algorithms: three that are direct implementations of previous approaches, one that is an implementation of a previous approach that is optimized to reduce communication, and one that is a novel approach based on graph contraction. Under weak scaling and for certain classes of graphs, the graph contraction algorithm scales consistently better than the four other algorithms. Furthermore, it uses significantly less memory than two of the alternative methods and is of the same order in terms of memory as the other two.

1. Introduction

Due to the increasing rate at which scientists are able to gather data, graph analysis applications often require the processing power of large, distributed-memory machines. Whether this is due to the size of the problem or the computational resources required, scientists need efficient algorithms to utilize these large machines. One fundamental algorithm used in many graph analysis applications is connected component labeling, which can be used to identify and track regions of interest throughout the life of a simulation [1–3] or locating communities in social network graphs [4]. Identifying connected components is a well understood problem on serial machines and usually employs a breadth/depth first traversal technique or utilizes the union-find data structure.

Mapping these techniques to work efficiently on distributed-memory machines is not necessarily straight-forward. As a result, there is a considerable body of research associated with developing parallel connected component labeling algorithms. This research includes development of novel algorithms that are easier to parallelize [5–11], approaches to efficiently implement connected component labeling algorithms on distributed-memory systems [3,12–16], and studies of their scalability [3,15]. However, despite this prior work, various aspects related to the computational and memory complexity of these algorithms and their scalability have not been thoroughly analyzed in a unified way. Understanding how these methods perform and scale is important due to the wide range of hardware configurations in existing and emerging distributed-memory parallel architectures.

* Corresponding author at: University of Minnesota, Minneapolis, MN 55455, USA.
E-mail addresses: jiverson@cs.umn.edu (J. Iverson), kamath@llnl.gov (C. Kamath), karpyis@cs.umn.edu (G. Karypis).

http://dx.doi.org/10.1016/j.parco.2015.02.005
0167-8191/

In this paper, five algorithms for computing connected component labeling on distributed-memory parallel systems are investigated. Of these five methods, three are direct implementations of previously published algorithms, one is an implementation based on a previous method that is optimized to reduce communication overhead, and the last is a novel method, based on repeatedly contracting the input graph at successive steps of a reduction process. Each of the algorithms is analyzed from an asymptotic perspective, providing parallel runtimes and memory requirements, assuming an equally distributed graph. Each of the algorithms is experimentally evaluated with a set of strong and weak scaling experiments, and these experimental results are compared against the expected theoretical performance. Under weak scaling and for certain classes of graphs, the graph contraction algorithm introduced in this work scales consistently better than the four other algorithms. Furthermore, it uses significantly less memory than two of the alternative methods and is of the same order in terms of memory as the other two.

The rest of the paper is organized as follows. In Section 2, definitions for terms and language used throughout the paper is provided. In Section 3, two serial algorithms for computing a connected component labeling are presented and analyzed. In Section 4, the framework for computing connected component labeling on distributed-memory parallel systems is introduced. In Section 5, five distributed-memory connected component algorithms are presented and their runtimes and memory requirements analyzed. In Section 6, the experimental design is outlined and in Section 7, the results of the experiments are presented along with a discussion of the findings. Finally, in Section 8, conclusions about this work is presented along with directions for future research regarding connected component labeling on distributed-memory systems.

2. Definitions and notation

A graph $G = (V, E)$ consists of a set of vertices V and a set of edges E. The maximum shortest path between any two connected vertices in G will be referred to as $\delta(G)$, also known as the diameter of G. Note that if $d_1, d_2, \ldots, d_n$ are the diameters of the subgraphs induced by the connected components of G, then $\delta(G) = \max(d_1, d_2, \ldots, d_n)$.

Throughout the paper, it is assumed that the connected component labeling (CCL) is being performed in a distributed-memory system made up of P processes, connected together by an interconnect network. When a message is passed between two processes, the cost of starting the communication in hardware is represented by the variable t_s and the cost of communicating a single word of data by t_w. The cost of a unit of local computation time is denoted t_c.

When computing the runtime of algorithms, T_s is the time elapsed between the beginning and the end of the algorithm execution on a serial computer. In the case of a parallel system, the elapsed time is measured from the time that the first process starts executing until the last process stops executing and is referred to as T_p. A summary of these notations is presented in Table 1.

A key element to the distributed-memory CCL algorithms is the notion of the component adjacency graph (CAG), which is used to represent the connected components identified locally by the individual processes. The component adjacency graph $G_c = (V_c, E_c)$, contains a vertex for each connected component that was identified by the various processes based on their local portion of the graph. Two vertices $u_i, u_j \in V_c$ are connected, i.e., $(u_i, u_j) \in E_c$, if there is at least one edge in G that connects a vertex in G's connected component corresponding to u_i with a vertex in G's connected component corresponding to u_j. Section 4 contains a more detailed discussion of the CAG and its construction.

3. Serial algorithms for finding connected-components

There are two algorithms for efficiently finding connected components on a serial machine. The first is based on performing a breadth-first or depth-first traversal [17]. This algorithm finds the connected components by repeatedly starting a

Table 1
Summary of notations.

Notation	Definition
t_s	Cost of starting a communication
t_w	Cost of communicating a single word over network
t_c	Cost of local computation
T_s	Serial runtime
T^c	Execution time to build CAG
T_p^c	Execution time to resolve global connectivity
T_p	Parallel runtime
T_o	Parallel overhead
G_c	Component adjacency graph (CAG)
V_c	Vertices of CAG, representing connected components of G
E_c	Edges of CAG, signifying at least one edge in G which connects a vertex from each of the corresponding components
$\delta(G)$	Max shortest path between two connected vertices in G
β	Number of adjacent processes
ω	Relationship between size of input graph and size of CAG

traversal from unvisited vertices. Any vertex reached during the traversal is marked as visited and is assigned to the same connected component as the root vertex of the traversal. The cost of the traversal based approach is linear on the size of the input, i.e., $O(|V| + |E|)$.

The second algorithm is based on the disjoint-set forest data structure and is called the union-find algorithm [18]. In this algorithm, the connected components are represented by disjoint-sets of vertices, which are encapsulated in a union-find data structure. In this data structure, each element (vertex) has a pointer to an element in its disjoint-set and exactly one element from each disjoint-set points to itself. This element with a self-pointer is designated as the root element and is used as a reference to identify the set of elements that belong to the same disjoint-set. The disjoint-set to which an element belongs can be determined using the FIND operation, which follows the pointers of the elements until it reaches an element pointing to itself, at which point the id of the representative vertex becomes the label of the disjoint-set. Two disjoint-sets can be combined using the UNION operation, which sets the pointer of the root element of one set to point to an element in the other set. To identify connected components, the union-find algorithm starts by making each vertex to belong to its own disjoint-set. Then, each of the edges of G are visited and the endpoints of each edge are unioned together. Each of these steps has an amortized cost of $O(\alpha(|V|))$ (using union-by-rank and path-compression heuristics), where $\alpha()$ is the inverse Ackermann function [18]. Since a connected component labeling requires iterating all vertices and performing a UNION of the endpoints of every edge, the complexity of this algorithm is $O(|V| + \alpha(|V|)|E|)$. In [18] it is shown that $\alpha()$ is effectively less than four for all practical input sizes. Since $\alpha()$ is constant for all potential inputs covered in this paper, the complexity can be simplified to $O(|V| + |E|)$, which is the same as algorithms based on graph traversal. This means that the cost of computing CCL in serial is

$$T_s = (|V| + |E|). \tag{1}$$

4. Distributed-memory connected-component framework

The distributed-memory connected component algorithms studied herein, consist of the following steps. First, the graph is divided equally among the processes. Next, each process identifies the connected components that exist in its local subgraph using either of the two approaches discussed in Section 3. However, because the input graph may contain components which become fragmented when the graph is distributed among the processes, another step that identifies and merges those connected components that span process boundaries, is required. This step, which is referred to as *global connectivity resolution*, is the key element of the studied distributed-memory CCL algorithms and is discussed in Section 5. The final step is to use the mapping from local component labels to globally merged labels to assign the final labeling to the vertices of the input graph. This process is summarized in Algorithm 1.

Algorithm 1. Distributed-memory CCL

1: Each process calculates its local CCL
2: Processes give local components globally unique labels
3: Generate CAG by collapsing edges that do not span process boundaries
4: Find CCL of CAG

4.1. Component adjacency graph

The key data structure used in the global connectivity resolution algorithms is the component adjacency graph (CAG), which is a graph whose vertices represent the connected components identified locally by the individual processes, see Section 2. The CAG is constructed in parallel as follows. Given a CCL of the local portion of the input graph G, each process initializes its portion of the CAG to contain a single vertex for each of its locally identified connected components.

Next, the processes assign globally unique labels to each of their connected components. This can be as simple as assigning to each connected component a label derived from the smallest vertex label contained in the connected component, see Fig. 1. If consecutive labels are required per process, then a distributed prefix sum of local connected component counts can be computed, at which point, each process will assign consecutive labels to its local connected components starting from its computed prefix sum value.

Then, the globally unique labels of the CAG vertices are propagated to the corresponding local vertices of G (see Fig. 1b). Consider the component composed of vertices 7, 8, and 12 in Fig. 1a, of which vertices 7 and 8 are in process 0 and assigned to a local component with the globally unique label of 2 in Fig. 1b, while the single vertex 12 is in process 2 and assigned to the local component with the globally unique label of 6.

Next, adjacent processes exchange the labels of their local vertices of G, at which point each process contains all the information it needs to generate the CAG edges that it owns. This is done by iterating the local edges of G. Whenever an edge crosses the process boundary, a new edge is added to the CAG between the vertices of G_c corresponding to the connected

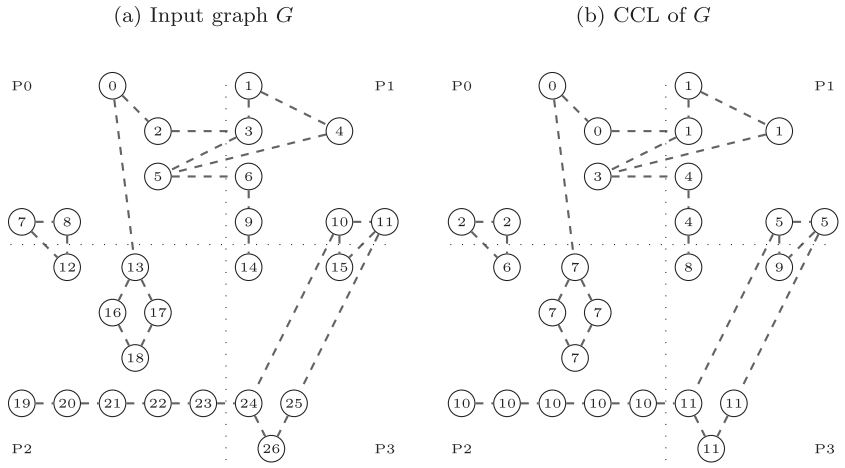

Fig. 1. An input graph, G, along with the corresponding CCL of the local graphs, labeled with globally unique labels—each process is assigned the vertices located in its quadrant.

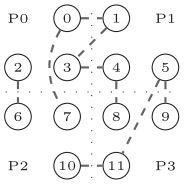

Fig. 2. The CAG, G_c, corresponding to the input graph from Fig. 1a.

components that each endpoint of the local edge belongs to. If the CAG edge already exists, it is not added again. Again, consider the component composed of the vertices labeled 2, 2, and 6 in Fig. 1b. In this component there is a single edge between the vertices labeled 2, which will be discarded since it is an edge local to process 0 and an edge between the vertex labeled 6 and each of the vertices labeled 2. Since both of the edges from the vertex labeled 6 will be represented as (2, 6), this edge need only be added once, which can be seen in Fig. 2.

Note that based on this construction, it is possible that vertices will exist within V_c, which have no outgoing edges. These vertices represent connected components which are contained completely within the boundaries of a single process, i.e., a connected component which contains no edges that cross process boundaries. Thus, the size of G_c can be substantially reduced if during construction, or as a post-processing step, these zero degree vertices of V_c are removed. In the discussion that follows it is assumed that this simplification in G_c has been done and V_c contains only vertices with non-zero degree.

The cost of constructing the CAG in parallel is $3(|V|/P + |E|/P)t_c$ for the creation of CAG nodes, propagation of globally unique labels, and creation of CAG edges. When the communication required for globally unique labels is implemented using a prefix scan, there is also a cost of $(t_s + t_w)\log P$. Finally, the exchange of labels with adjacent processes, carries a cost of $\beta t_s + (|E|/P)t_w$, where β is the number of adjacent processes. Putting these together, the parallel runtime for constructing the CAG is

$$T^c = 3\left(\frac{|V|}{P} + \frac{|E|}{P}\right)t_c + (t_s + t_w)\log P + \beta t_s + \left(\frac{|E|}{P}\right)t_w. \tag{2}$$

5. Resolving global connectivity

The critical step in the distributed-memory connected component framework described in the previous section is the method used to resolve the global connectivity. It is easy to show that the problem of resolving global connectivity among local connected components becomes the problem of finding the connected components of the component adjacency graph G_c. Given the distribution of G among the processes, each edge in G_c will connect components identified by different processes, and as such, that edge will cross process boundaries.

In the rest of this section, five distributed-memory algorithms for finding the connected components of G_c are described and their parallel runtimes and memory requirements analyzed. When computing the parallel runtime, it is assumed that the CAG is distributed evenly across the processes, i.e., each process is responsible for $|V_c|/P$ vertices and $|E_c|/P$ edges. The

memory requirements for each algorithm will focus on the amount of memory they require beyond that needed for constructing and storing the CAG, which is the same for all algorithms.

5.1. All-reduce (AR)

The first approach for resolving global connectivity is based on a distributed-memory algorithm introduced in [15]. In this algorithm, all processes communicate their edges of the CAG with every other process, then redundantly compute the connectivity resolution, requiring no further communication. Each process shares its portion of the CAG via an all-gather operation, which costs $t_s \log P + (|V_c| + |E_c|)t_w$ [19]. Once all edges have been received, each process computes locally the global connectivity resolution by performing a CCL of the CAG. The local CCL, which can be performed using either of the algorithms described in Section 3, will have a computational cost of $(|V_c| + |E_c|)t_c$, resulting in a theoretical runtime for the AR algorithm of:

$$T_p^c = t_s \log P + (|V_c| + |E_c|)t_w + (|V_c| + |E_c|)t_c. \tag{3}$$

Since each process needs to store the entire CAG, the memory complexity of AR is

$$M_p^c = O(|V_c| + |E_c|). \tag{4}$$

5.2. Union-find merging (UFM)

The second approach for resolving global connectivity is based on a parallel random-access machine (PRAM) algorithm for merging union-find data structures that was introduced in [7]. The algorithm uses a reduction tree to merge partially complete union-find data structures, that initially resides on each process. This can be done in $\log P$ steps and results in a single process storing the union-find structure corresponding to all processes.

The first step is for each process to compute its partial union-find data structure. This local data structure is of size $|V_c|$ and is computed by traversing the vertices and edges of the local portion of the CAG and unioning the endpoints of each of the edges (i.e., $\forall(u_i, u_j) \in E_c$, UNION$(u_i, u_j)$). Computing the local union-finds carries a cost of $(|V_c|/P + |E_c|/P)t_c$.

Next, a reduction tree communication pattern is used to progressively merge these local union-find data structures. At each level of the reduction tree, pairs of processes communicate by having the process with the higher rank, send its partial union-find to the process with the lower rank. The lower rank process then merges the two partial union-find structures, creating a more complete partial union-find structure. This process continues until the final level of the reduction tree, at which point, the last active process creates the complete union-find which captures all the information encoded in the partial union-finds.

The merging of two union-find data structures, UF_1 and UF_2 is performed as follows. Since each of the union-find data structures has size equal to the number of vertices in the CAG, the ith element in UF_1 corresponds to the same vertex, u_i, as the ith element of UF_2. Therefore, the disjoint-set that u_i belongs to in UF_2 can be identified by performing FIND(UF_1, i), and likewise for UF_2. Then, if there is a discrepancy between the disjoint-set to which u_i belongs, the two sets can be unioned, UNION(FIND(UF_1, i), FIND(UF_2, i)). This can be thought of as updating the first union-find with the information contained in the second. Thus, after all elements of the two union-finds have been iterated, the first union-find will contain all the merges which it had previously contained, as well as those of the second union-find. Since merging two union-finds is based on iterating over their elements, the cost of this procedure is $|V_c|t_c$. The process of merging union-find data structures is illustrated on an example in Fig. 3 with four processes using the CAG from Fig. 1a. In each of the three steps, the state of the parent pointers in the union-find data structure of each active process is shown using a different line type. In Step 1, each process has a view based only upon the outgoing edges of its vertices in the CAG. In Step 2 P_0 and P_1 exchange as do P_2 and P_3, resulting in the states seen in Step 2. Then P_0 and P_2 exchange their states from Step 2, giving the final result which can be seen in Step 3 as the complete union-find.

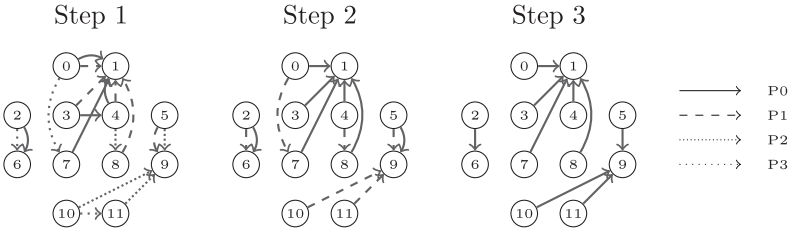

Fig. 3. Step-by-step walk through of the UFM algorithm. In the UFM algorithm, each active process has its own local view of the union-find data structure. This is represented in each of the three steps by assigning a unique line type to the pointers belonging to each process' local view. The legend indicates which line type is assigned to each process.

To reduce the amount of local work and data exchanged during each of the $\log P$ steps, the process sending its partial union-find to its partner, sends only those entries which do not point to themselves. Recall that each entry in the union-find represents a connected component which was identified in some process' local portion of G. In this case, the entries which do not point to themselves, identify those components identified in local portions of G, which belong to the same global connected component in G.

Assuming that the CAG is evenly distributed among the processes, one can expect that the amount of data being communicated will double in each of the $\log P$ steps of the reduction tree. Since the communication performed in the first step is $|V_c|/P$, the aggregate amount of data being communicated over the $\log P$ steps to is $\sum_{i=0}^{\log(P)} 2^i |V_c|/P = |V_c|$. Since the receiving process receives only those entries which have been updated in previous steps, the aggregate amount of local work required to merge the union-find structures is also reduced. In fact, with this optimization, merging the union-find structures requires only that the receiving process iterate through those entries which it received, meaning that the aggregate amount of local work, over all $\log P$ steps, is also $|V_c|$.

Once all partial union-finds have been merged into a complete union-find located on a single process, the complete union-find is broadcast back to the remaining processes. The cost of this operation is $2(t_s \log P + |V_c| t_w)$, using the broadcast algorithm which performs a scatter followed by an all-gather, which is described in [19]. This cost, along with the previously discussed costs, give a parallel runtime for the UFM algorithm of:

$$T_p^c = \left(\frac{|V_c|}{P} + \frac{|E_c|}{P}\right) t_c + 3(t_s \log P + |V_c| t_w) + |V_c| t_c. \tag{5}$$

Since each process needs to store a complete union-find data structure, the memory requirement of UFM is

$$M_p^c = O(|V_c|). \tag{6}$$

5.3. Graph contraction (GC)

The GC algorithm takes the idea of distributed union-find data structures one step further. This algorithm resembles the UFM algorithm in that it also uses a reduction tree. However, to reduce the runtime of the algorithm, the GC algorithm only communicates the vertices whose connected components have not yet been completely resolved. As a result, the amount of data being communicated between processes reduces in the course of the successive communication steps.

The GC algorithm is able to identify the connected components that have been resolved by repeatedly modifying the CAG at each reduction step. Specifically, at each level of the reduction tree, the active processes are split into pairs and each process communicates its local portion of the current CAG to its partner. Let $G_{c,i} = (V_{c,i} \cup V_{c,j}, E_{c,i} \cup E_{c,j})$ be the portion of the CAG stored at process P_i, that contains both its local portion of the CAG and the portion which it just received from process P_j. The GC algorithm then identifies the connected components that exist in $G_{c,i}$ and generates its new local portion of the CAG, $G^*_{c,i}$, by collapsing all the vertices in $V_{c,i} \cup V_{c,j}$ that belong to the same connected components. During this process, $E^*_{c,i}$ is updated accordingly and any vertices that have zero degree are removed from $V^*_{c,i}$. This process is shown in Fig. 4, where the edges being collapsed are represented by solid lines.

If one assumes that half of the vertices and their associated edges in $G_{c,i}$ are eliminated, then the size of each active processes' portion of the CAG will not increase in successive steps. Therefore, at each step, the time required for data communication by an active process is $(|V_c|/P + |E_c|/P)t_w$. Under the same assumptions, at each step, the cost of merging two CAGs is $(|V_c|/P + |E_c|/P)t_c$, as it can be done in time linear to the number of vertices/edges in the graphs involved. Thus, the overall parallel runtime of GC is

$$T_p^c = \left(t_s + \left(\frac{|V_c|}{P} + \frac{|E_c|}{P}\right) t_w + \left(\frac{|V_c|}{P} + \frac{|E_c|}{P}\right) t_c\right) \log P. \tag{7}$$

Note that if the above assumption does not hold, then the communication and computation cost will be higher and in the worst case will be equivalent to that of an all-to-one gather of the graph.

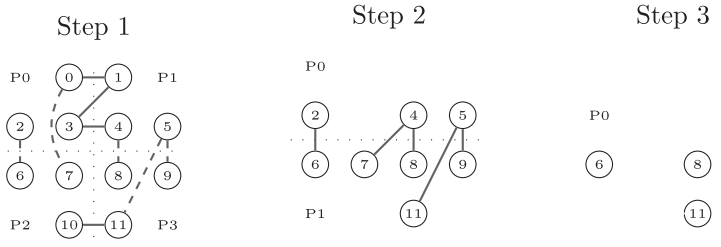

Fig. 4. Step-by-step walk through of the GC algorithm. The dotted lines $(\cdots)$ in the figure above represent the process boundaries. In the GC algorithm, the process boundaries change in each step, signifying the transfer of a local portion of the CAG from one process to another.

Since a receiving process is required to store its local portion of the CAG as well the portion it receives, it must reserve memory for both. Storage for the local portion is the same as the storage required for the portion being received, making the memory requirement for the GC algorithm:

$$M_p^c = O\left(\frac{|V_c|}{P} + \frac{|E_c|}{P}\right). \tag{8}$$

5.4. Label propagation (LP)

The LP algorithm diverges from the approaches described thus far as it repeatedly updates the label of each vertex in the CAG until the labels converge. This approach consists of a series of iterations. The iterations start with each process assigning a label matching the vertex id to each of its vertices in V_c, so $l(u_i) = i$. Then, for each edge in the CAG, the labels of the adjacent vertices are exchanged. Each process then updates its local labels by assigning to each vertex the minimum of its label and the label of its adjacent vertices. The algorithm terminates when no labels have changed in a single iteration.

Each iteration requires every process to communicate its vertices with adjacent processes and then to iterate through each of its vertices and edges to see if a given vertex-label should be updated, incurring a cost of $t_s + (|E_c|/P)t_w + (|V_c|/P + |E_c|/P)t_c$. To detect that no changes have occurred, an all-to-all reduction (sum) on a count of each of the processes local changes is performed. When the result of the reduction is 0, the algorithm is finished. This operation carries a cost of $(t_s + t_w)\log P$. If one assumes that there are $\delta(G_c)$ relabeling iterations before convergence, then the asymptotic runtime is

$$T_p^c = \left(t_s + \frac{|E_c|}{P}t_w + \left(\frac{|V_c|}{P} + \frac{|E_c|}{P}\right)t_c + (t_s + t_w)\log P\right)\delta(G_c). \tag{9}$$

Since the LP algorithm is based on repeatedly updating the labels of the CAG vertices, it requires storage for the labels of adjacent vertices, of which there are $O(|E_c|/P)$. Thus, its memory requirement is

$$M_p^c = O\left(\frac{|E_c|}{P}\right). \tag{10}$$

5.5. Distributed union-find (DUF)

The DUF algorithm, which is a modification of the concurrent read concurrent write PRAM algorithm of [6], is similar to the LP algorithm. It however, updates edges during each relabel iteration in order to reduce the number of iterations required for label convergence.

Once the CAG has been constructed, each process assigns its local vertices of G_c a parent pointer, which initially points to themselves. In Figs. 5, 6, any vertex without an outgoing parent pointer is assumed to have a parent pointer to itself. These

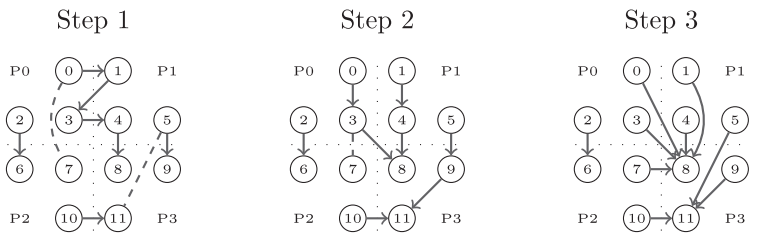

Fig. 5. Step-by-step walk through of the marking parent-pointer phase of the DUF algorithm.

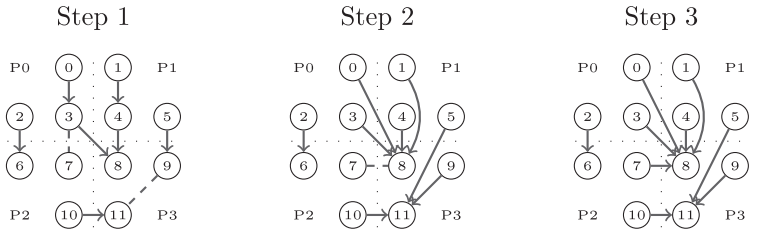

Fig. 6. Step-by-step walk through of the short-cutting phase of the DUF algorithm, assuming the corresponding parent-pointers at each step of Fig. 5. The dotted lines (⋯) in Figs. 5 and 6 above represent the process boundaries. In the DUF algorithm, the process boundaries do not change, as each process retains its local portion of the CAG throughout and only changes the parent pointers associated with each of its vertices.

were left out however to avoid congestion in the graph. Then, each process iterates through its local vertices of the CAG and updates the parent pointer of each vertex to point to the first adjacent vertex which has an id higher than itself. This is demonstrated in the first step of Fig. 5, which shows the original edges left as dashed edges and the parent pointers as solid lines with an arrow pointing toward the parent. This constraint maintains acyclic relationships among parent pointers. Updating parent pointers requires that each process communicates, with its adjacent processes, the parent pointer of each of the shared endpoints. This whole operation will incur a cost of $t_s + (|E_c|/P)t_w + (|V_c|/P + |E_c|/P)t_c$.

The next step is to shortcut the parent pointers. This is achieved by updating each vertex' parent pointer to point to its grandparent. This is visualized in Fig. 6, which shows the short-cut counter-part for each step of Fig. 5. This requires the processes to again share the parent pointer information of their vertices and to again iterate through the vertices and edges, possibly updating parent pointers along the way, costing $t_s + (|E_c|/P)t_w + (|V_c|/P|E_c|/P)t_c$. Finally, each process must pass each of its edges to its parent. Fig. 6 also shows the result of each vertex giving ownership of its remaining input edges (dashed) to its parent. For example it can be seen that edge (5, 11) in Step 1 of Fig. 5 becomes edges (9, 11) in Fig. 6. It is possible that a sequence of steps can occur when no new parent pointers are introduced, but only existing pointers are updated and input edges moved to new parents.

These two steps are repeated until no parent pointers can be updated, in which case each connected component in the CAG is be represented by a star graph, which can be seen in the last step of Fig. 6. The same termination detection algorithm as described in Section 5.4 can be used here, costing $(t_s + t_w)\log P$. At this point, the labels of each component can be assigned simply as the id of the root vertex of each star. Since the constraint mentioned above prohibits forming cycles of parent pointers, the maximum number of iterations of this algorithm will be equal to $\log(\delta(G_c))$, since when pointers are updated the second time during each iteration, the distance each node is from the root is halved. Putting this all together parallel runtime can be stated as:

$$T_p^c = \left(2(t_s + \frac{|E_c|}{P}t_w + \left(\frac{|V_c|}{P} + \frac{|E_c|}{P}\right)t_c) + (t_s + t_w)\log P\right)\log(\delta(G_c)). \tag{11}$$

To keep track of the parent pointers for the CAG vertices, each process in the DUF algorithm must maintain an array of parent pointers, incurring a storage cost of $O(|V_c/P|)$. During the execution of the algorithm, edges are transferred to the process of the parent pointer. A given process is equally likely to transfer edges out of its local view as it is to receive edges because the parent pointer of a remote vertex exists in its local view. Thus, during successive steps of the DUF algorithm, the number of edges which exist on a given process does not increase significantly, meaning that no extra storage is required for the process' new view of the CAG. Thus, the memory requirements for the DUF algorithm can be expressed as:

$$M_p^c = O\left(\frac{|V_c|}{P}\right). \tag{12}$$

5.6. Scalability analysis

In Section 5.1–5.5, all parallel runtimes for resolving global connectivity, T_p^c, were computed in relation to the average runtime, assuming that the CAG graph was distributed evenly across processes. These expressions are summarized in Table 2.

To understand the theoretical scalability of these algorithms, their isoefficiency functions [19] were computed. This was done using the same set of assumptions as those used to derive the runtime expressions (Table 2). The isoefficiency function, which defines the rate at which the problem size must grow in-order to achieve a fixed efficiency as the number of processes is increased, has been shown to be an effective method for analyzing and comparing the scalability of different algorithms [19]. These isoefficiency functions are shown in Table 3. In deriving these expressions, it is assumed that the size of the CAG is related to the size of the original graph G via the expression $|V_c| = |V|^{\omega}$ for $0 \leqslant \omega < 1$.

To illustrate how these expressions were derived, the steps required for the derivation of the isoefficiency function for the AR algorithm is presented below. The derivation of the other isoefficiency functions is similar.

Table 2
Parallel runtimes for resolving global connectivity.

Algorithm	T_p^c
AR	$t_s \log P + (\|V_c\| + \|E_c\|)t_w + (\|V_c\| + \|E_c\|)t_c$
UFM	$\left(\frac{\|V_c\|}{P} + \frac{\|E_c\|}{P}\right)t_c + 3(t_s \log P + \|V_c\|t_w) + \|V_c\|t_c$
GC	$\left(t_s + \left(\frac{\|V_c\|}{P} + \frac{\|E_c\|}{P}\right)t_w + \left(\frac{\|V_c\|}{P} + \frac{\|E_c\|}{P}\right)t_c\right)\log P$
LP	$\left(t_s + \frac{\|E_c\|}{P}t_w + \left(\frac{\|V_c\|}{P} + \frac{\|E_c\|}{P}\right)t_c + (t_s + t_w)\log P\right)\delta(G_c)$
DUF	$\left(2(t_s + \frac{\|E_c\|}{P}t_w + \left(\frac{\|V_c\|}{P} + \frac{\|E_c\|}{P}\right)t_c) + (t_s + t_w)\log P\right)\log(\delta(G_c))$

Table 3
Isoefficiency functions, where $|G_c| = |G|^{\omega}$ with $0 \leqslant \omega < 1$.

Algorithm	Isoefficiency
AR	$\max\left(P^2, P^{(1/(1-\omega))}\right)$
UFM	$\max\left(P^2, P^{(1/(1-\omega))}\right)$
GC	$\max\left(P^2, \log P\right)^{(1/(1-\omega))}$
LP	$\max\left(P^2, \delta(G_c)^{(1/(1-\omega))}\right)$
DUF	$\max\left(P^2, \log\left(\delta(G_c)\right)^{(1/(1-\omega))}\right)$

Table 4
Information about datasets d1, d2, and d3, each with three variants. $|V_c|$ and $\delta(G_c)$ are listed for 2–32 processes.

	d1	d2	d3
$\|V\|$	16.7 M	41.4 M	100.6 M
$\|E\|$	50.1 M	122.0 M	185.5 M
# CC	5.4 K	13.6 K	5.4 K
$\|V_c\|$	695, 851, 1.1 K, 1.8 K, 2.7 K	8 K, 21 K, 46 K, 94 K, 188 K	10 K, 30 K, 76 K, 189 K, 466 K
$\delta(G_c)$	3, 5, 8, 9, 13	8, 14, 28, 60, 112	9, 17, 38, 82, 162
# CC	280.5 K	683.9 K	438.7 K
$\|V_c\|$	11 K, 18 K, 27 K, 39 K, 51 K	25 K, 68 K, 163 K, 327 K, 610 K	112 K, 321 K, 723 K, 1.5 M, 3.2 M
$\delta(G_c)$	3, 11, 16, 24, 28	6, 13, 20, 41, 79	12, 22, 46, 85, 171
# CC	4.1 M	2.8 M	2.9 M
$\|V_c\|$	74 K, 112 K, 142 K, 194 K, 238 K	21 K, 59 K, 147 K, 309 K, 590 K	218 K, 634 K, 1.4 M, 2.8 M, 5.6 M
$\delta(G_c)$	4, 10, 14, 22, 24	5, 11, 24, 47, 90	10, 14, 31, 50, 96

Table 5
Information about datasets d4 and d5, each with three variants. $|V_c|$ and $\delta(G_c)$ are listed for 2–32 processes.

	d4	d5
$\|V\|$	4.8 M	18.5 M
$\|E\|$	85.7 M	523.6 M
# CC	11.8 K	1.1 M
$\|V_c\|$	715 K, 1.2 M, 1.6 M, 1.9 M, 2.2 M	80 K, 129 K, 157 K, 175 K, 191 K
$\delta(G_c)$	6, 6, 6, 6, 6	9, 9, 9, 9, 9
# CC	53.8 K	3.0 M
$\|V_c\|$	711 K, 1.2 M, 1.6 M, 1.9 M, 2.1 M	83 K, 140 K, 171 K, 194 K, 209 K
$\delta(G_c)$	7, 8, 8, 8, 8	11, 12, 12, 13, 15
# CC	110.3 K	4.3 M
$\|V_c\|$	697 K, 1.2 M, 1.6 M, 1.9 M, 2.1 M	89 K, 149 K, 182 K, 206 K, 217 K
$\delta(G_c)$	7, 7, 8, 8, 8	10, 10, 13, 14, 16

Derivation of AR isoefficiency. Given that the parallel runtime of an algorithm is $T_p = T^c + T^c_p$, and that the parallel overhead of an algorithm is $T_o = PT_p - T_s$, the parallel overhead of the AR algorithm can be simplified as follows:

$$T_o = N(t_c + t_w) + P\log P(t_s + t_w) + P\beta t_s + N^{\omega}(t_w + t_c). \tag{13}$$

In the equation, $N = (|V| + |E|)$ and $N^{\omega} = (|V_c| + |E_c|)$. Then to find the isoefficiency of AR, the rate, in terms of P, at which N must be increased in order to maintain a constant efficiency is found by examining each term in T_o.

$$N \sim N(t_c + t_w) \rightarrow N \sim 1, \tag{14a}$$

$$N \sim P\log P(t_s + t_w) \rightarrow N \sim P\log P, \tag{14b}$$

$$N \sim P\beta t_s \rightarrow N \sim P\beta, \tag{14c}$$

$$N \sim N^{\omega}(t_w + t_c) \rightarrow N \sim P^{1/1-\omega}. \tag{14d}$$

Since $\beta \leqslant P$, Eq. 14c can be at most P^2, making the asymptotically highest rate of increase of N to maintain constant efficiency, the maximum of Eqs. 14c and 14d Thus, $\max(P^2, P^{1/1-\omega})$ is the isoefficiency function for the AR algorithm. In our experiments, the value of ω ranged from 0.39 up to 0.95 and had a mean value of 0.70. Given that $\omega = 0.70$, then the isoefficiency function for the AR algorithm becomes $P^{3.\bar{3}}$, and likewise for the rest of the isoefficiency functions.

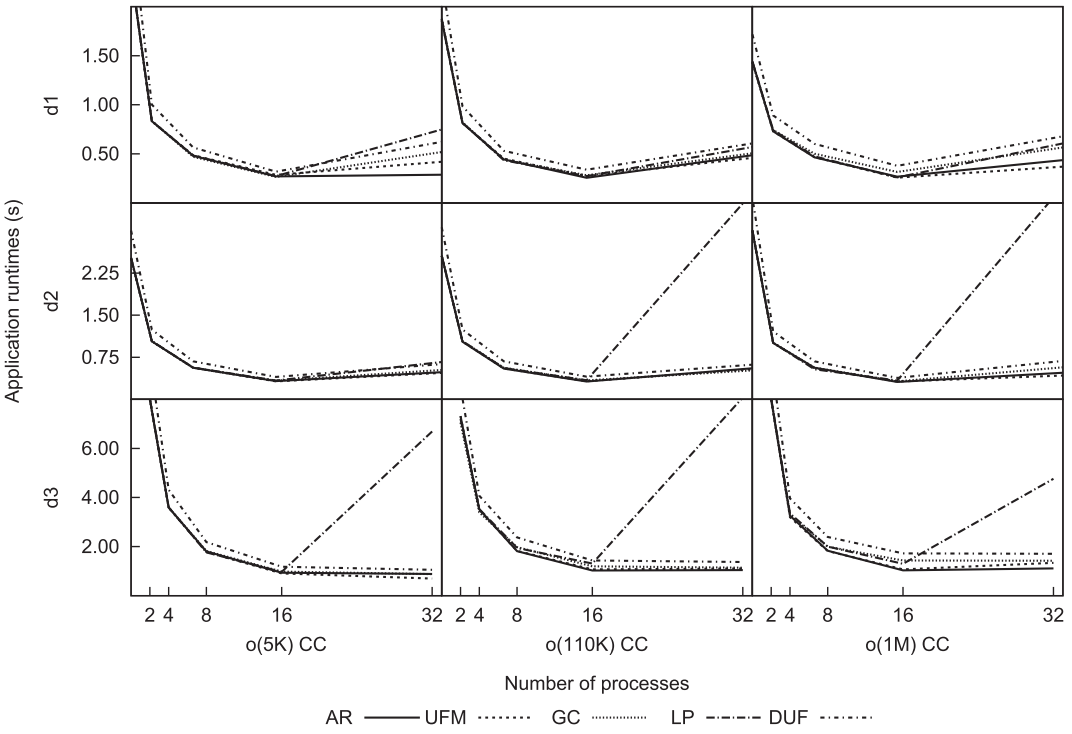

Fig. 7. Strong scaling parameter study for data sets d1, d2, and d3.

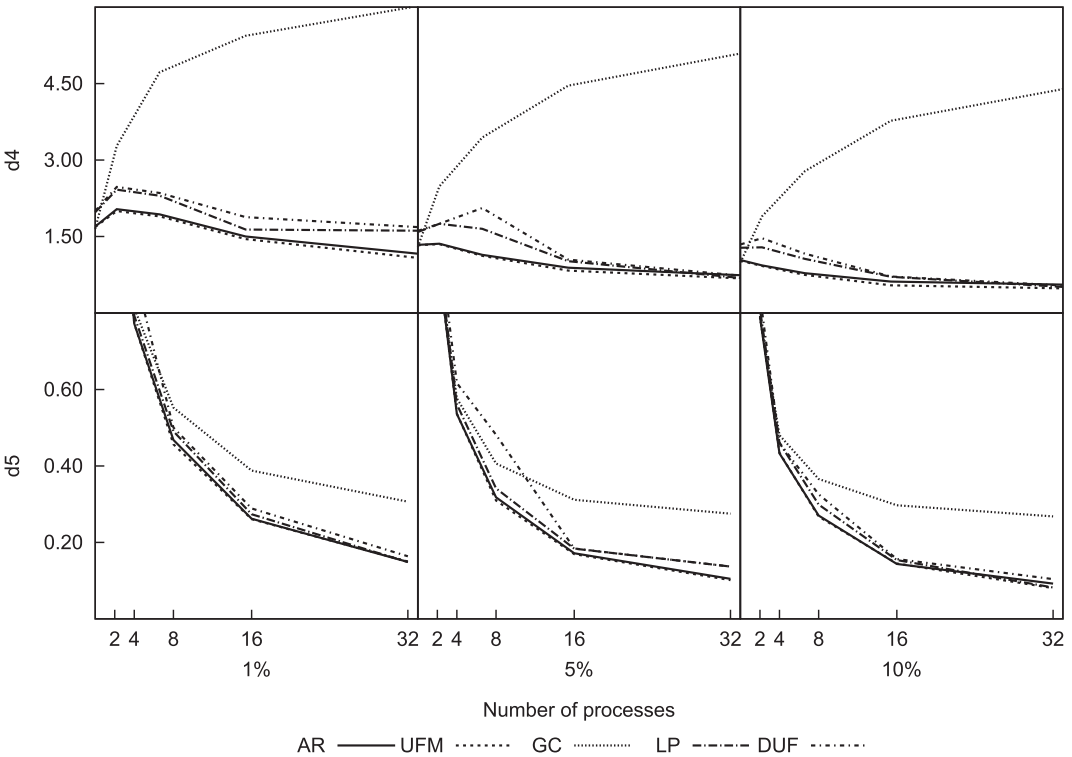

Fig. 8. Strong scaling parameter study for data sets d4 and d5.

6. Experimental evaluation

6.1. Datasets

To explore the performance characteristics of these algorithms, five real world datasets obtained from UMN, NASA, SNL, Stanford, and Rutgers were used. These first three of these datasets correspond to either the nodal or dual graphs of finite element meshes used for a range of problems in the computational fluid dynamics domain. The last two represent network graphs, specifically the link structure of LiveJournal, a social networking site, and the web graph of the United Kingdom.

For the finite element meshes, the problem of coherent structure identification as the method for creating connected components was used. The size and number of connected components is controlled by a parameter ϵ, which specifies the maximum allowed difference between the values associated with the nodes belonging to the same connected component [20]. For the network graphs, the problem was simply to identify connected regions of the network. To evaluate the performance of the CCL algorithms under different scenarios, for each dataset, three different graphs that have progressively larger number of connected components were generated. For the finite element graphs, this was done by decreasing the value of ϵ and for the network graphs, this was achieved by removing increasing percentages of edges from the highest degree

Table 6
CAG size for each GC reduction step of 32 process experiment.

Step	d3		d5	
	$\|V_c\|$	$\|E_c\|$	$\|V_c\|$	$\|E_c\|$
1	466,152	1,078,202	219,974	869,404
2	189,853	485,311	206,430	785,464
3	76,842	183,721	183,463	601,048
4	31,373	65,047	150,109	401,014
5	10,741	19,688	91,152	198,835

Table 7
Aggregate strong scaling runtimes (s) for all datasets. Results are presented for the five algorithms for 2–32 processes.

	P				
	2	4	8	16	32
d1					
AR	5.65	2.37	1.37	0.79	1.22
UFM	5.64	2.37	1.40	0.80	1.19
GC	5.66	2.42	1.42	0.95	1.57
LP	5.67	2.39	1.42	0.80	1.81
DUF	5.62	2.37	1.46	0.80	1.75
d2					
AR	8.02	3.07	1.65	0.95	1.45
UFM	8.05	3.08	1.75	0.95	1.87
GC	8.04	3.09	1.66	1.14	1.27
LP	8.10	3.09	1.65	1.13	7.98
DUF	8.06	3.08	1.66	1.04	1.55
d3					
AR	23.33	10.37	5.45	2.99	3.03
UFM	23.37	10.29	5.43	3.05	3.13
GC	23.00	10.29	5.79	3.61	3.44
LP	23.23	10.47	5.70	3.57	19.53
DUF	23.47	10.72	5.88	3.75	4.35
d4					
AR	4.06	4.33	3.85	3.01	2.52
UFM	4.06	5.43	3.73	2.86	2.14
GC	4.87	5.45	5.60	3.36	3.53
LP	3.93	8.63	11.40	15.20	16.50
DUF	4.90	5.55	5.81	3.42	3.56
d5					
AR	3.05	1.74	1.05	0.60	0.42
UFM	3.05	1.74	1.03	0.57	0.33
GC	3.12	1.84	1.13	0.61	0.36
LP	3.10	1.87	1.83	0.99	0.85
DUF	3.14	1.85	1.17	0.62	0.38

vertices. Various statistics about the original datasets and the three graphs that were generated for each of them are shown in Tables 4, 5.

6.2. Experimental design

The performance evaluation is composed of three studies. The first is a strong scaling study in which the number of processes was increased for a fixed problem size. The second is a weak scaling study, in which the problem size per process remained fixed as the number of processes increased. The last is an input permutation study, where the vertices of the input graph are renumbered to change the initial distribution of the input graph. Three input permutations are studied. The first is no permutation, i.e., the data in the order in which it was generated, the second is a random permutation of the vertices, and the last is a renumbering of the vertices based on the partitions identified by the graph partition software Metis [21], where the graph is partitioned into a number of parts equal to the number of processes, then the vertices of each part are given consecutive vertex ids.

To perform the weak scaling performance studies, the datasets used for the strong scaling studies were up-sampled. In this case, only the finite element method datasets were used, as at the time of writing, the authors are unaware of an intuitive technique for up-sampling the network graphs. To up-sample the finite element datasets, triangles/squares in the tetrahedral and cube meshes, respectively, were located, and a new point placed inside the shape, with new edges connecting the new point to each of the existing points. The purpose of this up-sampling technique is to keep similar characteristics between the input graph and the up-sampled graph, i.e., number of components and communication patterns.

All experiments were run on the Minnesota Supercomputing Institutes *Itasca* machine, which is an HP Linux cluster with 1091 HP ProLiant BL280c G6 blade servers, each with two-socket, quad-core 2.8 Ghz Intel Xeon X5560 processors sharing 24 GB of system memory [22]. The system is connected with a 40-gigabit QDR InfiniBand interconnect. All algorithms were implemented by the authors in C++ using MPI for message passing between processes. Timing of the algorithms was done using the MPI_Wtime() function and the particular implementation of MPI used for testing was OpenMPI v1.5 with g++ v4.7.0. For all experiments, each application was run 20 times and the geometric mean of these execution times is reported herein. For any given set of experimental run, the minimum and maximum execution times were within 15% of each other, thus, detailed runtime error statistics have been omitted.

Table 8
Aggregate memory usage (MB) for all datasets. Results are presented for the five algorithms for 2–32 processes.

	P				
	2	4	8	16	32
d1					
AR	1.3	2.1	2.8	4.0	5.0
UFM	1.3	2.1	2.7	3.7	4.6
GC	1.5	2.5	3.4	4.2	4.7
LP	0.3	0.4	0.3	0.2	0.1
DUF	1.3	2.5	3.2	4.1	4.7
d2					
AR	0.9	2.4	5.9	12.5	24.3
UFM	0.8	2.3	5.7	11.7	22.2
GC	1.0	2.2	3.4	4.5	5.9
LP	0.2	0.4	0.4	0.4	0.4
DUF	1.0	2.2	3.2	4.5	5.6
d3					
AR	47.7	64.1	70.1	112.1	202.6
UFM	47.7	64.1	67.5	99.9	173.1
GC	47.7	64.1	59.6	60.5	67.6
LP	47.7	64.1	59.6	57.9	57.9
DUF	47.7	64.1	59.6	61.3	67.8
d4					
AR	62.8	107.0	150.9	196.3	240.6
UFM	50.4	76.9	95.5	109.5	120.1
GC	100.0	168.9	222.8	267.8	302.3
LP	12.6	12.9	11.0	8.7	6.7
DUF	70.3	110.4	156.7	203.2	251.3
d5					
AR	5.1	9.0	12.1	15.3	18.2
UFM	4.1	6.7	8.1	9.2	9.9
GC	7.5	13.1	17.2	20.5	22.6
LP	1.0	1.0	0.8	0.6	0.4
DUF	6.5	11.6	13.4	18.3	20.2

7. Results

7.1. Strong scaling

Fig. 7 shows the results of the strong-scaling study on 2–32 processes for datasets d1, d2, and d3. The sub-figures along each row correspond to results obtained for the same dataset but for different values of ϵ. In all the rows, the number of connected components increases from left-to-right and ranges from a few thousands to a few million connected components.

In terms of application runtime, all algorithms exhibit very similar scaling behavior up to 16 processes. However, at 32 processes, LP performs considerably worse than the rest, and its overall runtime increases considerably. The poor performance of LP on 32 processes can be attributed to the facts that (i) the number of iterations that it requires in order to terminate increases (as it depends on the value of $\delta(G_c)$ in Table 4 and (ii) that each of these iterations needs to communicate more data as the number of processes increases.

Fig. 8 shows the results of the strong-scaling study on 2–32 processes for datasets d4 and d5. The format of these results are the same as the results presented for the first three datasets. For both datasets, AR, UFM, and DUF, exhibit scaling behavior similar to their corresponding results in Fig. 7. However, LP exhibits much better scaling behavior than its corresponding results for the CFD datasets. This is expected due to nature of the graph structures of datasets d4 and d5, both of which require a small number of iterations to propagate labels throughout the graph, as can be seen in Table 4. Finally, this figure shows the relatively poor performance of the GC algorithm on these datasets. Just as the superior performance of LP was attributed to the structure of the two datasets, the inferior performance of GC can also be attributed to the structure of these two datasets. Contrary to the CFD datasets, datasets d4 and d5 exhibit much less of a regular structure; regular in the sense that adjacent vertices in the graph are adjacent to similar sets of vertices. This lack of regular structure, makes the reduction steps of the GC algorithm expensive, since the merging of two CAGs does not lead to a large reduction in the size of the resulting CAG. This is evidenced by Table 6, which shows the size of the CAG, aggregated across all processes, for each of the reduction steps using 32 processes on datasets d3 versus d5.

In addition, Tables 7 and 8 show the amount of time and memory required by the different algorithms for the all five datasets. These statistics were obtained by aggregating the time/memory required by all three graphs derived from that dataset (i.e., aggregation of the runs corresponding to the bottom row of Fig. 7). Comparing the performance of the five schemes, AR has the lowest aggregate runtime, with GC and UFM following right behind. However, the relative performance of the algorithms is considerably different, when their memory footprint is taken into account. From Table 8, LP, GC, and DUF require the least amount of memory, whereas the other two algorithms require significantly more memory for datasets d1,

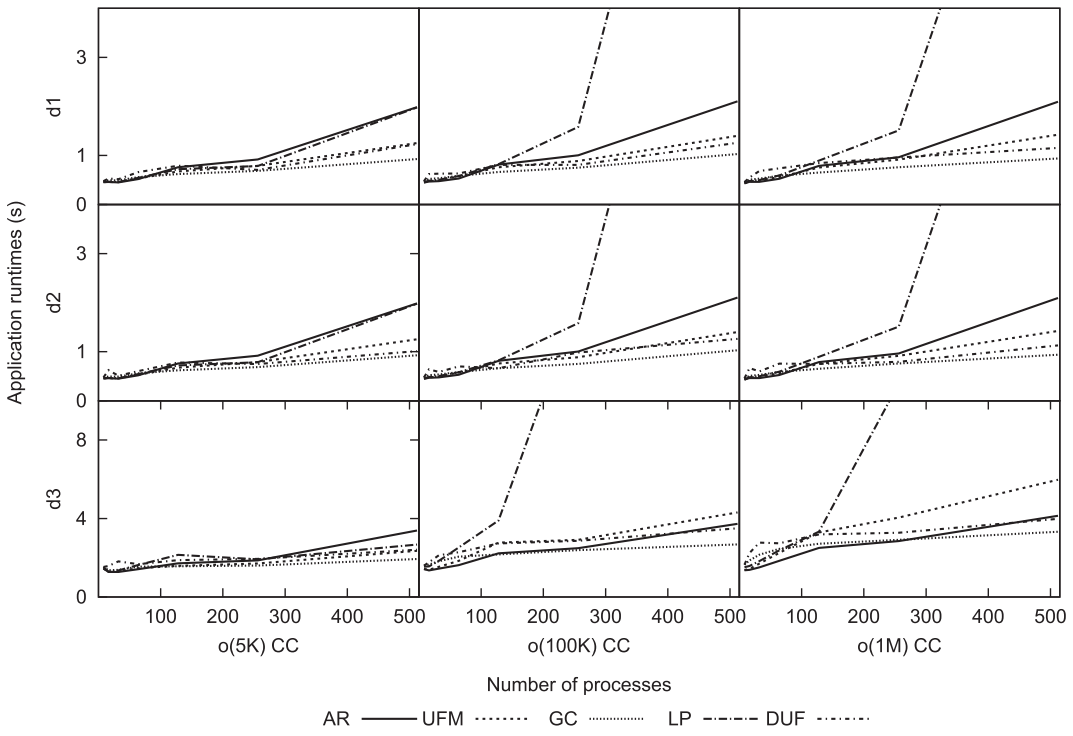

Fig. 9. Weak scaling parameter study for data sets d1, d2, and d3.

d2, and d3. The high memory requirements of AR and UFM are due to the fact that they require that the entire information associated with their underlying algorithms (CAG or the entire disjoint-set data structure) is aggregated to a single processor. Also, the low memory requirements of GC indicate that during the successive graph contraction operations, the size of the contracted graph decreases substantially, and it does not lead to any significant increases in its memory footprint. The same is not true for datasets d4 and d5, where GC's inability to successfully contract at each step causes it to have a much higher memory footprint. Furthermore, due to LP's simple exchange of vertex labels, it is able to maintain a low memory footprint for all datasets.

7.2. Weak scaling

Fig. 9 and Table 9 show the results of the weak-scaling parameter study on 8–512 processes. The per process problem size was selected to be the original problem size divided by eight, formally $|E|/8$. The format of these results are similar to the corresponding results presented for the strong-scaling study in the previous section.

These results show that none of the algorithms scale linearly as the number of processors is increased and the per process problem size is fixed. This should not be a surprise, as the parallel runtime complexity analysis presented for the various algorithms indicate that they have an isoefficiency function that is at least $O\left((\log P)^{(1/(1-a))}\right)$, see Table 3. However, these results provide experimental evidence about the real-life scalability of the various algorithms and how their performance is affected by the nature of the underlying problem (i.e., the number and size of the connected components). Specifically, these experiments show that among all five algorithms, GC scales consistently better than the rest, both for graphs with

Table 9
Aggregate weak-scaling runtimes (s) and memory usage (MB) for dataset d1, d2, and d3. Results are presented for the five algorithms for 8–512 processes.

	P						
	8	16	32	64	128	256	512
d1							
Runtimes							
AR	1.32	1.38	1.43	1.58	2.32	2.79	6.16
UFM	1.34	1.38	1.44	1.71	2.24	2.62	4.05
GC	1.40	1.51	1.53	1.73	1.93	2.19	2.89
LP	1.37	1.44	1.46	1.64	2.37	3.81	26.65
DUF	1.35	1.41	1.49	1.58	2.22	2.43	3.21
Memory							
AR	5.37	4.76	4.29	4.84	5.44	5.46	5.47
UFM	5.37	4.76	4.29	4.75	5.23	5.25	5.26
GC	5.37	4.76	4.29	3.93	3.68	3.13	3.02
LP	5.37	4.76	4.29	3.93	3.68	3.11	2.99
DUF	5.37	4.76	4.29	3.93	3.68	3.14	3.04
d2							
Runtimes							
AR	1.33	1.38	1.39	1.57	2.36	2.88	6.18
UFM	1.33	1.40	1.44	1.66	2.18	2.59	4.07
GC	1.41	1.52	1.55	1.73	1.96	2.27	2.92
LP	1.35	1.42	1.46	1.71	2.43	3.88	26.90
DUF	1.38	1.39	1.42	1.63	2.15	2.45	3.31
Memory							
AR	29.45	26.88	30.19	48.40	84.93	84.95	84.98
UFM	29.45	26.88	28.73	44.45	73.25	73.25	73.26
GC	29.45	26.88	25.58	24.96	23.20	12.01	10.10
LP	29.45	26.88	25.58	24.96	23.20	11.73	5.90
DUF	29.45	26.88	25.58	24.96	23.20	11.89	10.14
d3							
Runtimes							
AR	0.37	0.40	0.52	0.79	1.68	2.47	6.25
UFM	0.41	0.56	0.86	1.58	3.65	3.08	5.30
GC	0.84	1.20	1.76	2.46	2.99	3.71	4.69
LP	0.64	0.76	1.33	2.63	5.44	24.40	81.90
DUF	0.73	1.01	1.42	2.48	3.24	3.84	4.98
Memory							
AR	70.1	112.1	202.6	395.7	603.1	603.0	607.0
UFM	67.5	99.9	173.1	323.7	482.8	481.4	482.9
GC	59.6	60.5	67.6	79.0	83.0	79.6	77.9
LP	59.6	57.9	57.9	58.3	58.3	59.5	57.6
DUF	59.6	61.2	68.3	79.7	84.2	80.0	77.9

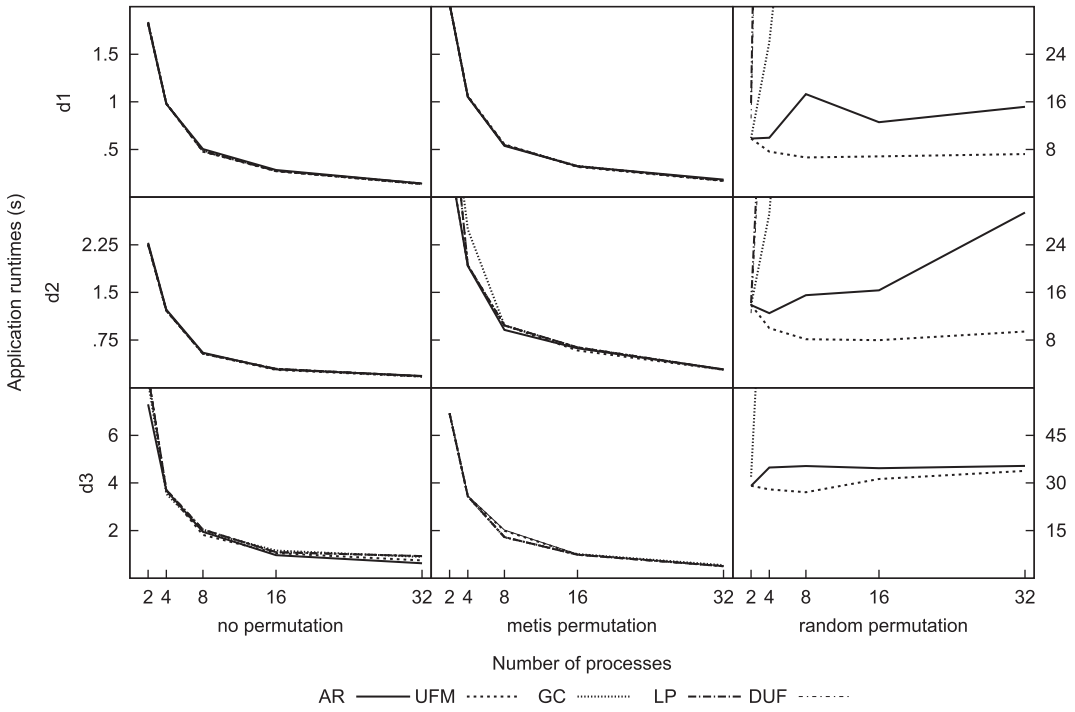

Fig. 10. Input graph permutation study.

few as well as many connected components. The performance of the other methods is more variable and it tends to get worse as the number of connected components increases. This is especially true for the LP and AR algorithms. In terms of overall execution time, Table 9 shows that GC requires less time than the rest, and that as the number of processes increases, the time required by LP increases considerably. Also, the memory requirements of the different methods are similar to those observed in the strong scaling study, with GC, LP, and DUF requiring significantly less memory than the other two methods. Moreover, the memory requirements of GC, LP, and DUF remain roughly constant as the number of processes increases.

7.3. Input graph permutation

Fig. 10 shows the results of the input permutation study on 2–32 processes. Like the previous studies, each row of the figure represents the results for a given dataset. In this case, the columns represent the different input permutations.

Firstly, these results show that for the datasets from the computational fluid dynamics domain, the original input permutation is quite close to the Metis permutation. This should be expected, since the programs that run this type of simulation often rely on tools like Metis to determine vertex distribution in a parallel setting. Secondly, these experiments highlight the importance of input permutation. Specifically, none of the algorithms performed well on a random permutation. This is due to the increase in communication associated with having adjacent vertices distributed across processes.

The superior performance of the AR and UFM algorithms can be explained by their reliance on global communication primitives. Both of these algorithms rely on gathering a global data structure to a single process, which means that the amount of data being communicated in the course of the algorithms is independent of the data distribution. The communication time can still be affected by data distribution, since different distribution will put different pressures on the network, but overall, the cost of these algorithms in terms of communication is independent of input permutation. In these experiments, the trade-off for the increased memory to gather the data structure is acceptable, since the data structures fit into the memory of a single process. However, as can be seen from Table 8, the memory footprint of the AR and UFM algorithms is significantly higher than the others. Thus, for larger problem sizes, the memory requirement of these two algorithms will be prohibitively high.

8. Conclusion

In this paper, five algorithms were explored for computing the connected component labeling of a graph that is distributed across nodes of a distributed-memory system. Theoretical complexity for each algorithm was derived under the assumption that the intermediate component adjacency graph is evenly distributed. Experimental results were presented for each method on a set of graphs arising from coherent structure identification in scientific computing and community

detection in social network analysis. The experimental results showed that under weak scaling the graph contraction algorithm, introduced in this work, scales more consistently than any of the other algorithms and uses approximately the same amount of memory as the most memory efficient of the algorithms. Since in many applications, connected component labeling is a step that needs to be performed in parallel with other types of computations, being able to perform this operation with minimal additional memory overheads, is an important requirement.

Acknowledgments

This work was supported in part by NSF – United States (IIS-0905220, OCI-1048018, and IOS-0820730) and by the DOE Grant USDOE/DE-SC0005013 (as part of the Exa-DM project, funded by Dr. Lucy Nowell, program manager, ASCR), and the Digital Technology Center at the University of Minnesota. Access to research and computing facilities was provided by the Digital Technology Center and the Minnesota Supercomputing Institute.

References

[1] F. Sadlo, R. Peikert, Efficient visualization of lagrangian coherent structures by filtered amr ridge extraction, IEEE Trans. Visualization Comput. Graphics 13 (6) (2007) 1456–1463.
[2] F. Zhang, S. Lasluisa, T. Jin, I. Rodero, H. Bui, M. Parashar, In-situ feature-based objects tracking for large-scale scientific simulations, in: High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion, 2012, pp. 736–740. doi:10.1109/SC.Companion..100.
[3] K.P. Gaither, H. Childs, K.W. Schulz, C. Harrison, W. Barth, D. Donzis, P. Yeung, Visual analytics for finding critical structures in massive time-varying turbulent-flow simulations, in: Computer Graphics and Applications, IEEE CG&A, 2012, pp. 34–35. doi:10.1109/MCG.2012.63.
[4] R. Kumar, J. Novak, A. Tomkins, Structure and evolution of online social networks, in: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '06, ACM, New York, NY, USA, 2006, pp. 611–617. doi:10.1145/1150402.1150476.
[5] D.S. Hirschberg, A.K. Chandra, D.V. Sarwate, Computing connected components on parallel computers, Commun. ACM 22 (8) (1979) 461–464, http://dx.doi.org/10.1145/359138.359141.
[6] Y. Shiloach, U. Vishkin, An o(log n) parallel connectivity algorithm, J. Algorithms 3 (1) (1982) 57–67.
[7] G. Cybenko, T. Allen, J. Polito, Practical parallel union-find algorithms for transitive closure and clustering, Int. J. Parallel Programm. 17 (1988) 403–423.
[8] C.A. Philips, Parallel graph contraction, in: Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '89, ACM, New York, NY, USA, 1989, pp. 148–157. doi:10.1145/72935.72952.
[9] J. Greiner, A comparison of data-parallel algorithms for connected components, in. Proc. 6th Ann. Symp. Parallel Algorithms And Architectures (SPAA-94), 1994, pp. 16–25.
[10] A. Krishnamurthy, S. Lumetta, D. Culler, K. Yelick, Connected components on distributed memory machines, Third DIMACS Implementation Challenge 30 (1997) 1–21.
[11] F. Manne, M. Patwary, A scalable parallel union-find algorithm for distributed memory computers, Parallel Process. Appl. Math. (2010) 186–195.
[12] L. Buš, P. Tvrdík, A parallel algorithm for connected components on distributed memory machines, Recent Adv. Parallel Virtual Mach. Message Passing Interface (2001) 280–287.
[13] U.N. Raghavan, R. Albert, S. Kumara, Near linear time algorithm to detect community structures in large-scale networks, Phys. Rev. E 76 (3) (2007) 036106.
[14] B. Wu, Y. Du, Cloud-based connected component algorithm, in: 2010 International Conference on Artificial Intelligence and Computational Intelligence (AICI), vol. 3, 2010, pp. 122–126. doi:10.1109/AICI.2010.360.
[15] C. Harrison, H. Childs, K.P. Gaither, Data-parallel mesh connected components labeling and analysis, in: Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization, EG PGV'11, 2011, pp. 131–140. doi:10.2312/EGPGV/EGPGV11/131-140.
[16] T. Seidl, B. Boden, S. Fries, Cc-mr finding connected components in huge graphs with mapreduce, in: P. Flach, T. Bie, N. Cristianini (Eds.), Machine Learning and Knowledge Discovery in Databases, Lecture Notes in Computer Science, vol. 7523, Springer, Berlin Heidelberg, 2012, pp. 458–473.
[17] E. Moore, The Shortest Path through a Maze, Bell Telephone System, 1959.
[18] R. Tarjan, Efficiency of a good but not linear set union algorithm, J. ACM (JACM) 22 (2) (1975) 215–225.
[19] V. Kumar, A. Grama, A. Gupta, G. Karypis, Introduction to Parallel Computing, vol. 110, Benjamin/Cummings, USA, 1994.
[20] J. Iverson, C. Kamath, G. Karypis, Fast and effective lossy compression algorithms for scientific datasets, in: Euro-Par 2012 Parallel Processing, 2012, pp. 843–856.
[21] G. Karypis, V. Kumar, Multilevel graph partition and sparse matrix ordering, in: Intl. Conf. on Parallel Processing, 1995, pp. 113–122.
[22] MSI, Itasca. http://www.msi.un.edu/hpc/itasca.