Optimizing Graph Algorithms on Pregel-like Systems*

Semih Salihoglu
Stanford University
semih@cs.stanford.edu

Jennifer Widom
Stanford University
widom@cs.stanford.edu

ABSTRACT

We study the problem of implementing graph algorithms efficiently on Pregel-like systems, which can be surprisingly challenging. Standard graph algorithms in this setting can incur unnecessary inefficiencies such as slow convergence or high communication or computation cost, typically due to structural properties of the input graphs such as large diameters or skew in component sizes. We describe several optimization techniques to address these inefficiencies. Our most general technique is based on the idea of performing some serial computation on a tiny fraction of the input graph, complementing Pregel's vertex-centric parallelism. We base our study on thorough implementations of several fundamental graph algorithms, some of which have, to the best of our knowledge, not been implemented on Pregel-like systems before. The algorithms and optimizations we describe are fully implemented in our open-source Pregel implementation. We present detailed experiments showing that our optimization techniques improve runtime significantly on a variety of very large graph datasets.

1. INTRODUCTION

Executing graph algorithms efficiently and at scale is important for many applications that process data in the form of a large graph. Examples include recommendation algorithms that run on large social networks [39], clustering algorithms on gene expression data [47], spam detection on the web graph [45], and many others. As graphs grow to sizes that far exceed the memory of a single machine, applications need to perform their computations on distributed systems. Google's *Pregel* [35], and its open-source implementations, such as *Giraph* [17] and *GPS* [42], are distributed message-passing systems targeted to large-scale graph computations. Like *MapReduce* [12] and *Hadoop* [20] for record-oriented data, Pregel-like systems offer transparent scalability, automatic fault-tolerance, and a simple programming interface based around implementing a small set of functions.

*This work was supported by the National Science Foundation (IIS-0904497), the Boeing Corporation, a KAUST research grant, and a research grant from Amazon Web Services.

Proceedings of the VLDB Endowment, Vol. 7, No. 7

This paper tackles the challenge of implementing graph algorithms efficiently on Pregel-like systems, which can be surprisingly difficult and require careful optimizations. We base our discussion on thorough implementations of several fundamental graph algorithms. We identify a variety of inefficiencies that arise when executing these algorithms on Pregel-like systems, and we describe optimization techniques, some of which are applicable across multiple algorithms, to address them. Some of the algorithms we cover have, to the best of our knowledge, not been implemented on Pregel-like systems before: the Coloring algorithm for finding strongly connected components [38], Boruvka's algorithm for finding a minimum spanning forest [11], a graph coloring algorithm based on Luby's algorithm for finding maximal independent sets [33], and the 1/2-approximation algorithm to maximum matching in general weighted graphs [40].

In the remainder of this section we provide brief background, and a summary of our approach and results. For reference, Tables 1 and 2 list the algorithms and optimization techniques covered in the paper.

1.1 Pregel Overview

The computational framework introduced by Pregel is based on the *Bulk Synchronous Parallel (BSP)* computation model [46]. At the beginning of the computation, the vertices of the graph are distributed across *Worker* tasks running on different compute nodes. Computation is broken down into iterations called *supersteps*, and all workers synchronize at the end of each superstep. Algorithms are implemented in a vertex-centric fashion inside a *vertex.compute()* function, which gets called on each vertex exactly once in every superstep. Inside *vertex.compute()*, vertices receive messages from the previous superstep, update their local states, and send messages to other vertices. In GPS [42] and Giraph [17], an optional *master.compute()* function is executed by the *Master* task between supersteps to perform serial computation, and for coordination in algorithms that are composed of multiple vertex-centric stages.

1.2 Costs of Computation

Broadly, there are four different costs involved when executing an algorithm on Pregel-like systems: (a) **communication**, i.e., number of messages transmitted between compute nodes; (b) **number of supersteps**; (c) **memory**, i.e., size of the local state stored per vertex; and (d) **computation** performed by vertices in each superstep. The optimization techniques we offer in this paper focus on reducing the first two costs: communication and number of supersteps. For the algorithms we consider, memory size and local computation are not dominant in overall run-time, nor do they appear to have room for significant improvement.

Name	Description	Section
Strongly Connected Components (SCC)	The Coloring algorithm from [38]	3.1
Minimum Spanning Forest (MSF)	Parallel version of Boruvka's algorithm [11]	3.2
Graph Coloring (GC)	Greedy algorithm based on Luby's parallel maximal independent set algorithm [33]	3.3
Approximate Maximum Weight Matching (MWM)	1/2-approximation algorithm for general graphs [40]	3.4
Weakly Connected Components (WCC)	Algorithm from [25]	3.5

Table 1: Algorithms.

Name	Description	Algorithms	Inefficiency	Section
Finishing Computations Serially (FCS)	Performs serial version of the algorithm, or a phase of the algorithm, inside *master.compute()*	SCC, GC, MSF, MWM, Backward-Traversal phase of SCC	Large-diameter graphs, skew in component sizes, small-size independent sets	4
Storing Edges At Subvertices (SEAS)	Stores the edges of supervertices in a distributed fashion among its subvertices	MSF	High-cost single phase	5
Edge Cleaning On Demand (ECOD)	Edges are cleaned only when they are used as part of the computation	MWM, MSF	High-cost single phase	6
Single Pivot (SP)	Detects giant component efficiently by starting the computation from a single vertex	SCC, WCC	Skew in component sizes	7

Table 2: Optimization Techniques.

Sometimes, system-level optimizations—those that do not require any changes to the graph algorithm itself—can be used to reduce communication and memory. For example, some distributed graph processing systems, such as KDT [34], PowerGraph [18], and GPS [42], use graph partitioning techniques that can reduce the communication cost significantly when executing some algorithms on input graphs with skewed degree distributions. In this paper we focus on optimizations that are algorithmic and do not appear to have any system-level equivalents. Whether system-level alternatives to some of our optimization techniques can be designed is an interesting question for future work.

1.3 Optimization Techniques

In this paper we propose four optimization techniques:

- **Finishing Computations Serially (FCS)**: Some algorithms or phases of algorithms may converge very slowly (i.e., execute many supersteps), while working on a tiny fraction of the input graph. FCS monitors the size of the "active" graph on which the computation is executing. If the active graph becomes small enough, FCS sends it to the master, which performs the end of the computation serially inside *master.compute()*, then sends the results back to the workers. In the algorithms we consider, slow convergence typically stems from structural properties of the graph, such as skew in components sizes and small-size maximal independent sets. From our experiments, we find that FCS can eliminate between 20% to %60 of total superstep executions (up to 16713 supersteps) when applied to several of our algorithms and their phases on large graphs.
- **Storing Edges At Subvertices (SEAS)**: SEAS is an optimization we apply primarily to our MSF algorithm (Table 1), in which sets of vertices (called *subvertices*) are merged to form *supervertices*. In the natural Pregel implementation of supervertex formation, subvertices send their adjacency lists to the supervertex, then become inactive. SEAS instead retains the adjacency lists of subvertices and keeps them active. SEAS effectively avoids the cost of sending adjacency lists to the supervertex, at the expense of incurring some communication cost between subvertices and supervertices, with overall run-time benefits.
- **Edge Cleaning On Demand (ECOD)**: "Edge cleaning" is a common operation in graph algorithms, in which vertices delete some neighbors from their adjacency lists based on some or all of their neighbors' values. The natural Pregel implementation of edge cleaning can be expensive: vertices send each other their values in one superstep, then clean their adjacency lists in another superstep. ECOD avoids the edge cleaning phase completely, removing "stale" edges only when they are discovered later in the computation.
- **Single Pivot (SP)**: The SP optimization technique was proposed originally in [41] for the WCC algorithm (Table 1). SP avoids unnecessary communication when detecting very large components in graphs with skewed component sizes. We show that SP is also applicable to the SCC algorithm. While SP shows only modest improvements for WCC in the setting of [41], it shows good improvements for both SCC and WCC in our setting.

1.4 Outline of the Paper

- Section 2 reviews the API of Pregel [35], and the API of GPS [42], the open-source Pregel software we use for our work. We also describe our experimental setup and datasets.
- Section 3 describes the algorithms we studied for this paper. The Pregel implementations of some of our algorithms, to the best of our knowledge, have not been described before, and were quite challenging.
- Section 4 covers our FCS optimization technique, which can eliminate many superstep executions when algorithms converge slowly. FCS is based on performing serial computation inside *master.compute()* on a small fraction of the input graph.
- Section 5 covers our SEAS optimization for Boruvka's minimum spanning forest algorithm.
- Section 6 covers our ECOD optimization technique, which can be used to eliminate the edge cleaning phases in some algorithms.
- Section 7 reviews the SP optimization from [41] and discusses how skew in component sizes can yield unnecessarily high com-

Name	Vertices	Edges	Description
sk-2005	51M	1.9B (d), 3.5B (u)	Web graph of the .sk domain from 2005
twitter	42M	1.5B (d), 2.7B (u)	Twitter "who is followed by who" network
friendster	125M	1.8B (d), 3.1B (u)	Friendster social network
uk-2005	39M	750M (d), 1.4B (u)	Web graph of the .uk domain from 2005
random-2.5B	500M	2.5B (d), 4.3B (u)	Graph with uniformly random edges

Table 3: Graph datasets.

munication cost in the strongly and weakly connected components algorithms we study.

- Sections 8 and 9 discuss related and future work, respectively.

Sections 4–7 all include extensive experiments demonstrating the benefits of our optimization techniques. All of our algorithms and optimizations are fully implemented on GPS [42] and are available for public download [19].

2. PRELIMINARIES

- *Vertex class*: Programmers subclass the *Vertex* class and code the vertex-centric logic of the computation by implementing the *vertex.compute()* function. Inside *vertex.compute()*, vertices can access their values, their incoming messages, and a map of *aggregators* (see below). Each vertex has an active/inactive flag. The system terminates computation when all vertices become inactive.
- *VertexValue class*: Encapsulates the user-defined state associated with each vertex.
- *Message class*: Encapsulates the messages sent between vertices.
- *Aggregators (Global Objects)*: Objects visible to all vertices and used for coordination, data sharing, and statistics aggregation. In GPS, aggregators are called global objects. When multiple vertices update the local copy of an object during a superstep, the system merges the updates using a user-specified merge function at the end of the superstep.
- *Master class*: Introduced by GPS and later adopted by Giraph [17]. Programmers can optionally subclass the *Master* class, and implement the *master.compute()* function, which gets called at the beginning of each superstep. The *Master* class can store its own local data and update the global objects before they are broadcast to the vertices. When algorithms are comprised of multiple vertex-centric computations, such as the algorithms we describe in this paper, the *master.compute()* function is used primarily to encapsulate the code that coordinates the different vertex-centric computations. In this paper, we will also use *master.compute()* in one of our optimization techniques.

2.1 Experimental Setup

The directed (d) and undirected (u) graphs we used in our experiments are specified in Table 3.[1] We used three different clusters for our experiments with varying number of compute nodes (m) and workers (w): (1) Large-EC2(m, w): Amazon EC2's large instance machines (four virtual cores and 7.5GB of RAM); (2) Medium-EC2(m, w): Amazon EC2's medium instance machines (two virtual cores and 3.75GB of RAM); (3) Local(m, w): our local cluster's machines (32 cores and 64GB of RAM). The machines in all our setups were running Red Hat Linux OS. We ran our experiments with fault-tolerance off and we ignore the initial data loading stage in our measurements.

We note that the run-time results we report may vary on a Pregel-like system other than GPS. However, except for our randomized SP optimization, the number of supersteps our algorithms and optimizations take will be exactly the same across systems. The relative network I/O effects of our optimizations will also be similar in all Pregel-like systems if the graph is partitioned randomly across workers as we do in this paper. The differences in the actual network I/O across systems would be due to the differences in the encoding and serialization of the IDs and messages between vertices.

We also note that we have not repeated our experiments in every cluster and every possible compute node and worker configuration. Experiments, not reported in the paper due to space constraints, suggest that the relative performance benefits of our optimizations do not vary significantly across configurations.

3. ALGORITHMS

We next describe the five graph algorithms we study in this paper (Table 1). To the best of our knowledge, Pregel implementations of the algorithms we present for the following problems have not been published before: strongly connected components (SCC), minimum spanning forest (MSF), graph coloring (GC) based on finding maximal independent sets (MIS), and maximum weight matching (MWM). We also note that implementing even the basic versions of SCC and MSF is quite challenging, taking more than 700 lines of code. Readers anxious to jump straight to our optimization techniques may skip ahead to Section 4, coming back to this section as a reference.

Most of our algorithms consist of multiple computational steps, which we call "phases" (not to be confused with the multiple supersteps that occur within each phase). For example, an algorithm might prune the graph in one phase and traverse it in another. In our implementations of these algorithms, a "phase" global object stores the current phase of the algorithm that is executing. The *master.compute()* function contains the logic of which phase should be executed in the next superstep, depending on the current phase and possibly other global objects. As a simple example, in the SCC algorithm (Section 3.1), when the current phase is "Forward-Traversal-Rest", the code fragment in Figure 1 is used by the master class to determine whether to switch the phase to "Backward-Traversal-Start".

For each phase, the *Vertex* class contains one subroutine implementing the vertex-centric logic of the phase. The *vertex.compute()* function calls the appropriate subroutine according to the value of the phase global object. As an example, Figure 2 shows the skeleton of the *vertex.compute()* function for SCC. All of our algorithms are implemented using this general pattern.

[1]The Web and the Twitter graphs were provided by "The Laboratory for Web Algorithmics" [31], using software packages WebGraph [8], LLP [7], and UbiCrawler [6]. The original Friendster social graph is undirected; we assign a random direction to each edge. For algorithms that take as input weighted graphs, we assign each edge a weight between 0 and 1 uniformly at random.

```
1  int numVerticesWithUpdatedColorIDs =
2      getGlobalObject("num-updated").value();
3  if (numVerticesWithUpdatedColorIDs > 0) {
4    setGlobalObject("phase", FW_TRAVERSAL_REST.value());
5  } else { setGlobalObject("phase",
6                           BW_TRAVERSAL_START.value());}
```

Figure 1: Example *SCCMaster* code for switching phases.

```
1  public class SCCVertex extends
2    Vertex<SCCVertexValue, SCCMessage> {
3   public void compute(Iterable <SCCMessage> messages) {
4    Phase phase = getPhase(getGlobalObject("phase"));
5    switch(phase) {
6    case TRANSPOSE_GRAPH_FORMATION_1: doTGF1();
7    case TRANSPOSE_GRAPH_FORMATION_2: doTGF2();
8    case TRIMMING: doTrimming();
9    case FW_START: doFwTraversalStart();
10   case FW_REST: doFwTraversalRest(messages); ... }}}
```

Figure 2: Skeleton code for *SCCVertex.compute()*.

3.1 Strongly Connected Components

We implement the parallel Coloring algorithm from [38] for finding strongly connected components. Figure 3 shows the original algorithm, with four phases:

1. **Transpose Graph Formation**: The algorithm first constructs the transpose of the input graph G (line 2).
2. **Trimming**: In the Trimming phase (line 4), the algorithm identifies *trivial* SCCs: vertices with only incoming or only outgoing edges (or neither).
3. **Forward-Traversal**: In the Forward-Traversal phase, which is encapsulated in the *MaxForwardReachable()* subroutine call on line 6, the algorithm traverses G in parallel from each vertex. During the traversals, each vertex v is *colored* by the maximum ID of the vertex that can reach v (possibly v itself). The Forward-Traversal phase has two properties: (1) G is partitioned into disjoint sets of vertices according to their colors, called *color sets*. (2) If S_i is the color set containing vertices colored i, then SCC_i, the SCC that vertex i belongs to, is entirely contained in S_i.
4. **Backward-Traversal**: In the Backward-Traversal phase (lines 7 –11), the algorithm detects one SCC for each color set S_i, by doing a traversal from vertex i in the transpose of G and limiting the traversal to only the vertices in S_i. The detected SCCs are then removed from the graph.

The algorithm repeats the Trimming, Forward-Traversal, and Backward-Traversal phases, each time detecting and removing from the graph one or more SCCs. It terminates when there are no vertices left in the graph.

In our distributed implementation of the algorithm, vertices contain two fields: (1) *colorID* stores the color of a vertex v in the Forward-Traversal phase and identifies v's SCC at the end of the computation, i.e., vertices with the same *colorID* after termination are in the same SCC. (2) *transposeNeighbors* stores the IDs of v's neighbors in the transpose of the input graph. The four phases in our distributed version operate as follows.

1. **Transpose Graph Formation**: Requires two supersteps. In the first superstep, each vertex sends a message with its ID to all its outgoing neighbors, which in the second superstep are stored in *transposeNeighbors*.

```
1  Coloring(G(V, E))
2  G^T = constructTransposeGraph(G)
3  while V ≠ ∅
4   Trim G and G^T
5   // colors vertices into disjoint color sets
6   MaxForwardReachable(G, start from every v ∈ V)
7   foreach p ∈ P in parallel:
8    if color(p) == p:
9      let S_p be the vertices colored p
10     SCC_p = S_p ∩ BackwardReachable(G^T, p)
11     remove SCC_p from G and G^T
```

Figure 3: Original Coloring algorithm for computing SCCs [38].

```
1  public void doFwStart() {
2    value(). colorID = getId();
3    sendMessages(getOutgoingNeighbors(),
4                 new SCCMessage(getId()));}
5  public void doFwRest(Iterable <SCCMessage> messages) {
6    int maxColorID = findMaxColorID(messages);
7    if (maxColorID > value().colorID) {
8      sendMessages(getOutgoingNeighbors(),
9                   new SCCMessage(value().colorID));
10     updateGlobalObject("updated-vertex-exists", true);}}
```

Figure 4: *SCCVertex* subroutines for the Forward-Traversal phase.

2. **Trimming**: Takes one superstep. Every vertex with only incoming or only outgoing edges (or neither) sets its *colorID* to its own ID and becomes inactive. Messages subsequently sent to the vertex are ignored.
3. **Forward-Traversal**: Figure 4 shows the subroutines implementing the Forward-Traversal phase in the *Vertex* class. There are two subphases: *Start* and *Rest*. In the Start phase, each vertex sets its *colorID* to its own ID and propagates its ID to its outgoing neighbors. In the Rest phase, vertices update their own colorIDs with the maximum colorID they have seen, and propagate their colorIDs, if updated, until the colorIDs converge. The *Master* sets the phase global object to Backward-Traversal when the colorIDs converge.
4. **Backward-Traversal**: We again break the phase into *Start* and *Rest*. In Start, every vertex whose ID equals its *colorID* propagates its ID to the vertices in *transposeNeighbors*. In each of the Rest phase supersteps, each vertex receiving a message that matches its *colorID*: (1) propagates its colorID in the transpose graph; (2) sets itself inactive; (3) sets the "converged-vertex-exists" global object (false at the start of the superstep) to true. Messages subsequently sent to the vertex are ignored. The *Master* sets the phase global object back to *Trimming* when "converged-vertex-exists" remains false at the end of a superstep.

3.2 Minimum Spanning Forest

We implement the parallel version of Boruvka's MSF algorithm from [11], referring the reader to [37, 11] for details. Figure 5 shows the original algorithm. The algorithm repeats four phases of computation in iterations, each time adding a set of edges to the MSF S it constructs, and removing some vertices from the graph until there are no vertices left.

1. **Min-Edge-Picking**: In parallel, each vertex v picks its current minimum-weight edge (v, u). Ties are broken by picking the

```
1  Boruvkas_MSF(G(V, E))
2  S = ∅
3  while V ≠ ∅
4    foreach v ∈ V in parallel:
5      e = PickMinWeightEdge(v)
6      S = S ∪ e
7      v.sv = FindSupervertexOfConjoinedTree()
8      removeNeighborsInSameConjoinedTree()
9      relabelIDsOfNeighborsInDifferentConjoinedTrees ()
10   foreach sv r ∈ V in parallel:
11     r.adjacencyList = mergeAdjacencyListsOfConjoinedTree();
12   remove all subvertices from V
```

Figure 5: Parallel version of Boruvka's MSF algorithm [11].

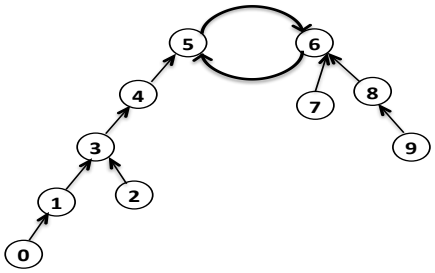

Figure 6: Example of a conjoined-tree.

edge with minimum destination ID. Each picked edge (v, u) is added to S. As proven in [11], the vertices and their picked edges form disjoint subgraphs $T_1, T_2, \ldots, T_k$, each of which is a *conjoined-tree*: two trees, the roots of which are joined by a cycle. Figure 6 shows an example of a conjoined-tree. In the figure, vertex 0 picks vertex 1 as its minimum-weight edge, vertex 1 picks vertex 3, vertex 2 picks vertex 3, etc. We refer to the vertex with the smaller ID in the cycle of T_i as the *supervertex* of T_i, for example vertex 5 in Figure 6. All other vertices in T_i are called *subvertices*. The following phases merge all of the subvertices of each T_i into the supervertex of T_i.

2. **Supervertex-Finding**: Each vertex finds the supervertex of the conjoined-tree it belongs to (line 7).
3. **Edge-Cleaning-and-Relabeling**: Each vertex v performs one of two operations for each of its neighbors u: v removes u if v and u are in the same conjoined-tree (line 8), or relabels u with the supervertex of the conjoined-tree u belongs to, possibly u itself (line 9). For correctness of the final output, the algorithm stores the original source and destination IDs on each edge, which remain unchanged throughout the computation.
4. **Supervertex-Formation**: The algorithm merges the relabeled adjacency lists of each T_i at its supervertex, keeping the minimum-weight edges for duplicates (lines 10-12). All subvertices as well as supervertices with no edges are removed from the graph.

In our distributed implementation, each vertex stores a *type* and a *pointer* field, which are used in the Supervertex-Finding phase (explained below). Each vertex v also has a *pickedEdgeSrcID* and a *pickedEdgeDstID* field (initialized to null), which respectively store the original source and destination IDs of the last edge v picks, i.e., the edge v picks in the last Min-Edge-Picking phase it participates in. At the end of the computation, these fields identify the edges that are in S.

1. **Min-Edge-Picking**: Each vertex picks its minimum-weight edge and writes its *pickedEdgeSrcID* and *pickedEdgeDestID* fields.
2. **Supervertex-Finding**: For finding supervertices we implement the *Simple Pointer Jumping Algorithm* from [11]. There are two subphases: *Question* and *Answer*. We explain the subphases using the example conjoined-tree from Figure 6. Initially each vertex v sets its *type* to *Unknown* and *pointer* to the neighbor v picked in Min-Edge-Picking. In the first *Question* phase every vertex v sends a *question message* to $v.pointer$. In our example, vertex 0 sends a message to 1, and 5 and 6 send messages to each other. In the first *Answer* phase, 5 and 6 see that they've sent each other messages and discover that they are part of the cycle of the conjoined-tree. 5 sets its type to *Supervertex* and 6 to *PointsAtSupervertex*. Every other vertex sets its type to *PointsAtSubvertex*. In addition, if a vertex v receives a question, it replies with an answer that contains the ID of *v.pointer* and whether $v.pointer$ is the supervertex. For example, 1 sends 0 the message <3, *isSupervertex*:false>, while 6 sends 7 and 8 the message <5, *isSupervertex*:true>.

   From then on, we execute the *Question* and *Answer* phases in iterations until every vertex points to the supervertex of its conjoined-tree. Let d_i be the longest distance of any leaf vertex in the conjoined-tree T_i to its supervertex, and let d_{max} be the maximum over all d_i. Supervertex-Finding takes $log(d_{max})$ supersteps.
3. **Edge-Cleaning-and-Relabeling**: Takes two supersteps. First, each vertex v sends its ID and supervertex ID to all of its neighbors. The supervertex ID is effectively the new ID for v. In the second superstep, vertex v for each of its edges $e = (v, u)$ either deletes e if u has the same supervertex ID, or relabels e to point to u's new ID.
4. **Supervertex-Formation**: Takes two supersteps. First, every subvertex sends its edges to its supervertex and becomes inactive. Then, each supervertex merges and stores these edges, keeping the minimum-weight for duplicates.

3.3 Graph Coloring (GC)

Graph coloring is the problem of assigning a color to each vertex of an undirected graph such that no two adjacent vertices have the same color. We implement the greedy algorithm from [16]. The algorithm iteratively finds a maximal independent set (MIS) of vertices, i.e., a maximal set of vertices such that no pair of vertices are adjacent. The algorithm assigns the vertices in each MIS a new color, then removes them from the graph, until there are no vertices left in the graph.

For finding an MIS we use Luby's classic parallel algorithm [33]. The algorithm maintains three sets of vertices:

- *S*: The MIS being constructed. Starts empty and grows in iterations.
- *NotInS*: Vertices that have at least one edge to a vertex in *S* and as a result cannot be in *S*.
- *Unknown*: Vertices that do not have an edge to any vertex in *S* but are not yet in *S*.

In each iteration of MIS, each *Unknown* vertex *v* is first tentatively added to *S* with $\frac{1}{2 \times degree(v)}$ probability. Suppose v and some of its neighbors, $u_1, u_2, ..., u_k$, are added to *S*. Then, *v* is kept in *S* only if its ID is less than all IDs of $u_1, u_2, ..., u_k$. Otherwise, *v* is put back to *Unknown*. By putting only the minimum ID vertex into *S*, the algorithm guarantees that two neighbor vertices are not added to *S*, i.e., vertices in *S* are independent. If a vertex *w* is in *Unknown* and has an edge to any of the vertices that were added to *S*, then *w* is put

into *NotInS*. Finally, vertices that remain in *Unknown* decrement their degree counts by the number of their neighbors that were put into *NotInS*. The iterations continue until there are no *Unknown* vertices.

In our distributed implementation of GC, vertices have *type* and *degree* fields that are used in the MIS construction, and a *color* field to store the color of the vertex (initialized to null). There are six phases:

- **MIS-Degree-Initialization**: Executed once for each MIS the algorithm constructs. Takes two supersteps. First, each vertex remaining in the graph sets its *type* to *Unknown* and sends an empty message to all its neighbors. In the second superstep, vertices set their *degree* fields to the number of messages they receive.
- **Selection**: Takes one superstep. Each vertex v sets its type to *TentativelyInS* with $\frac{1}{2\times degree(v)}$ probability, then notifies its neighbors with a message containing its ID.
- **Conflict-Resolution**: Takes one superstep. Each vertex v that is tentatively in *S* inspects the IDs of its messages. If v has the minimum ID among its messages, it sets its type to *InS*, then sends an empty "neighbor-in-set message" to its neighbors. Otherwise, v sets its type back to *Unknown*.
- **NotInS-Discovery-and-Degree-Adjusting-1**: If v receives a neighbor-in-set message, v sets its type to *NotInS*, becomes inactive, and sends an empty "decrement degree" message to its neighbors.
- **Degree-Adjusting-2**: Every vertex v that is of type *Unknown* decreases its degree by the number of messages it receives. If there are remaining *Unknown* vertices, the master sets the phase back to *Selection*. Otherwise, the MIS construction is complete and the master sets the phase to *Color-Assignment*.
- **Color-Assignment**: Each vertex that is of type *InS* sets its *color* field to a new color, which is broadcast by the master inside a global object, and becomes inactive. Vertices of type *NotInS* set their types back to *Unknown*.

3.4 Approximate Maximum Weight Matching (MWM)

A maximum weight matching (MWM) of a weighted undirected graph G is a set of edges *M* such that each vertex v is adjacent to at most one edge in *M* (i.e., *M* is a matching) and there is no other matching that has higher weight than *M*. We implement the approximate MWM algorithm from [40]. In each iteration of MWM, vertices select their maximum-weight neighbors. If u and v select each other, the edge (u, v) is added to *M*, and u and v (along with all edges pointing to them) are removed from the graph. The iterations continue until there are no vertices left in the graph. A proof that the algorithm computes a 1/2-approximation to the maximum matching in the graph can be found in [40]. In our implementation of the algorithm, we store a *pickedEdge* field per vertex, which is initially null. There are three phases:

- **Max-Weight-Edge-Picking**: Every vertex picks its maximum weight neighbor *u* (ties are broken by picking the neighbor with minimum ID), stores it tentatively in its *pickedEdge* field, and sends a message to u containing v's ID.
- **Match-Discovery**: If v and u have picked each other, they send a notification message to their neighbors that they are matched and become inactive.
- **Removing-Matched-Neighbors**: Each unmatched vertex v receives messages from its matched neighbors, and removes them from v's adjacency list. The master sets the phase back to Max-Weight-Edge-Picking if there are unmatched vertices, otherwise terminates the computation.

3.5 Weakly Connected Components (WCC)

We implement the distributed HCC algorithm from [25]. HCC consists of a single phase, which is identical to the *Forward-Traversal* phase of the SCC algorithm from Section 3.1. In iterations, vertices propagate their IDs and keep the maximum ID they have seen until convergence. Similar to our implementation of Forward-Traversal, we break the phase into two subphases. In the *Start* phase, which takes one superstep, vertices initialize their *wccIDs* to their own IDs and propagate their IDs to their neighbors. In the *Rest* phase, vertices update their own *wccIDs* with the maximum *wccID* they have seen, and propagate their *wccIDs* (if updated). The *Rest* phase continues until *wccIDs* converge.

4. FINISHING COMPUTATIONS SERIALLY (FCS)

We now describe our first optimization technique, *Finishing Computations Serially* (FCS). FCS addresses slow convergence in algorithms by performing some serial computation on a tiny fraction of the input graph. Sections 4.1 and 4.2 give a high-level description and explain the implementation of FCS, respectively. Section 4.3 discusses the benefits and overheads of FCS. In Sections 4.4 and 4.5, we apply FCS to the SCC algorithm from Section 3.1, addressing slow convergence due to large diameters and skewed component sizes, respectively. In Section 4.6 we apply FCS to the GC algorithm (Section 3.3), which converges slowly on graphs with small-size maximal independent sets. Sections 4.4–4.6 include experimental results.

4.1 High-level Description

Sometimes, an algorithm or a phase of the algorithm may converge very slowly, i.e., execute for a large number of supersteps, while executing on a very small fraction of the input graph, which we refer to as the *active-subgraph*. Since Pregel-like systems synchronize and exchange coordination messages at each superstep, slow convergence can significantly degrade performance. The premise of FCS is to avoid a large number of these small superstep executions by finishing the computation on a small active-subgraph serially, inside *master.compute()*. FCS monitors the size the active-subgraph. Once the size of the active-subgraph is below a threshold (5M edges by default), it sends the active-subgraph to the master, which performs the rest of the computation serially, and sends the results back to the workers. In the remainder of this section we call the vertices that are in the active-subgraph, i.e., those that can contribute to the computation in the remainder of the algorithm or a phase of the algorithm, as *potentially-active* (not to be confused with Pregel's active/inactive flag). We call a vertex *certainly-inactive* otherwise.

FCS can be applied to algorithms in which the size of the active-subgraph shrinks throughout the computation. All of the algorithms we study in this paper, except for WCC, have this "shrinking active-subgraph" property. For example, the SCC algorithm (Section 3.1) detects several components in each iteration and removes them from the graph, decreasing the size of the active-subgraph.

FCS can also be applied to individual phases, if the active-subgraph of the phase is shrinking. As an example, consider the Backward-Traversal phase of SCC. A vertex v becomes certainly-inactive in two ways: (1) v receives a message that contains its colorID, then v discovers its component; or (2) no vertex in the graph propagates a message containing v's colorID. In the second case, v will

not discover its component in the rest of the phase and will be assigned a new colorID in the next iteration. In contrast, recall the Forward-Traversal phase, in which vertices update and propagate their colorIDs by the maximum until convergence. In this phase, no vertex becomes certainly-inactive until the phase is complete, as we cannot tell with certainty that a vertex will not update its colorID later without observing the entire graph—a larger colorID may be propagating from another part of the graph to the vertex.

4.2 Implementation

Our implementations of FCS all use three global objects, to store: (1) the number of edges in the active-subgraph; (2) the active-subgraph when serial computation is triggered; (3) the results of the serial execution (e.g., the component IDs in SCC), which are used by the potentially-active vertices to update their values in a superstep following serial execution. Implementations of FCS in specific algorithms and phases differ in the way the potentially-active vertices are identified, and the way the serial computation is performed inside *master.compute()*. In Sections 4.4–4.6, we explain the implementation differences among the algorithms and phases we apply FCS to.

4.3 Cost Analysis

FCS avoids additional superstep executions after serial propagation is triggered. On the other hand, it incurs the overhead of: (a) monitoring the size of the active-subgraph, which involves potentially-active vertices incrementing a global object; (b) serial computation at the master; (c) communication cost of sending the active-subgraph to the master and the results of the computation back to the workers; (d) one superstep execution for vertices to read the results. The actual benefits and overheads depend on the algorithm and the graph, and we report experimental results in Sections 4.4–4.6. We note that we expect FCS to yield good benefits only when the algorithm or a phase converges very slowly. For example, FCS can be applied to MSF, in which the active-subgraph shrinks, but convergence is not slow.

We also note that in general one can trigger FCS as soon as the active subgraph is as large as the size of the memory of of the master worker. Triggering FCS earlier would reduce the number of superstep executions and communication, at the expense of parallel computation. In practice, there will be an optimal threshold that minimizes the run-time of the computation. However, we advise keeping a very low threshold (such as 5M edges) instead of trying to find an optimal one. The motivation for FCS is to detect and avoid situations in which the algorithm cannot parallelize a computation due to the existence very small subgraphs, instead of performing serial computations on large subgraphs.

4.4 FCS for Backward-Traversal Phase of SCC (FCS-BT)

It has been observed that some real-world graphs have very large actual diameters but very small "effective diameters" [10, 32]. In other words, although most vertices are very close to each other, a small fraction of the vertices are far apart. On a graph with diameter d, the Backward-Traversal phase of SCC can take d supersteps: a vertex v might be at distance d from the vertex that starts propagating v's colorID. However, if the graph has a short effective diameter, most vertices may become certainly-inactive after a few supersteps, with only a small fraction of the vertices participating in the rest of the supersteps. To avoid some of these final superstep executions, we apply FCS to Backward-Traversal, and refer to this optimization as *FCS-BT*.

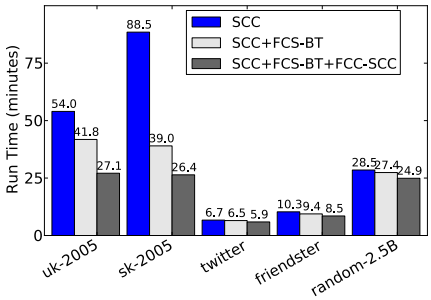

Figure 7: FCS-BT and FCS-SCC, Medium-EC2(90, 90).

In order to monitor the size of the subgraph, we identify a vertex v as potentially-active if: (1) v has not yet received a message containing its colorID; and (2) there is at least one vertex propagating a message containing v's colorID. We use a global object *propagating-colorIDs-set*, which is updated by vertices that propagate messages. Vertices that have not found their components look at this object to check whether their colorIDs are being propagated. For serial computation, we do a simple serial breadth-first search traversal at the master.

4.4.1 Experiments

To evaluate the overall run-time benefits, we applied FCS-BT to the SCC algorithm and ran experiments with and without the optimization. Figure 7 (ignore the "FCS-SCC" bars for now; see next section) shows the results. FCS-BT yields 1.3x and 2.3x run-time improvements on the *uk-2005* and *sk-2005* web graphs, respectively, when applied to the baseline SCC algorithm. In terms of supersteps, FCS-BT reduced the total number of supersteps by 28% in *uk-2005* (from 4546 to 3278) and 56% in *sk-2005* (from 6509 to 2857). FCS-BT does not show much improvement on non-web graphs, which have significantly smaller diameters than web-graphs. As a result, on non-web graphs the total number of supersteps the Backward-Traversal phases take is small (≤ 50) so eliminating them does not significantly improve performance. Over all five experiments, monitoring the size of the active-subgraph slowed down the run-time of supersteps before serial computation on average by 1.3%, which was minor compared to the benefits of FCS-BT.

4.5 FCS for SCC (FCS-SCC)

As observed in [32], real graphs can have skewed component sizes and exhibit a large number of very small components. Even if the small-size components comprise a small fraction of the original graph, detecting them may take a large number of iterations. Suppose 100 (say) small-size components are connected to each other. Then, in the Forward-Traversal phase, a vertex with a large ID from one component can color many other components and prevent their detection. (Recall the algorithm from Section 3.1.) In an entire iteration, the algorithm may detect only a few of the 100 connected components. Therefore, we may have to run many iterations of SCC before detecting all 100 components. Applying FCS to the SCC algorithm as a whole (called *FCS-SCC*) can eliminate some of these iterations. In our implementation of FCS-SCC, we monitor the size of the active-subgraph simply by counting the edges of the vertices whose components have not been discovered. For serial computation, we use Kosaraju's classic SCC algorithm [43].

4.5.1 Experiments

To evaluate the additional run-time benefits of FCS-SCC over FCS-BT, we repeated our experiments from Section 4.4.1, using FCS-BT in the Backward-Traversal phase and FCS-SCC for the overall algorithm. Figure 7 shows that FCS-SCC yields between

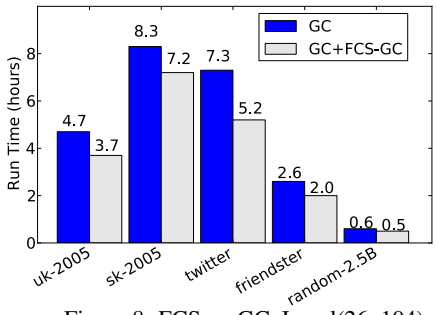

Figure 8: FCS on GC, Local(26, 104).

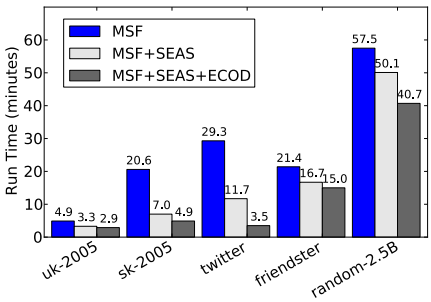

Figure 9: SEAS and ECOD on MSF, Large-EC2(76, 152).

1.1x to 2.4x additional run-time improvement across our five datasets. FCS-SCC decreased the number of superstep executions by up to an additional 39%. Similar to FCS-BT, FCS-SCC performs better on web-graphs, which have very large diameters. Because of the diameter, executing the forward and backward traversals can take a very large number of supersteps; avoiding them increases performance significantly. We note that the time it took to monitor the active-subgraph and execute Kosaraju's serial SCC algorithm was negligible compared to the rest of the computation in all of our experiments.

4.6 FCS for GC (FCS-GC)

The GC algorithm from Section 3.3 iteratively finds a maximal independent set (MIS) M in the graph, gives all of the vertices in M a color, and removes them from the graph. Thus, the active-subgraph shrinks throughout the computation. When executing GC on our datasets, we also observed that over time the active-subgraph gets denser, and as a result the independent sets get smaller. Nearing the end of the algorithm, we can be left with a small clique and need to find as many independent sets as there are vertices in the clique. Each iteration takes at least five supersteps (see Section 3.3), and hence on a clique of only 100 vertices, the algorithm executes at least 500 supersteps. We apply FCS to GC (called *FCS-GC*) to eliminate some of these supersteps. In our implementation, we use a serial version of the algorithm also based on finding independent sets: We compute the sets greedily, putting each vertex in a queue, then one by one putting each vertex v in the set if none of v's neighbors is already in the set.

4.6.1 Experiments

To evaluate the benefits of FCS-GC, we ran the GC algorithm with and without the optimization. Figure 8 shows a sample of our experiments. FCS-GC yields between 1.1x to 1.4x run-time benefits, reducing the superstep executions between 10% to 20%. (For example, on the sk-2005 graph, FCS-GC reduced the number of superstep executions from 85980 to 69267.) On active-subgraphs of less than 5M edges, the serial computation takes under 5 seconds and is negligible compared to the rest of the computation. Similar to FCC-SCC, the cost of monitoring the active-subgraph was also negligible.

5. STORING EDGES AT SUBVERTICES (SEAS) IN MSF

5.1 High-level Description

Recall the MSF algorithm from Section 3.2. In the Supervertex-Formation phase, a supervertex s receives and merges the adjacency lists of its subvertices, a high-cost operation. Our SEAS optimization instead stores the edges of a supervertex s in a distributed fashion among all of its subvertices. With the adjacency lists of supervertex s distributed, s must pick its minimum-weight edge in a distributed fashion. Moreover, if s is merged into another supervertex in a particular iteration, it has to notify its subvertices of the new supervertex they belong to. Nevertheless, the added work is offset by the improved performance of avoiding the very costly phase of sending and merging the adjacency lists of all subvertices. We note that SEAS can be applied to other algorithms that form supervertices during the computation but are not covered in this paper, e.g., Karger's classic randomized minimum cut algorithm [27] or the METIS graph partitioning algorithm [36].

5.2 Implementation

In our implementation of SEAS, subvertices store a pointer to their latest supervertices. The Min-Edge-Picking phase is now performed in two supersteps. In the first superstep, subvertices send their local minimum-weight edges to the supervertex. In the second superstep, a supervertex s picks the minimum of its local edge and the edges it receives. The Supervertex-Finding and Edge-Cleaning-and-Relabeling phases are performed as usual. Instead of Supervertex-Formation, we perform a new phase called *New-Supervertex-Notification*, which takes three supersteps: (1) Suppose we are in iteration i. Every subvertex from iteration $i-1$ sends a message to its latest supervertex containing its ID. (2) Every supervertex s from iteration $i-1$ sends a message back to its supervertices containing the ID of its new supervertex (possibly s itself). (3) Subvertices from iteration $i-1$ update their supervertices with the ID they receive.

5.3 Cost Analysis

Let $G_i(V_i, E_i)$ be the remaining graph in the ith iteration of the baseline MSF, and let SV_i be the number of subvertices that have at least one edge remaining in the ith iteration when SEAS is on. In iteration i, SEAS avoids the computation of merging edge lists and inserting merged edges at supervertices. However, SEAS's effects on communication are twofold. If we assume for simplicity that all of the edges of supervertices in iteration i+1 come from their subvertices, then SEAS roughly avoids $|E_i|$ amount of communication in iteration i. On the other hand SEAS incurs $3*|SV_i|$ additional communication: each subvertex sends one message to its supervertex in the Min-Edge-Picking and sends one and receives one message in New-Supervertex-Formation. Therefore, whether SEAS increases or decreases communication depends on how the sizes of E_i and SV_i change in each iteration. In effect, we avoid the communication and computation performed in the Supervertex-Formation phase, which is proportional to the number of edges in the graph, at the expense of increasing the cost of Min-Edge-Picking and incurring the costs of New- Supervertex-Formation, which are proportional to the number of vertices in the graph.

5.4 Experiments

To evaluate the effects of our SEAS optimization, we ran the MSF algorithm both with and without the optimization. Figure 9 shows the results on our five datasets (ignore the MSF+SEAS+ECOD bars for now). We find with SEAS the overall performance increases between 1.15x and 3x across our data sets. We also note that SEAS decreases the total communication cost of the algorithm by up to 1.4x. Recall from the previous section that we expect SEAS to perform better when $|E_i|$ is large (SEAS avoids more communication and computation) and $|SV_i|$ is low (SEAS incurs less extra communication). Therefore SEAS' performance depends on the ratio of $|E_i|$ to $|SV_i|$. Random-2.5B is the sparsest graph we experimented with (on average each vertex has only 5 vertices) and $|SV_i|$ was very high in each iteration of MSF (always $\geq$ 400M vertices). As a result, SEAS improved performance less significantly for random-2.5B than other graphs.

6. EDGE CLEANING ON DEMAND

6.1 High-level Description

"Edge cleaning" is a common graph operation in which vertices delete some of their neighbors according to the neighbors' values. For example, in the Removing-Matched-Neighbors phase of the MWM algorithm from Section 3.4, unmatched vertices remove edges to their matched neighbors. Other examples include the (not surprisingly named) Edge-Cleaning-and-Relabeling phase in SCC, as well as phases from other algorithms that are not covered in this paper. (For example, Karger's classic randomized minimum cut algorithm forms supervertices in iterations and "cleans" edges within supervertices [27].) In the natural implementation of edge cleaning on Pregel-like systems, vertices send messages to their neighbors in one superstep, and remove neighbors in another, possibly based on the contents of the messages. This implementation incurs a communication cost proportional to the number of edges in the graph, which might be expensive. In addition, sometimes an edge *e* might be deleted unnecessarily: even if we keep *e* around, *e* may never be used again or otherwise affect the computation.

Our ECOD optimization technique keeps stale edges around instead of deleting them. Vertices "pretend" every edge is valid, and a stale edge *e* is discovered and deleted only when a vertex attempts to use *e* as part of the computation. Among the algorithms we studied in this paper, ECOD can be applied to MWM and MSF, although implementations of ECOD on different algorithms differ in the way stale edges are discovered. We next describe the implementations and cost analysis of ECOD for each algorithm.

6.2 Implementation for MWM

Recall that in the Match-Discovery phase of our implementation of MWM in Section 3.4, when a vertex v finds its match u (say), v sends all its neighbors a notification message and becomes inactive. In the following Removing-Matched-Neighbors phase, all of v's unmatched neighbors remove v from their adjacency lists. With ECOD, instead v sends a message only to its neighbors that requested a match with v, and only those neighbors remove v from their adjacency lists in the Removing-Matched-Neighbors phase; v's other unmatched neighbors keep their "stale" edges to v. In addition, v now stays active in order for its neighbors to discover their stale edges to v: If in a future iteration of the algorithm a vertex z uses its stale edge to v and requests to match with v, v sends z a notification message and z removes its stale edge to v; otherwise, edge (z, v) is never removed. As we discuss below, ECOD may decrease the number of vertices that converge in each iteration; i.e. vertices that find a match or clean all their edges and discover that they will remain unmatched in the remainder of the computation. To avoid very slow convergence we switch to regular edge-cleaning if the number vertices that converge is below a threshold number of vertices (1% of all the vertices in the graph by default).

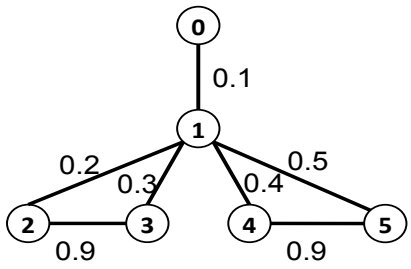

Figure 10: Example graph for ECOD cost analysis.

6.3 Cost Analysis for MWM

ECOD decreases the amount of communication and computation a matched vertex generates: instead of sending a message to all neighbors, matched vertices send messages only to those neighbors that request to match with them. As a result, similar to SEAS, ECOD effectively avoids costs that are proportional to the number of edges in the graph, at the expense of incurring costs that are proportional to the number of vertices. In our experiments (discussed momentarily), we observed significant performance benefits from this tradeoff. In addition, ECOD can avoid the communication and computation of deleting some edges unnecessarily. For example, consider an edge (v, u), and assume that v and u both match with other vertices in the first iteration. Without ECOD, they would send each other unnecessary notification messages.

On the other hand, ECOD may slow down the convergence of the algorithm, decreasing the number of vertices that match (or that discover not to match any other vertex) in each iteration, which increases the number of iterations. Consider the simple graph in Figure 10. Without ECOD, the algorithm takes two iterations. In the first iteration, vertex 2 matches with 3, and 4 matches with 5. In the second iteration, 1 matches with 0. Using ECOD, the algorithm takes five iterations. In the first four iterations, vertex 1 attempts to match with its heaviest edges 5, 4, 3, and 2 one by one and fails. After removing these edges, it finally matches with 0 in the fifth iteration.

6.4 Implementation for MSF

Recall that in the Edge-Cleaning-and-Relabeling phase of MSF (Section 3.2), vertices send their supervertex IDs to their neighbors in the first superstep. In the second superstep, vertices remove their neighbors that have the same supervertex ID from their adjacency lists. With ECOD, we omit this phase completely. As a result, during Min-Edge-Picking, vertices cannot pick their minimum weight edges directly in one superstep, as some of their edges may be stale. Instead, we execute a *Stale-Edge-Discovery* phase, during which vertices discover whether or not their minimum weight edges are stale. In the first superstep, each vertex v tentatively picks its minimum weight edge u (say) and sends a "question" message to u containing v's ID and v's supervertex ID, i (say). In the second superstep, if u belongs to a different supervertex j (say), it sends an answer message back to v containing the value j. In the third superstep, if v receives an answer message, it successfully picks u as its minimum-weight edge and relabels it to j; otherwise v removes u. We run the Stale-Edge-Discovery phase a threshold number of times (two by default). We then run a final *All-Stale-Edges-Discovery* phase, in which vertices that have not yet successfully

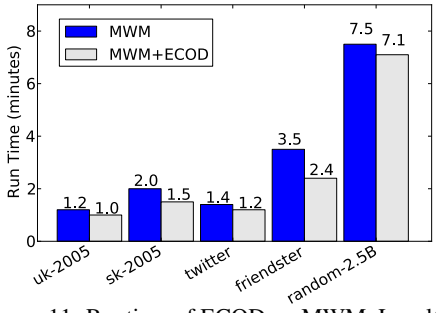

Figure 11: Runtime of ECOD on MWM, Local(20, 80).

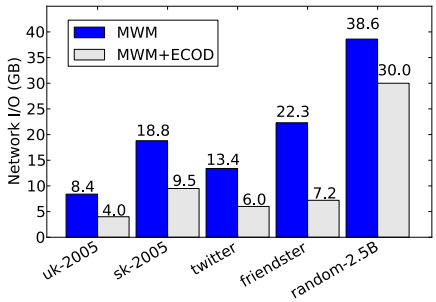

Figure 12: Network I/O of ECOD on MWM, Local(20, 80).

picked their minimum-weight edges send a question message to all their neighbors and clean all of their stale edges.

6.5 Cost Analysis for MSF

ECOD avoids the communication and computation of the Edge-Cleaning-and-Relabeling phase. On the other hand, ECOD incurs the extra communication and computation cost of question and answer messages in the new Stale-Edge-Discovery and All-Stale-Edges-Discovery phases. Without ECOD, in each Edge-Cleaning-and-Relabeling phase, a message is sent over each remaining edge in the graph. As a result, if an edge (v, u) is not picked as a minimum-weight edge or cleaned for j iterations, v will send exactly j messages over (v, u). In contrast with ECOD, v can exchange as few as one question and one answer message with u, and certainly no more than j messages. There are two cases:

1. (v, u) is picked as a minimum-weight edge in the jth iteration: Then, in the jth iteration, v exchanges one question and one answer message with u in the Stale-Edge-Discovery phase, and successfully picks u as its minimum-weight edge. In the first $j-1$ iterations, v sends a message to u only if v cannot pick its minimum-weight edge in the Stale-Edge-Discovery phases, and has to clean all its stale edges in All-Stale-Edges-Discovery.
2. (v, u) is cleaned in the jth iteration: v exchanges one questions and answer message with u and cleans (u, v). For the first $j-1$ iterations, the situation is exactly as case (1) and v sends a message to u only if v has to run All-Stale-Edges-Discovery in the first $j-1$ iterations.

As a result v cumulatively sends between 1 and j messages to u in both cases. Therefore the amount of communication of ECOD is less than or equal to baseline MSF's. We have observed in our experiments that with ECOD, significantly fewer messages are sent over edges in general.

On the negative side, ECOD increases the overall number of superstep executions. It avoids two superstep executions of the Edge-Cleaning-and-Relabeling phase but executes the three superstep Stale-Edge-Discovery phase several times.

6.6 Experiments

We evaluated ECOD on both MWM and MSF. As shown in Figures 11 and 12, ECOD improves run-time of MWM by up to 1.45x and decreases the total communication cost between 1.3x to 3.1x across our data sets. ECOD increased the number of supersteps between 1.7x to 2.2x. For MSF, we applied ECOD in combination with our SEAS optimization (Section 5). The results were included in Figure 9. When used in combination with SEAS, ECOD yields between 1.2x and 3.3x additional run-time benefit. ECOD also decreased the total communication cost of MSF by up to 1.9x across our experiments. For MSF, the overall performance improvements on uk-2005 were modest because ECOD improved the communication cost only by 1.03x.

7. SINGLE PIVOT OPTIMIZATION (SP)

In this section we describe how skew in component sizes can yield unnecessarily high communication cost in the component detection algorithms we study. We review the *Single Pivot* (SP) optimization, which was originally described in reference [41] to improve the performance of the WCC algorithm (Section 3.5). We show that the optimization can also be applied to the SCC algorithm from Section 3.1. While reference [41] reports minor performance benefits running WCC on two small synthetic graphs (fewer than 240M edges), we report major benefits on our larger real-world graphs for both WCC and SCC.

7.1 High-level Description and Implementation

In addition to a large number of small-size components, graphs with skewed component sizes typically exhibit a single "giant" component, which contains a significant fraction of the vertices in the graph [10]. The SP optimization, originally described in [41] for WCC, is designed to detect giant components efficiently. Initially, the optimization picks a single vertex r (called the *pivot*) and finds the component that r belongs to, by propagating r's ID along its neighbors (either in WCC or the Forward-Traversal phase of SCC). The process is repeated until a large component is found, or a threshold number of iterations is reached. At that point the original algorithm is used for the remainder of the graph. To implement SP, we added a new initial phase, *Random-Pivot-Picking*, to the WCC and SCC algorithms. In this phase, every vertex updates a custom global object that picks one of the vertices as pivot uniformly at random. Then, only the pivot starts propagating its ID in the WCC algorithm and the Forward-Traversal phase of the SCC algorithm.

7.2 Cost Analysis

The existence of a giant component can incur unnecessary costs in the WCC algorithm: Let w be the vertex with maximum ID in the giant component. Then all propagation messages inside the giant component, except those that contain the ID of w, are unnecessary: they will not contribute to the final *wccID* values.

The situation is potentially worse for the SCC algorithm. Let *CON* denote the (usually) larger set of vertices that are connected to the giant (strongly-connected) component, and let w be the vertex with maximum ID in *CON*. There are two possibilities: (1) If w is in the giant component, then in the Forward-Traversal phase, we incur the same unnecessary propagation messages as in WCC. (2) If w is not in the giant component, then we will detect the SCC that w belongs to, which typically is much smaller than the giant one. As a result, much of the computation and messages will be repeated in a later iteration.

If SP picks a pivot vertex from the giant component, it avoids all of the unnecessary propagation messages and computation costs of detecting the giant component. On the other hand, for each iteration

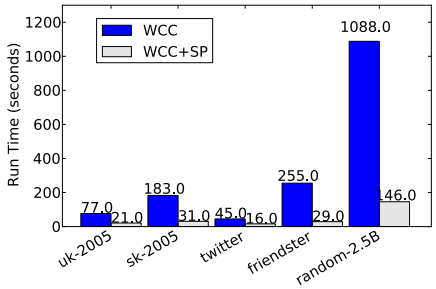

Figure 13: SP on WCC, Large-EC2(100, 100).

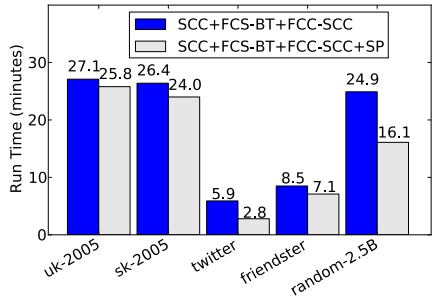

Figure 14: SP on SCC, Medium-EC2(90, 90).

failing to pick a pivot r from the giant component, SP decreases the parallelism of the algorithms: instead of detecting multiple components in an iteration, SP detects only r's component. As a result, SP might increase the total number of superstep executions. Also, in the Forward-Traversal phase of SCC, picking a pivot from outside the giant component can result in unnecessary propagation messages among many vertices that do not belong to the pivot's component. However, SP always incurs less communication cost than baseline WCC. This follows from the observation that with SP there will be exactly one message sent over each edge in the component of pivot r, which is the cost incurred only in the first iteration of baseline WCC.

7.3 Experiments

Figure 13 shows the results of our experiments adding SP to WCC. Because SP introduces randomness, we repeated each experiment three times and report the average of our measurements. As shown, SP improves the run-time on average between 2.7x to 7.4x across all of our input graphs. Note because the giant component in our undirected graphs is very large, consisting of more than 90% of the vertices, in all of our trials SP detected the giant-component in the first try.

For SCC, SP can be used in combination with FCS-BT and FCS-SCC. To measure the additional benefits of SP, we repeated our experiments from Section 4.5.1 with SP. Figure 14 shows the results. We again ran each experiment three times and report the average run-times. SP yields between 1.1x to 2.1x additional run-time benefits. In all our trials SP detected the giant component within two tries. We note that the improvements were modest on our web graphs. The run-time of SCC is dominated by the large number of supersteps the Forward-Traversal phases execute due to large diameters of the web graphs. None of the optimization techniques we know of can avoid these superstep executions. Finally, we note that considering all our optimizations together (FCS-BT, FCS-SCC, and SP), the run-time of the baseline SCC algorithm improved between 1.45x and 3.7x.

8. RELATED WORK

We first review related work in optimizing computations on Pregel-like systems. Then we discuss related work on the algorithms we cover in this paper.

References [34, 18, 42] describe graph partitioning techniques for assigning graph vertices to machines, with the goal of reducing the communication cost of algorithms. These techniques are effective when vertices send the same message to all of their neighbors in each phase of the algorithm. However, these techniques are less suitable when vertices sometimes communicate with a single neighbor or supervertex, such as in the SCC, MSF, MWM, and GC algorithms we cover in this paper. Combining messages [12, 35] is another technique that can be used to reduce the communication cost of algorithms. This technique can be applied only when vertices aggregate their messages using the same commutative and associative function in every phase—a condition that does not hold for most of the algorithms we cover in this paper (except WCC). The SP optimization was introduced originally in [41] for finding weakly-connected components in undirected graphs. They evaluate the optimization on two small synthetic graphs that exhibit skew. They report minor performance benefit for one graph and minor performance loss for the other. We report significant performance improvements when finding both strongly and weakly-connected components in large graphs.

A variety of parallel algorithms exist for the graph problems we cover in this paper: SCC [15, 44], MSF [13, 29], MWM [14, 40], and GC [16, 24]. Some of these algorithms are designed for the PRAM model and are not suitable for vertex-centric implementations, and some have been observed to not perform well on large-scale real graphs [22, 44]. To the best of our knowledge, none have been implemented on Pregel-like systems.

A variety of other graph algorithms have been implemented on Pregel-like systems. The original Pregel paper [35] describes several algorithms including PageRank, single-source shortest paths to all vertices, a randomized maximal matching in a bipartite graph, and an algorithm to cluster similar vertices in a graph. Reference [41] describes Pregel implementations of several social network analysis algorithms: computing the degrees of separation and the clustering coefficients of vertices, computing the diameter of a graph, and finding the k-cores, triangles, and k-trusses in a graph. Reference [23] describes Pregel implementations of two other algorithms used in social network analysis: computing the conductance [26] of a graph and an approximation algorithm to compute the betweenness centrality [9] of vertices in a graph. Aside from the SP optimization in [41], none of these references describe algorithm-level optimizations to their baseline implementations.

9. FUTURE WORK

In our thorough algorithm implementations for this paper, we observed that some phases, such as trimming, propagation of values until convergence, and constructing the transpose of the input graph, appear across several different algorithms. We are in the process of developing a library of basic graph operations that can be reused across algorithms on Pregel-like systems. Our library consists of the vertex-centric subroutines of our graph operations, which can be called directly inside *compute()* functions. We plan to integrate the optimization techniques from this paper into our implementation of these subroutines to make executions more efficient. Then we plan to develop a higher-level language for specifying Pregel-like computations, whose primitives are the graph operations in our library. Such a language would be analogous to the Pig language for specifying MapReduce computations [**?**], which has proven very successful. Our optimization techniques can also

be used when compiling this high-level language into native *compute()* functions.

10. REFERENCES

[1] F. N. Afrati, A. D. Sarma, S. Salihoglu, and J. D. Ullman. Upper and Lower Bounds on the Cost of a Map-reduce Computation. *Proceedings of the VLDB Endowment*, 6(4), Feb. 2013.

[2] F. N. Afrati and J. D. Ullman. Optimizing multiway joins in a map-reduce environment. *IEEE Transactions on Knowledge and Data Engineering*, 23(9), 2011.

[3] J. Barnat, P. Bauch, L. Brim, and M. Ceska. Computing Strongly Connected Components in Parallel on CUDA. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, 2011.

[4] J. Barnat, J. Chaloupka, and J. C. van de Pol. Distributed Algorithms for SCC Decomposition. *Journal of Logic and Computation*, 2009.

[5] P. Beame, P. Koutris, and D. Suciu. Communication Steps for Parallel Query Processing. In *Proceedings of the 32Nd Symposium on Principles of Database Systems*, 2013.

[6] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. UbiCrawler: A Scalable Fully Distributed Web Crawler. *Software: Practice & Experience*, 34(8), 2004.

[7] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the International Conference on World Wide Web*, 2011.

[8] P. Boldi and S. Vigna. The Web Graph Framework I: Compression Techniques. In *Proceedings of the International Conference on World Wide Web*, 2004.

[9] U. Brandes. A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology*, 25(2), 2001.

[10] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph Structure in the Web. *Computer Networks*, 33(1-6), 2000.

[11] S. Chung and A. Condon. Parallel Implementation of Boruvka's Minimum Spanning Tree Algorithm. In *Proceedings of the International Parallel Processing Symposium*, 1996.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the Symposium on Operating System Design and Implementation*, 2004.

[13] F. Dehne and S. Götz. Practical Parallel Algorithms for Minimum Spanning Trees. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, 1998.

[14] F. Manne and R. H. Bisseling. A Parallel Approximation Algorithm for the Weighted Maximum Matching Problem. In *In Proceedings of the International Conference on Parallel Processing and Applied Mathematics*, 2007.

[15] L. Fleischer, B. Hendrickson, and A. Pinar. On Identifying Strongly Connected Components in Parallel. In *Proceedings of the IPDPS Workshops on Parallel and Distributed Processing*, 2000.

[16] A. H. Gebremedhin and F. Manne. Scalable Parallel Graph Coloring Algorithms. *Concurrency - Practice and Experience*, 12(12), 2000.

[17] Apache Incubator Giraph. http://incubator.apache.org/giraph/.

[18] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2012.

[19] GPS Source Code. https://subversion.assembla.com/svn/phd-projects/gps/trunk.

[20] Apache Hadoop. http://hadoop.apache.org/.

[21] J. M. Hellerstein. The Declarative Imperative: Experiences and Conjectures in Distributed Logic. *SIGMOD Record*, 39(1), 2010.

[22] S. Hong, N. C. Rodia, and K. Olukotun. On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-World Graphs. Technical report, Stanford University, March 2013. http://ppl.stanford.edu/papers/techreport2013_hong.pdf.

[23] S. Hong, S. Salihoglu, J. Widom, and K. Olukotun. Tech Report: Compiling Green-Marl into GPS. Technical report, Stanford University, October 2012. http://ppl.stanford.edu/papers/tr_gm_gps.pdf.

[24] J. R. Allwright and R. Bordawekar and P. D. Coddington and K. Dincer and C. L. Martin. A Comparison of Parallel Graph Coloring Algorithms. Technical report, Syracuse University, 1995. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.45.4650.

[25] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System – Implementation and Observations. In *Proceedings of the IEEE International Conference on Data Mining*, 2009.

[26] R. Kannan, S. Vempala, and A. Veta. On Clusterings: Good, Bad and Spectral. In *Symposium on Foundations of Computer Science*, 2000.

[27] D. R. Karger. Global Min-Cuts in RNC, and Other Ramifications of a Simple Min-Cut Algorithm. In *Proceedings of the Symposium on Discrete Algorithms*, 1993.

[28] H. Karloff, S. Suri, and S. Vassilvitskii. A Model of Computation for MapReduce. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, 2010.

[29] I. Katriel, P. Sanders, and J. L. Traeff. A Practical Minimum Spanning Tree Algorithm Using the Cycle Property. In *Proceedings of the European Symposium on Algorithms*, 2003.

[30] Koutris, P. and Suciu, D. Parallel Evaluation of Conjunctive Queries. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 2011.

[31] The Laboratory for Web Algorithmics. http://law.dsi.unimi.it/datasets.php.

[32] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph Evolution: Densification and Shrinking Diameters. *ACM Transactions on Knowledge Discovery from Data*, 1(1), 2007.

[33] M. Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *SIAM Journal on Computing*, 15(4), 1986.

[34] A. Lugowski, D. Alber, A. Buluç, J. R. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis. A Flexible Open-source Toolbox for Scalable Complex Graph Analysis. In *Proceedings of the SIAM Conference on Data Mining*, 2012.

[35] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2011.

[36] METIS Graph Partition Library. http://exoplanet.eu/catalog.php.

[37] J. Nešetřil, E. Milková, and H. Nešetřilová. "Otakar Borůvka on Minimum Spanning Tree Problem Translation of Both the 1926 Papers, Comments, History". *"Discrete Mathematics "*, 233(1–3), 2001.

[38] S. M. Orzan. *On Distributed Verification and Verified Distribution*. PhD thesis, Free University of Amsterdam, 2004. https://www.cs.vu.nl/en/Images/SM%20Orzan%205-11-2004_tcm75-258582.pdf.

[39] O. Phelan, K. McCarthy, and B. Smyth. Using Twitter to Recommend Real-time Topical News. In *Proceedings of the ACM Conference on Recommender Systems*, 2009.

[40] R. Preis. Linear Time 1/2-Approximation Algorithm for Maximum Weighted Matching in General Graphs. In *Proceedings of the Symposium On Theoretical Aspects of Computer Science*, 1998.

[41] L. Quick, P. Wilkinson, and D. Hardcastle. Using Pregel-like Large Scale Graph Processing Frameworks for Social Network Analysis. In *Proceedings of the International Conference on Advances in Social Networks Analysis and Mining*, 2012.

[42] S. Salihoglu and J. Widom. GPS: A Graph Processing System. In *Proceedings of the International Conference on Scientific and Statistical Database Management*, 2013.

[43] M. Sharir. A Strong-connectivity Algorithm and Its Applications in Data Flow Analysis. *Computers & Mathematics with Applications*, 7(1), 1981.

[44] T. H. Spencer. Time-work Tradeoffs for Parallel Algorithms. *Journal of the ACM*, 44(5), 1997.

[45] S. Vadapalli, S. R. Valluri, and K. Karlapalem. A Simple Yet Effective Data Clustering Algorithm. In *Proceedings of the IEEE International Conference on Data Mining*, 2006.

[46] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8), 1990.

[47] C. Zhong, D. Miao, and R. Wang. A Graph-theoretical Clustering Method Based on Two Rounds of Minimum Spanning Trees. *Pattern Recognition*, 43(3), 2010.