Beyond Synchronous: New Techniques for External-Memory Graph Connectivity and Minimum Spanning Forest

Aapo Kyrola, Julian Shun, and Guy Blelloch

Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213, USA
{akyrola,jshun,guyb}@cs.cmu.edu

Abstract. GraphChi [16] is a recent high-performance system for external memory (disk-based) graph computations. It uses the Parallel Sliding Windows (PSW) algorithm which is based on the so-called *Gauss-Seidel* type of iterative computation, in which updates to values are immediately visible within the iteration. In contrast, previous external memory graph algorithms are based on the *synchronous* model where computation can only observe values from previous iterations. In this work, we study implementations of connected components and minimum spanning forest on PSW and show that they have a competitive I/O bound of $O(\text{sort}(E)\log(V/M))$ and also work well in practice. We also show that our MSF implementation is competitive with a specialized algorithm proposed by Dementiev et al. [10] while being much simpler.

1 Introduction

Research on external-memory graph algorithms was an active field in the 1990s and early 2000s, however not much work has been done on external-memory algorithms for fundamental graph problems since then. Recently, fueled by the interest to study large social networks and other massive graphs, there has been renewed interest for large-scale disk-based graph computation. In 2012, Kyrola et al. proposed GraphChi [16], which uses the *Parallel Sliding Windows* (PSW) algorithm for external-memory graph computation with the vertex-centric programming model. More recently, alternative solutions have been proposed, most notably X-Stream [20] and TurboGraph [12]. TurboGraph works efficiently only on modern Solid State Disks (SSD) that can support hundreds of thousands of random disk accesses per second, while GraphChi and X-Stream work efficiently even on traditional spinning disks.

In this paper, we study how GraphChi's PSW technique can be used efficiently to solve classic problems of graph connectivity and finding the minimum spanning forest (MSF) of a graph. Analyzing PSW is particularly interesting, since it expresses iterative computation using the so called Gauss-Seidel model (abbreviated G-S)[1] of computation in contrast to the Bulk Synchronous Parallel

[1] We borrow terminology from the study of iterative linear system solvers.

J. Gudmundsson and J. Katajainen (Eds.): SEA 2014, LNCS 8504, pp. 123–137, 2014.

(BSP) model of X-Stream. While in the BSP model program execution can only observe values computed on the previous iteration, in the G-S model the most recent value for any item is available for computation. We show by simulations and theoretical analysis that the G-S model can reduce the number of costly passes over the graph data significantly compared to BSP.

We show how to use PSW to write simple implementations of external-memory (EM) algorithms for connected components and MSF, in contrast to many previous algorithms that are difficult to implement. These algorithms take advantage of the G-S form of computation. We describe a problem called *minimum label propagation* (MLP) in which vertices update its identifier with the minimum of its identifier and its neighbors' identifiers. We show that a G-S implementation of this problem gives speedup over the traditional synchronous implementation, and use it to compute connected components and MSF. Our MSF algorithm is a graph contraction-based algorithm which repeatedly applies a single step of MLP. We prove that it terminates in a logarithmic number of iterations, and has an expected I/O bound of $O(\text{sort}(E)\log(V/M))$. We also show experimentally that it is competitive with the only available external-memory MSF implementation, while being much simpler. For connected components, we present two algorithms—a simple one based on MLP that requires a number of iterations proportional to the graph diameter, and one based on graph contraction (with the same I/O bound as for MSF). We show experimentally that for low-diameter graphs, the label propagation algorithm is competitive, while the contraction-based algorithm is efficient on general graphs.

2 Related Work

Chiang et al. [8] describe the first external-memory (deterministic) algorithm for MSF, and it has an I/O complexity bound of $O(\min(\text{sort}(V^2), \log(V/M)\text{sort}(E)))$. Kumar and Schwabe [14] give an improved deterministic algorithm with a bound of $O(\text{sort}(E)\log(B) + \log(V)\text{scan}(E))$. Arge et al. [3] give a deterministic algorithm requiring $O(\text{sort}(E)\log\log(B))$. The best I/O bound is for a randomized algorithm by Abello et al. [1] using $O(\text{sort}(E))$ I/O's with high probability. As MSF can be used for connected components, these bounds apply for external-memory connected components as well. There has been work on many other external-memory graph algorithms as well (see [13] for a survey).

As far as we know, the only experimental work on external-memory connected components is by Lambert and Sibeyn [17] and Sibeyn [21]. The implementations by Lambert and Sibeyn [17] do not have theoretical guarantees on general graphs and the implementation by Sibeyn [21] is only provably efficient for random graphs. For external-memory MSF, Dementiev et al. [10] present an implementation which requires $O(\text{sort}(E)\log(V/M))$ I/O's in expectation.

Convergence of iterative asynchronous computation has been studied in other settings, for example by Bertsekas and Tsitsiklis [5] for parallel linear system solving and Gonzalez et al. [11] in the context of probabilistic graphical models.

3 Preliminaries

A graph is denoted by $G = (V, E)$ where V is the set of vertices and E is a set of tuples (u, v) such that $u, v \in V$. When clear from the context, we also use V and E to refer to the number of vertices and number of edges in G, respectively.

I/O Model. Our analysis is based on the *I/O model* introduced by Aggarwal and Vitter [2]. The cost of a computation is the number of block transfers from external memory (disk) to main memory (RAM) or vice versa, and any computation that is done with data in RAM is assumed to be free. When modeling the I/O complexity, the following parameters are defined: N is the number of items in the problem instance, M is the number of items that can fit into main memory, and B is the number of items per disk block transfer. Fundamental primitives for I/O efficient algorithms are *scan* and *sort* (generalized prefix-sum). Their respective I/O complexities were derived by Aggarwal and Vitter [2]: $\text{scan}(N) = O(N/B)$ and $\text{sort}(N) = O((N/B)\log_{(M/B)}(N/B))$.

Jacobi (synchronous) and Gauss-Seidel (asynchronous) Computation. We now define formally the semantics of the two different models of computation. We borrow terminology from the numerical linear algebra literature since the commonly used terms "synchronous" and "asynchronous" have various other meanings in the context of numerical computation. In the following definitions, we denote by $x_i(t) \in A$ to be the value of a variable x_i (associated with an edge or vertex indexed by i) after iteration $t \geq 1$; $x(0)$ is the initial value of x and A is the domain of the variables.

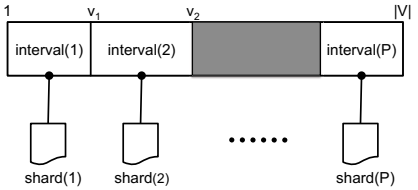
Definition 1. Jacobi (Synchronous) Model: *A function F that computes $x_i(t)$ depends only on values from the previous iteration, i.e. $F(v) := F(\{x_j(t-1)\}_{\forall j}) \to A$.*

To define the Gauss-Seidel computation, we define **schedule**, which determines the order of vertex "updates" in a vertex-centric computation.

Definition 2. *Let $\pi := V \to \{1, 2, ...|V|\}$ be a bijective function. Then $\pi(v)$ defines an update* **schedule** *so that if $\pi(u) < \pi(v)$, then vertex u will be updated before vertex v.*

The simplest schedule $\pi(v) = v$ updates vertices in the order they are labeled and is the default schedule used in GraphChi. In this paper, we study random schedules where π is a uniformly random permutation. We now give a definition of G-S computation that extends the traditional definition with a schedule:

Definition 3. Gauss-Seidel (Asynchronous) Model: *A function F that computes $x_i(t)$ uses the most recent values of its dependent variables. That is, $F(v) := F(\{x_i(t)\}_{i|\pi(i)<\pi(v)} \cup \{x_j(t-1)\}_{j|\pi(j)>\pi(v)}) \to A$.*




```
1: procedure PSW(G, updateFunc)
2:     for interval I_i ⊂ V do
3:         G_i := LoadSubgraph(I_i)
4:         for v ∈ G_i.V do
5:             updateFunc(v, G_i.E[v])
6:         UpdateToDisk(G_i)
```

Fig. 1. Left: The vertices of graph (V, E) are divided into P intervals. Each interval is associated with a shard, which stores all edges that have destination vertex in that interval. Right: Pseudo-code for the main loop of Parallel Sliding Windows. Note that both for-loops can iterate in random order.

Parallel Sliding Windows. We now introduce the framework used to implement the algorithms in this paper. Parallel Sliding Windows (PSW) is based on the *vertex-centric* model of computation [16]. The state of the computation is encapsulated in the graph $G = (V, E)$, where $V = \{0, 1, \ldots, |V| - 1\}$, E is a set of ordered[2] tuples (src, dst) such that $src, dst \in V$. We associate a value (data) with each vertex and edge, denoted by d_v and d_e respectively. PSW executes programs that are presented as imperative vertex *update-functions* with the form: `updateFunc`$(v, E[v])$. This function is passed a vertex v, and arrays of the in- and out-edges of the vertex (denoted as $E[v]$, where $E[v] = \{(src, dst) \mid src \in V \vee dst \in V\}$). The vertex data and the data for its incident edges are accessible via pointer. Values of other vertices cannot be accessed. The update function is executed on each vertex in turn (under some schedule), with Gauss-Seidel semantics, i.e. changes to edge values are immediately visible to subsequent updates.

PSW executes the programs on a sequence of (partial) subgraphs $G_i \subset G$, $i \in \{1, .., P\}$, where each subgraph fits into memory. Each subgraph contains vertices of a continuous interval I_i of vertices: $I_1 = \{1, \ldots, a_1\}, \{a_1+1, \ldots, a_2\}, \ldots, I_P = \{a_{P-1}, \ldots, N\}$ so that $\bigcup_{i=1..P} I_i = V$ and $\forall i \neq j,\ I_i \cap I_j = \emptyset$. In addition to vertex values, each subgraph contains all the edges of those vertices[3]. After loading a subgraph, PSW executes the update function on the vertices and then writes the changes to vertex and edge values back to disk (see the pseudo-code in Fig. 1). Note that the order of processing the subgraphs, as well as the order that the update function is invoked on the vertices of the subgraph, can be arbitrary.

We now describe how PSW stores the edges on disk, and how the vertex intervals I_i are defined. Each interval I_1 is associated with a file, called `shard(i)`. `shard(i)` stores all the *in-edges* (and their associated values) of vertices in interval I_i (see Fig. 1). Moreover, the edges in a shard are stored in sorted order based on their source vertex. The size of a shard must be less than the available memory M, and is typically set to $M/4$. To create the shards, we first sort all the edges based on their destination vertex ID, with an I/O cost of sort(E).

[2] For undirected graphs, we simply ignore the direction and it can be chosen arbitrarily.
[3] The subgraph is partial since it does not include vertex values for neighbors outside of the interval.

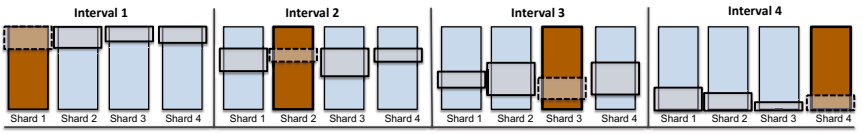


Fig. 2. Visualization of one iteration of PSW. In this example, vertices are divided into four intervals, each associated with a shard. The computation proceeds by constructing a subgraph of vertices one interval at a time. In-edges for the vertices are read from the corresponding shard (in dark color) while out-edges are read from all of the shards.

Then we create one shard at a time by scanning the sorted edges from the beginning, and add edges to a new shard until it reaches its maximum size (after which the shards are sorted in-memory prior to storing on disk). The vertex intervals and P are thus defined dynamically during the preprocessing phase. The second phase has an I/O cost of scan(E).

Each shard is split into two components: an immutable *adjacency shard* and a mutable *data shard*. Edges are stored in the adjacency shard as follows: for each vertex u that has out-edges stored in the shard, we write u followed by its number of edges. For each edge we store the neighbor vertex ID. The edge values are stored in the data-shard as a flat array A so that $A[i]$ has the value of i^{th} edge. For each interval we also store a flat array on disk containing the vertex values for the vertices in the interval.

We now describe how the edges for a subgraph g_i are loaded from disk. First, the in-edges (and their values) for vertices in interval I_i are all contained in `shard(i)`. Thus, that shard is loaded completely into memory. Secondly, the out-edges of the subgraph are contained in contiguous blocks in each of the shards, since we had sorted the edges by their source ID in the preprocessing phase. Note that we can easily store in memory the boundaries for the out-edges in each shard. After loading the blocks containing the edge adjacencies and values, we construct the subgraph in memory so that each edge is associated with a pointer to the location in a block that was loaded from disk, so all modifications are made directly to the data blocks. We can then execute the update function on each of the vertices in interval I_i. After finishing the updates, the edge data blocks are rewritten back to disk replacing the old data. Thus, all changes are immediately visible to subsequent updates (for the next subgraph), giving us Gauss-Seidel semantics (note that updates within each subgraph are done in a G-S manner in memory). This process is illustrated in Fig. 2.

Theoretical Properties of PSW. PSW can process any graph that fits on disk. The original GraphChi paper [16] states that there must be enough memory to store any one vertex and its edges, but we later describe in Section 5 how to get around this limitation. As described in [16], one iteration (pass) over the graph, in which both in- and out-edges of vertices are updated, has cost $PSW(G)$, where $2(E/B) + (V/B) \leq PSW(G) \leq 4(E/B) + (V/B) + \Theta(P^2)$.

For the $\Theta(P^2)$ term to not dominate the cost of PSW, we assume that $E < M^2/(4B)$ (recall $P = 4E/M$). This condition is easily satisfied in practice as a typical value of M for a commodity machine is around 8 GB, a typical value of B is on the order of kilobytes and current available graph sizes are less than a petabyte. Many algorithms only modify out-edges, and read in-edges in which case the I/O complexity is only $PSW(G)/2$. PSW also requires a preprocessing step for creating the shards, which has an I/O cost of sort(E). Note that due to the assumption $E < M^2/(4B)$, we have that $\text{sort}(E) = O((E/B)\log_{M/B}(M^2/(4B))) = O(E/B)$.

Graph Contraction with PSW. We can implement graph contraction under the PSW framework by allowing update-functions to write edges to a file (in a buffered manner). After the computation is finished, we create the contracted graph from the emitted edges and remove duplicate edges in the process. This has same cost as the preprocessing of the graph, which is sort(E). We can then continue executing PSW on the newly created contracted graph.

4 Minimum Label Propagation

We use *minimum label propagation (MLP)* as a subroutine in our minimum spanning forest algorithm. It can also be used directly to compute the connected components of a graph. The algorithm works by initializing each vertex with a label equal to its ID, and on each iteration, updating (in random order) the vertices' labels with the smallest of its neighbors' labels and its own labels.

The pseudo-code for computing connected components with MLP is shown in Algorithm 1. With PSW, and in the external-memory setting, vertices must communicate their labels via edges. Each edge (u, v) has two fields, *leftLabel* and *rightLabel*, where leftLabel contains the label of the smaller of u and v, and rightLabel contains the label of the other vertex. The functions in lines 1–6 implement this logic. The vertex values are initialized with the vertex IDs on line 16. We note that if MLP is run until convergence (i.e. until all edges "agree"), vertices in the same connected component will have the same label (in fact, if MLP is run until convergence, each edge can store only one field as both vertices will eventually have the same label).

Our algorithm executes the `MLPUpdate` update function (lines 8–14) using PSW as long as any vertex changes its label during an iteration (lines 18–20). `MLPUpdate` is passed a vertex and its edges. On line 11, it finds the minimum label written to its incident edges by its neighbors. If that label is smaller than the previous minimum label, it is written to all adjacent edges on line 14.

I/O Complexity. We analyze the number of iterations of MLP required until all connected vertices in each connected component of a graph share the same label. We refer to the the ***distance*** $\text{dist}(u, v)$ between vertices $u, v \in V$ as the length of the path from u to v with the fewest number of edges. If u and v are not connected, then $\text{dist}(u, v)$ is undefined. The ***diameter*** of a graph, denoted as D_G, is the maximum $\text{dist}(u, v)$ between any two connected vertices $u, v \in V$.

Algorithm 1. Minimum-label Propagation (MLP)

```
1: function GetNeighborLabel(vertex, edge)
2:     if vertex.ID < edge.ID then return edge.rightLabel
3:     else return edge.leftLabel
4: function SetMyLabel(vertex, edge, newLabel)
5:     if vertex.ID < edge.ID then edge.leftLabel = newLabel
6:     else edge.rightLabel = newLabel
7: global var changed
8: procedure MLPUpdate(vertex, vertexedges)
9:     var minLabel = vertex.label
10:    for edge ∈ vertexedges do
11:        minLabel = min(minLabel, GetNeighborLabel(vertex, edge))
12:    if vertex.label ≠ minLabel then changed = true
13:        vertex.label = minLabel
14:        for edge ∈ vertexedges do SetMyLabel(vertex, edge, minLabel)
15: procedure RunMLP(G)
16:    for vertex ∈ G.V do vertex.label = vertex.ID
17:    changed = true
18:    while changed = true do
19:        changed = false
20:        PSW(G, MLPUpdate)
```

The following lemma states the number of iterations a synchronous computation requires for convergence, and its proof is straightforward.

Lemma 1. *Let the vertex with the minimum identifier be v_{min}. Then the Jacobi (synchronous) computation of minimum label propagation requires exactly* $\max_{v \in V} dist(v, v_{min}) \leq D_G$ *iterations to converge.*

Clearly, the Gauss-Seidel model requires at most as many iterations as the synchronous model of computation so the I/O complexity is at most $D_G \times PSW(G) = O(D_G(V + E)/B)$. But with G-S, the computation can converge in fewer iterations because on each iteration the minimum label can propagate over multiple edges. We can study this analytically on a simple chain graph: Let C_n be a chain graph with n vertices $V_n = \{1, 2, ..., n\}$ and $n - 1$ edges $E_n = \{(1, 2), (2, 3), ..., (n - 1, n)\}$.

Theorem 1. *On a chain graph C_n, the expected number of iterations required for the Gauss-Seidel computation for convergence of MLP is* $(n - 1)/(e - 1) \approx 0.582(n - 1)$. *The synchronous computation requires exactly $n - 1$ iterations.*

Proof. The smallest label is 1 ("minimum label"), and on each iteration the label "advances" one or more steps towards the end of the chain. In the beginning of an iteration, let u be the vertex furthest from vertex 1 (the beginning of the chain) that has already assigned label 1. Vertex $u + 1$ will receive label 1 after it is updated. Now, if $\pi(u + 2) > \pi(u + 1)$, also vertex $u + 2$ will receive label 1 during the same iteration. Similarly, if $\pi(u + 3) > \pi(u + 2) > \pi(u + 1)$, the label

reaches $u+3$, and so on. We see that the probability that the minimum label advances exactly k steps is the probability of a permutation $\pi(u+1), \ldots, \pi(u+k), \pi(u+k+1)$ where the permutation is ascending from $\pi(u+1)$ to $\pi(u+k)$ but $\pi(u+k+1) < \pi(u+k)$. The probability of such a permutation is $\frac{k}{(k+1)!}$. Let X be the random variable denoting the number of steps the minimum label advances in the chain during one iteration. Then for large n, $\mathbb{E}[X]$ approaches $\sum_{k=1}^{\infty} \frac{k^2}{(k+1)!} = e - 1$. The theorem follows from this and Lemma 1.

5 Minimum Spanning Forest and Graph Contraction

Previous algorithms for computing the minimum spanning forest (MSF) in the external-memory setting use different variations of graph contraction to recursively solve the problem. We implement a variation of Boruvka's algorithm [7] on PSW, based on the MLP algorithm. On each iteration, Boruvka's algorithm selects the minimum weight edge of each vertex. These minimum edges are surely part of the minimum spanning forest, and the induced graph consisting only of these minimum edges is a forest. In Boruvka's algorithm, each tree is contracted into one vertex, edges are relabeled accordingly and the computation is repeated on the contracted graph. Each edge in the contracted graph contains information of its identity in the original graph so that the MSF edges can be identified.

Min-Label Contraction (MLC) Algorithm. The MLP algorithm described in previous section can be used to implement graph contraction: Let (x, y) be the labels stored on edge $e = (u, v)$ after one or more iterations of MLP. Then, we output edge (x, y) for the contracted graph, unless $x = y$. If there are multiple copies of an edge (x, y) that are output for the new graph, they are merged into one edge. The number of vertices in the new graph is equal to the number of distinct labels at the end of last MLP iteration. See the description at the end of Sec. 3 for details on how the contraction step is implemented.

MSF: One-Iteration Min-Label Contraction on a Forest. The pseudocode for our MSF algorithm is shown in Algorithm 2. A super-step of the algorithm (lines 19–23) consists of three invocations of PSW. The first PSW executes the ChooseMinimum update function (lines 2–4), which marks the minimum weighted edge of each vertex by setting the `inMSF` field of the edge data. These minimum edges are part of the MSF, and induce a collection of subtrees (a forest) on the graph. The second PSW execution is similar to the MLP algorithm (Algorithm 1), but the minimum labels are selected only among edges that have `inMSF` set to true. On line 19, we initialize a new graph and then output relabeled edges to the new graph. The third PSW execution contracts the graph and writes it to file by calling the update function ContractionStep (lines 11-16). Finally, on line 23 we preprocess the new graph into shards that can be used for PSW on the next iteration. Note that we have omitted details on keeping track of the original identity of each edge.

In contrast to Boruvka's algorithm, our algorithm runs only one iteration of propagation and contraction per super-step. This does not guarantee that the

Algorithm 2. Minimum Spanning Forest using PSW

```
 1: global var outfile
 2: procedure ChooseMinimum(vertex, vertexedges)
 3:     var minEdge = [find minimum weighted edge of vertexedges]
 4:     minEdge.inMSF = true
 5: procedure MinimumLabelPropOneIter(vertex, vertexedges)
 6:     var minLabel = vertex.value
 7:     for edge ∈ vertexedges do
 8:         if edge.inMSF then
 9:             minLabel = min(minLabel,GetNeighborLabel(vertex, edge))
10:     for edge ∈ vertexedges do  SetMyLabel(vertex, edge,vertex.label)
11: procedure ContractionStep(vertex, vertexedges)
12:     for e ∈ vertexedges do
13:         if e.dst = vertex.ID then
14:             if e.leftLabel ≠ e.rightLabel then
15:                 writeEdge(outfile, e.leftLabel, e.rightLabel, e.value)
16:             if e.inMSF then outputToMSFFile(e)
17: procedure RunMSF(G)
18:     while |G.E| > 0 do
19:         PSW(G, ChooseMinimum)
20:         PSW(G, MinimumLabelPropOneIter)
21:         outfile = [initialize empty file]
22:         PSW(G, ContractionStep)
23:         G = PreprocessNewGraph(outfile)
```

trees will be completely contracted, but we will derive a lower bound on the number of vertices contracted on each step. In the G-S setting we assume that the unique labels are adversarial but the schedule of the vertices is random. Denote the label of a vertex v at the beginning of an iteration by $l(v)$. We only need to consider the min-label contraction problem on a tree. We want to show that a constant number of vertices will be contracted in each iteration, which allows us to bound the number of iterations of MSF by $O(\log(V/M))$.

Fact 1. *The number of degree-one (leaves) and degree-two vertices in a tree of V nodes is at least $V/3$.*

By Fact 1, we only need to consider the expected number of leaves and degree-two vertices contracted. In our algorithm, all vertices with the same label will be contracted into one. If a vertex ends up with the same label as a neighbor, at least one of the two will be contracted. Among the vertices with the same label, the vertex that is contracted can be chosen randomly. So on average, a vertex with the same label as a neighbor is contracted with at least $1/2$ probability.

Lemma 2. *For a tree with V_1 leaves, the expected number of leaves contracted is at least $V_1/4$.*

Proof. Consider the ID of a leaf v and its neighbor w. There are two cases: (1) $l(w) < l(v)$ and (2) $l(v) < l(w)$. In case (1), if $\pi(w) < \pi(v)$ then v will

get the same label as w. There is a 1/2 probability of the event $\pi(w) < \pi(v)$. In case (2), fix the ordering with respect to π for all vertices except v and w. Let $\pi(x) = \max(\pi(w), \pi(v))$. Consider the permutation from the start to $\pi(x)$ excluding $\pi(w)$ and $\pi(v)$. There are two sub-cases: (2a) the permutation causes the ID of w to become smaller than $l(v)$ after w is executed, and (2b) the permutation does not cause the ID of w to become smaller than $l(v)$ after w is executed. In case (2a), if $\pi(w) < \pi(v)$ then v will end up with the same label as w. In case (2b), if $\pi(v) < \pi(w)$ then w will end up with the same label as v. This is true because we know that all vertices before w do not reduce w's label to below $l(v)$. The probability of the desired ordering of $\pi(w)$ and $\pi(v)$ in either case (2a) or (2b) is 1/2 as the events $\pi(w) < \pi(v)$ and $\pi(v) < \pi(w)$ are equally likely. Thus for any initial labeling of the vertices, a leaf must fall into either case (1) or case (2), and have a 1/2 probability of having the same label as its neighbor. A leaf with the same label as its neighbor is contracted with at least 1/2 probability. By linearity of expectations, at least $V_1/4$ leaves are contracted.

Lemma 3. *For a tree with V_2 degree-2 vertices, the expected number of degree-2 vertices contracted is at least $V_2/6$.*

Proof. Consider a degree-2 vertex v with neighbors u and w. If $\pi(v) < \pi(u) < \pi(w)$ or $\pi(u) < \pi(v) < \pi(w)$ then w will not affect the resulting labels of v or u. Similarly if $\pi(v) < \pi(w) < \pi(u)$ or $\pi(w) < \pi(v) < \pi(u)$ then u will not affect the resulting labels of v or w. In either of these cases, we can consider u as a leaf and use the analysis of Lemma 2 because the neighbor that is after v in π does not affect whether v will be contracted. One of these two cases will happen with 2/3 probability. In the other orderings of u, v and w according to π we pessimistically assume that v is not contracted. Thus a degree-2 vertex will be contracted with at least $(2/3)\cdot(1/4) = 1/6$ probability. By linearity of expectations, the expected number of degree-2 vertices contracted is at least $V_2/6$.

By applying Fact 1 and Lemmas 2 and 3, we have the following theorem:

Theorem 2. *For a tree with V vertices, the number of vertices contracted in one iteration of min-label contraction is at least $V/18$.*

We note that our analysis applies for any (adversarial) labeling of the vertices.

Corollary 1. *The I/O complexity of our MSF algorithm is $O(sort(E)\log(V/M))$.*

Proof. By Theorem 2, after at most $\log_{18} V - \log_{18} M = O(\log(V/M))$ iterations, the number of vertices remaining will be at most M, at which point we can switch to a semi-external algorithm. Each iteration of the MSF algorithm requires $O(\text{sort}(E))$ I/O's. The result follows.

Our I/O complexity is worse than the $O(\text{sort}(E))$ bound of Abello et al. [1], but matches the bound of the only available external-memory MSF implementation by Dementiev et al. [10].

Dealing with Very High Degree Vertices. The original version of PSW requires any vertex and its edges to fit in memory. With the contraction procedure described earlier, it is possible that a vertex in the contracted graph gets many edges, possibly more than $O(M)$. We can address this issue by storing such high-degree vertices in their own shards. To find the minimum weighted edge or a neighbor ID, the order of the edges does not matter, so we can process such shards in parts, such that each part fits in memory. Since such shards only store edges for one vertex, this does not affect G-S semantics, and the I/O complexity bounds remain unchanged. Note that using this procedure, we can even remove the degree restriction on vertices in the *original* graph.

Connected Components. We can use the one-iteration MLC algorithm to compute connected components also. Instead of choosing the edge with minimum weight, each vertex chooses the neighbor with the minimum ID. During contraction, for each contracted vertex we keep pairs $(v, p(v))$ where $p(v)$ is the ID of its neighbor. On the way back up from the recursion we can relabel each vertex to be the same as its neighbor's label. Since the labels for the remaining are computed recursively we also have a list of pairs for them. This can be done by sorting the contracted vertex pairs by $p(v)$ and the remaining vertex pairs by v, and scanning them in parallel as done in [1]. The cost is $O(\text{sort}(V) + \text{scan}(V))$ per iteration, which is within our complexity bounds stated in Corollary 1.

6 Experiments

We use the following real-world and synthetic graphs in our experiments. *twitter_rv* is a graph of the Twitter social network with 41.7M vertices and 1.47B edges [15]. *uk-2007-05* is a subset of the UK WWW-network with 105M vertices and 3.8B edges [6]. The *web-Google* graph is a small web-graph with 0.5M vertices and 5M edges [18]. The *live-journal* graph is a social network graph with 5M vertices and 68M edges [4]. The first two are among the largest real-world graphs publicly available. We use synthetic k-grid graphs—each of the k-dimensional grids contain 100^k vertices, where each vertex has an edge to each of its 2^k neighbors. The *chain* graph is a 1-D grid graph.

Min-label Propagation Simulations. To our knowledge, it remains an open question to obtain a closed-form expression for the number of Gauss-Seidel iterations for MLP convergence on general graphs. Fortunately, it is simple to run simulations on arbitrary graphs and in Fig. 3, we show results on our input graphs. For the G-S computation, we randomized the schedule.

We see that the G-S computation always requires fewer iterations than the synchronous computation. The highest speedup in terms of number of iterations G-S achieves over Jacobi iterations is a 10-fold speedup on the 4d-grid. We note that on the grids, the advantage of G-S improves when the dimensionality (and thus the average number of edges) increases. Our intuition for this phenomenon is that between any pair of vertices, the number of possible paths for the minimum label to propagate increases rapidly with the average degree of vertices.

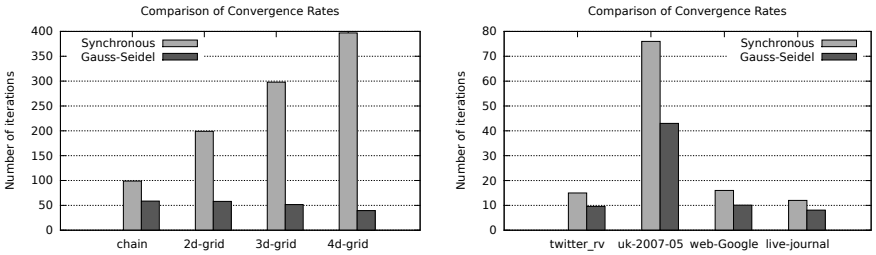


Fig. 3. Number of iterations required for convergence for synchronous and Gauss-Seidel label propagation on various graphs

Graph Contraction Simulations. In Theorem 2 we proved that at least a $V/18$ fraction of the vertices contract on each iteration. However, we found that the rate of contraction is much higher in practice. Although for MSF we only need to apply the one-iteration min-label contraction on trees, we conjecture that our contraction technique also works well on general graphs. We confirm the efficiency of the contraction technique by simulation. We compare our algorithm to Reif's random mate technique [19], where in each iteration vertices flip coins, and vertices that flipped "tails" pick a neighbor that flipped "heads" (if any) to contract with. Random mate gives the same I/O complexity as our algorithm since it contracts a constant fraction of the vertices per iteration. The simulation results are shown in Fig. 4. We see that for all input graphs, Gauss-Seidel under a randomized schedule achieves a better contraction rate (60–80%) rate than the synchronous version. The contraction rate observed is much higher than the bound indicated in Theorem 2. We see that both the synchronous and G-S versions of our algorithm achieve a higher contraction rate than random mate.

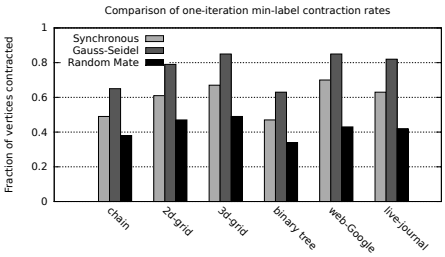


Fig. 4. The fraction of vertices contracted in one-iteration min-label contraction on various graphs. The binary tree has 4000 vertices.

Connected Components. We implemented two version of connected components on GraphChi (PSW). The first version runs the MLP algorithm until convergence. The implementation contains a simple optimization: if all vertices in a subgraph have converged, it is not reprocessed. This optimization requires only $O(P)$ memory. The second version is based on the one-iteration min-label

contraction technique described in Section 5. One optimization we used was that instead of inducing a collection of subtrees per iteration (as in MSF), we did propagation over all of the edges. We found this to work much better in practice. The timing results are shown in Fig. 5 (left). All experiments were run on a MacMini (2011) with an Intel I5 CPU, 8 GB of RAM, and a 1 TB rotational hard drive. The results clearly confirm that on low-diameter real-world graphs the MLP version works well, but with the grid that has very high diameter, the contraction-based algorithm is orders of magnitudes faster. On real-world graphs, the contraction algorithm is also competitive, and actually outperforms the MLP version on the large *uk-2007-05* web graph. Unfortunately, we could not compare with the implementation of Sibeyn [21] as it is unavailable.

Minimum Spanning Forest. Perhaps due to the difficulty of realizing the algorithms, the only other implemented external memory MSF algorithm is due to Dementiev et al. [10], which is available as an open-source implementation using STXXL [9]. In Fig. 5 (right) we show timing results on various graphs, using a 8-CPU AMD server with 32 GB of RAM and a rotational SCSI hard drive (we could not compile Dementiev's algorithm on the Mac Mini.). Based on the results, we see that neither algorithm is the clear winner, but the relative difference varies strongly between different graphs. This is not surprising since they employ different graph contraction techniques. This shows that our algorithm using PSW is competitive with a special-purpose MSF implementation. The advantage of our algorithm is that it is much simpler (requiring only tens of lines of simple code), being part of a general purpose framework.

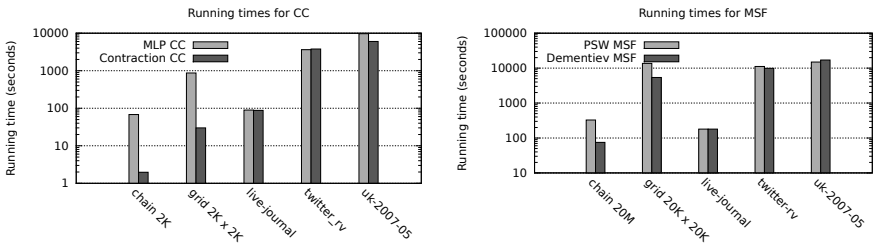


Fig. 5. Left: Timings for the MLP CC algorithm vs. contraction-based CC algorithm. Right: Timings for the PSW MSF algorithm vs. the Dementiev et al. [10] implementation. The numbers are averages—the standard deviations were less than 10%.

7 Conclusion

We have presented simple external-memory algorithms for connected components and minimum spanning forest implemented using GraphChi, and have proven an I/O complexity bound competitive with the only previous implementations for the problems. Our algorithms take advantage of the Gauss-Seidel

type of computation, which leads to an improvement in convergence rate over the synchronous type of computation used in previous external-memory algorithms for these problems. Parallel Sliding Windows is an exciting development in the research of external-memory graph algorithms as they provide a generic framework for designing and implementing simple and practical algorithms.

Acknowledgements. This work is supported by the National Science Foundation under grant number CCF-1314590. Kyrola is supported by a VMware Graduate Fellowship. Shun is supported by a Facebook Graduate Fellowship.

References

1. Abello, J., Buchsbaum, A.L., Westbrook, J.R.: A functional approach to external graph algorithms. Algorithmica 32(3), 437–458 (2002)
2. Aggarwal, A., Vitter, J., et al.: The input/output complexity of sorting and related problems. Commun. ACM 31(9), 1116–1127 (1988)
3. Arge, L., Brodal, G.S., Toma, L.: On external-memory MST, SSSP and multi-way planar graph separation. J. Algorithms 53(2), 186–206 (2004)
4. Backstrom, L., Huttenlocher, D., Kleinberg, J., Lan, X.: Group formation in large social networks: Membership, growth, and evolution. In: 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 44–54. ACM, New York (2006)
5. Bertsekas, D.P., Tsitsiklis, J.N.: Parallel and Distributed Computation: Numerical Methods. Prentice-Hall, Upper Saddle River (1989)
6. Boldi, P., Santini, M., Vigna, S.: A large time-aware web graph. ACM SIGIR Forum 42(2), 33–38 (2008)
7. Boruvka, O.: O jistem problemu minimalnim (about a certain minimal problem). In: Prace, Moravske Prirodovedecke Spolecnosti, pp. 37–58 (1926)
8. Chiang, Y.J., Goodrich, M.T., Grove, E.F., Tamassia, R., Vengroff, D.E., Vitter, J.S.: External-memory graph algorithms. In: 6th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 139–149. SIAM, Philadelphia (1995)
9. Dementiev, R., Kettner, L., Sanders, P.: STXXL: Standard template library for XXL data sets. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 640–651. Springer, Heidelberg (2005)
10. Dementiev, R., Sanders, P., Schultes, D., Sibeyn, J.: Engineering an external memory minimum spanning tree algorithm. In: Levy, J.-J., Mayr, E.W., Mitchell, J.C. (eds.) Exploring New Frontiers of Theoretical Informatics. IFIP, vol. 155, pp. 195–208. Springer, Heidelberg (2004)
11. Gonzalez, J., Low, Y., Guestrin, C.: Residual splash for optimally parallelizing belief propagation. In: International Conference on Artificial Intelligence and Statistics. pp. 177–184. JMLR (2009)
12. Han, W.S., Lee, S., Park, K., Lee, J.H., Kim, M.S., Kim, J., Yu, H.: TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In: 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 77–85. ACM, New York (2013)
13. Katriel, I., Meyer, U.: Elementary graph algorithms in external memory. In: Meyer, U., Sanders, P., Sibeyn, J. (eds.) Algorithms for Memory Hierarchies. LNCS, vol. 2625, pp. 62–84. Springer, Heidelberg (2003)

14. Kumar, V., Schwabe, E.J.: Improved algorithms and data structures for solving graph problems in external memory. In: 8th IEEE Symposium on Parallel and Distributed Processing, pp. 169–176. IEEE Press, New York (1996)
15. Kwak, H., Lee, C., Park, H., Moon, S.: What is Twitter, a social network or a news media? In: 19th International Conference on World Wide Web, pp. 591–600. ACM, New York (2010)
16. Kyrola, A., Blelloch, G., Guestrin, C.: GraphChi: Large-scale graph computation on just a PC. In: 10th USENIX Symposium on Operating Systems Design and Implementation, vol. 8, pp. 31–46. USENIX (2012)
17. Lambert, O., Sibeyn, J.F., Stadtwald, I.: Parallel and external list ranking and connected components. In: International Conference of Parallel and Distributed Computing and Systems, pp. 454–460. IASTED (1999)
18. Leskovec, J., Lang, K.J., Dasgupta, A., Mahoney, M.W.: Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. Internet Mathematics 6(1), 29–123 (2009)
19. Reif, J.H.: Synthesis of Parallel Algorithms. Morgan Kaufmann, San Francisco (1993)
20. Roy, A., Mihailovic, I., Zwaenepoel, W.: X-Stream: edge-centric graph processing using streaming partitions. In: 24th ACM Symposium on Operating Systems Principles, pp. 472–488. ACM, New York (2013)
21. Sibeyn, J.F.: External connected components. In: Hagerup, T., Katajainen, J. (eds.) SWAT 2004. LNCS, vol. 3111, pp. 468–479. Springer, Heidelberg (2004)