Accelerating Minimum Spanning Forest Computations on Multicore Platforms

Guojing Cong(✉), Ilie Tanase, and Yinglong Xia

IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA
{gcong,tanase,yxia}@us.ibm.com

Abstract. We propose new approaches for accelerating minimum spanning forest algorithms on shared-memory platforms. Our approaches improve cache performance and reduce synchronization overhead of the base algorithms. On our target platform these optimizations achieve up to an order of magnitude speedup over the best prior parallel *Borůvka* implementation.

Keywords: Minimum spanning forest · Locality · Synchronization

1 Introduction

Minimum spanning forest (MSF) and its special case minimum spanning tree (MST) are fundamental graph problems with practical applications (e.g., [3,10,17,18]). For MSF and MST, there exist a randomized time-work optimal algorithm and a deterministic logarithmic time algorithm on EREW PRAM [11,21], and a communication-optimal algorithm on BSP [1]. These theoretically fast algorithms have large constants in the asymptotic notation, and it is challenging to implement them for high performance. Moreover, these algorithms are not optimized for memory subsystem performance that is critical for modern architectures.

Recent experimental studies for MSF and related problems focus primarily on reducing the algorithmic overhead (e.g., see [4,6,20]). Implementations with more branches but fewer operations are shown to have performance advantages for the spanning tree (SF) and connected components (CC) problems (e.g., see [20]). Some breadth-first search (BFS) implementations optimize for the topology of specific inputs (e.g., low-diameter graphs) [5,7]. Agarwal *et al.* employ a bit-map data structure and optimize the locking mechanism for parallel BFS [2]. Hong *et al.* optimize the queues used in BFS [15] for bandwidth utilization. In general, these algorithms still exhibit random memory access behavior that results in poor memory subsystem performance. Some MSF implementations employ fine-grain synchronization with the number of locks scaling linearly with the input size. These implementations perform well on inputs of moderate sizes [12]. For large inputs they can exacerbate poor cache performance as their accesses are also random.

© Springer International Publishing Switzerland 2015
S. Hunold et al. (Eds.): Euro-Par 2015 Workshops, LNCS 9523, pp. 541–552, 2015.
DOI: 10.1007/978-3-319-27308-2_44

We consider improving locality and reducing synchronization to accelerate existing MSF implementations. Different from prior efforts that reduce the number of operations, the approaches we propose execute more instructions but with better locality. We propose three approaches that range from simple to sophisticated with different degrees of performance gain. The first approach implements graph contraction by updating the input data structure to improve locality. The second approach partitions the input edges and processes them in groups. The algorithm exhibits increasingly better locality as each group is processed. The third approach applies PRAM simulation on parallel memory accesses to remove locks and improve locality. Our optimization achieves up to an order of magnitude speedups over the base MSF implementation on our target platform.

We experiment with the most challenging types of graphs in terms of locality, that is, random graphs and scale-free graphs [14]. The input graph is represented as $G = (V, E)$, with $|V| = n$ and $|E| = m$. We create a random graph with n vertices and m edges by randomly adding m unique edges to the vertex set. Scale-free graphs are generated using the R-MAT model [9] with a = 0.45, b = 0.15, c = 0.15, d = 0.25. To complement these small diameter synthetic graphs, we also include six real-world networks from computer vision and social media. We defer their introduction to Sect. 6.

The rest of the paper is organized as follows. Section 2 introduces the base MSF algorithm that we optimize and our target platform. Section 3 presents the approach that compacts the input through edge updates. Section 4 introduces the meta algorithm that processes the edges in groups. Section 5 presents PRAM simulation that reduces synchronization and improves locality. Section 6 combines two meta approaches, and compares the performance of various implementations on both synthetic and real-world inputs. In Sect. 7 we give our conclusion and future work.

2 Base MSF Algorithm and Target Platform

For a weighted graph $G = (V, E)$, *Borůvka* start with n isolated vertices and m processors. Each processor inspects an edge $(u, v) \in E$, and if (u, v) has the minimum weight among all edges incident to u or v, (u, v) is labeled as an edge in the MSF. An edge (u, v) in the MSF causes grafting of one endpoint u to the other endpoint v or vice versa. Grafting creates $k \geq 1$ connected components in the graph, and each of the k components is then shortcut to a single super-vertex. One pass of graft and shortcut constitutes a *Borůvka* iteration. Grafting and shortcutting continue on the reduced graph $G' = (V', E')$ with V' being the set of super-vertices and E' being the set of edges among super-vertices until no grafting is possible.

Several implementations based on *Borůvka* are evaluated on symmetric multiprocessors by Bader and Cong [4]. Bor-AL employs parallel sort in *graft*, while Bor-FAL introduces a data structure that significantly reduces the cost of compacting the input. A hybrid algorithm is also proposed for MST that marries *Borůvka* with *Prim*. We choose a variant of *Borůvka* that uses locks [12] as our base MSF algorithm. It does not rely on other subroutines such as sort, and it uses roughly half of the memory consumed by Bor-AL and Bor-FAL. Its *Borůvka* iteration is shown in Algorithm 1. Due to limited space, for an edge (u, v), only grafting for vertex u is presented. In the algorithm, $I[i]$ and $Min[i]$, $1 \le i \le n$, represent the MSF edge (if any) incident to i and its weight, respectively. $D[i]$ is the supervertex that vertex i belongs to. At completion F contains the MSF edges found so far. Algorithm 2 shows the *Borůvka* algorithm.

Algorithm 1. *Borůvka*-iter(E, D)

```
 1: $F \leftarrow \emptyset$
 2: for $1 \le i \le n$ in parallel do
 3:     $Min[i] \leftarrow \infty$
 4: end for
    {graft}
 5: for each $e = (u, v) \in E$ in parallel do
 6:     lock($D[u]$)
 7:     if $D[u] \ne D[v]$ and $Min[D[u]] > w(e)$ then
 8:         $Min[D[u]] \leftarrow w(e)$
 9:         $D[D[u]] \leftarrow D[v]$
10:         $I[D[u]] \leftarrow \{e\}$
11:     end if
12:     unlock($D[u]$)
13: end for
14: for $1 \le i \le n$ in parallel do
15:     $F \leftarrow F \cup I[i]$
16: end for
    { shortcut }
17: for $1 \le i \le n$ in parallel do
18:     while $D[i] \ne D[D[i]]$ do
19:         $D[i] \leftarrow D[D[i]]$
20:     end while
21: end for
22: return $F$
```

Algorithm 2. *Borůvka* (E, D)

```
1: $F \leftarrow \emptyset$
2: for $1 \le i \le n$ in parallel do
3:     $D[i] \leftarrow i$, $I[i] \leftarrow \emptyset$
4: end for
5: repeat
6:     $F \leftarrow F \cup$ Borůvka-iter($E$, $D$)
7: until no grafting possible
8: return $F$
```

Algorithm 3. *Borůvka-updt* (E, D)

```
 1: $F \leftarrow \emptyset$
 2: for $1 \le i \le n$ in parallel do
 3:     $D[i] \leftarrow i$, $I[i] \leftarrow \emptyset$
 4: end for
 5: repeat
 6:     $F \leftarrow F \cup$ Borůvka-iter($E$, $D$)
 7:     for each $(u, v) \in E$ in parallel do
 8:         $(u, v) \leftarrow (D[u], D[v])$
 9:     end for
10: until no grafting possible
11: return $F$
```

Our target platform is an IBM P755 with four Power7 chips. Each chip has 8 cores running at 3.61GHz, with each core capable of four-way simultaneous multithreading. There are 12 execution units per core shared by the 4 hardware threads. Each core has 32KB L1, 256KB L2, and 4MB L3 caches.

Our experiments show that for large random graphs and scalefree graphs between 73 % and 81 % of machine cycles are wasted on cache misses for *Borůvka*, and only less than 1 % of time is spent on shortcut. Improving locality for *graft* can potentially reduce the execution times of *Borůvka*.

3 Update Edges for Locality

Accesses to D, *min* and I at lines 6–12 in Algorithm 1 are irregular. If (u, v) is the minimum-weight edge between the two components represented by $D[u]$

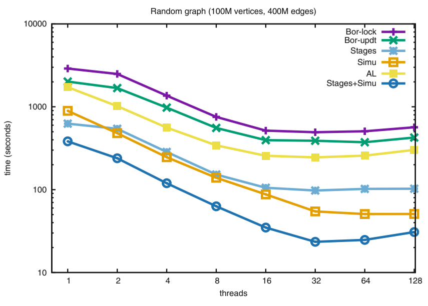

Fig. 1. Random graph

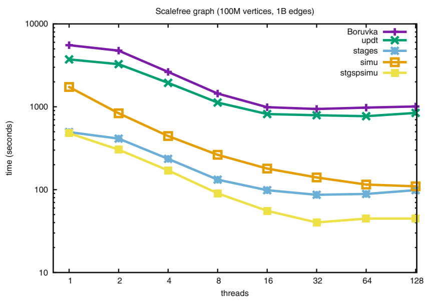

Fig. 2. Scalefree graph

and $D[v]$, the algorithm creates a union of the two by grafting one component to the other. While memory accesses to $D[u]$s, $D[v]$s and *etc.* determined by edges $(u,v) \in E$ are random, the D values evolve in a pattern that can be exploited for improving locality. In *Borůvka*, each iteration reduces the number of unique D values (at least by half for the largest connected component in the graph). Instead of retrieving the current components using u and v as indices, we introduce an *update* step after each *Borůvka* iteration that replaces each edge (u, v) with $(D[u], D[v])$. The revised algorithm *Borůvka-updt* is shown in Algorithm 3. The *update* step is done at lines 7–9.

The *update* step in *Borůvka-updt* increases the total number of operations and memory accesses in comparison to *Borůvka* (Algorithm 2). Indeed $2m$ extra memory accesses to D are issued at line 8 in each iteration. However, *update* makes accesses at lines 6–12 in Algorithm 1 increasingly more regular after each iteration. Indeed, the accesses at line 8 of Algorithm 3 themselves become more regular. This is because as the algorithm progresses, it becomes increasingly more likely for the two endpoints of an edge to touch on the same component (super-vertex).

We evaluate the performance improvement of *Borůvka-updt* over *Borůvka* on P755. The results with a random graph of 100 million (M) vertices and 400M edges and a scalefree graph of 100M vertices, 1 billion (B) edges are shown in Figs. 1 and 2, respectively. Speedups between 1.21 and 1.48 are achieved for the random graph, and speedups between 1.19 and 1.48 are achieved for the scalefree graph. The observed improvement is clearly due to better cache performance although more instructions are executed in *Borůvka-updt*.

4 Stages

Borůvka-updt is quite simple with modest performance gain. We propose a more sophisticated meta algorithm, *Stages*, that further improves cache performance.

Stages first partitions the edges in E into groups, $E_1, E_2, \cdots, E_g$, with $|E_i| > n/2$ $(1 \leq i \leq g-1)$ except possibly for E_g. Then *Borůvka* is applied to the subgraph induced by E_1. All resulting connected components are contracted to super-vertices, and the endpoints of each edge in E_2 are updated. Again *Borůvka* is applied to the subgraph induced by E_2. *Stages* continues until all edge groups are processed. When Stages terminates, an MSF for graph G is computed.

Let $w_{min}(E_i)$ and $w_{max}(E_i)$ be the minimum weight and maximum weight of edges in E_i, respectively. Algorithm 4 gives the formal description of *Stages*.

Algorithm 4. Stages(E, D)

```
1: $F \leftarrow \emptyset$
2: for $1 \leq i \leq n$ do
3:     $D[i] \leftarrow i$
4: end for
5: Partition $E$ into $g$ groups $E_1, E_2, \cdots,$
   $E_g$ with $w_{min}(E_i) \geq w_{max}(E_{i-1})$, $2 \leq$
   $i \leq g$
6: for $1 \leq i \leq g$ do
7:     $F \leftarrow F \cup$ Borůvka $(E_i, D)$
8:     if $i < g$ then
9:         for $(u, v) \in E_{i+1}$ in parallel do
10:            $(u, v) \leftarrow (D[u], D[v])$
11:        end for
12:    end if
13: end for
14: return $F$
```

We prove that *Stages* indeed computes a minimum spanning forest of G. We first show that F is a spanning forest.

Lemma 1. *The edges found by* Stages *form a spanning forest.*

Proof. Algorithm 4 repeatedly invokes *Borůvka* on the groups of edges. For each group, shortcut is done on D so that all vertices in the same connected component so far will have the same D value. When *Stages* terminates, for each vertex $u \in V$, $D[u]$ represents the final connected component u belongs to. So F is a spanning graph of G. In a *Borůvka* iteration, a vertex (or super-vertex) is grafted at most once by an edge, thus F is a forest.

Theorem 1. Stages *computes a minimum spanning forest.*

Proof. We assume without loss of generality that no two edges in G have the same weight. Denote the set of edges found by *Borůvka* and *Stages* F_B and F_S, respectively. For any edge $e \in F_B$, we show $e \in F_S$. In the beginning, the D values for the two endpoints of e are different. Suppose e is processed in group E_j, $1 \leq j \leq g$. After $E_1, \cdots, E_{j-1}$ are processed, the D values for the two endpoints can not be the same. Otherwise there exists a path in F_S connecting the two endpoints, and the weights of the edges on the path are all smaller than $w(e)$. Thus $e \notin F_B$ by the cycle property, a contradiction. As *Stages* invokes *Borůvka* with E_j, it computes a minimum spanning forest of a graph with e as one of its edges. The D values of the two endpoints of e must converge. The convergence must be caused by e, otherwise by the *Borůvka* algorithm, again there exists a path in E_j connecting the two endpoints of e with the weights of the edges less than $w(e)$, thus another contradiction. So $F_B \subseteq F_S$. By lemma 1, $|F_B| = |F_S|$, so $F_B = F_S$.

For large inputs the conflicts among processors competing for the same locks are rare. With p processors *Borůvka* takes $O\left(\frac{m+n}{p}\log^2 n\right)$ time. Let the number of edges in E_i be $q \cdot n$, $1/2 < q \leq m/n$, *Stages* takes $O\left(\frac{m}{pqn}(qn+n)\log^2 n + \frac{m}{p}\right)$ time. *Borůvka* and *Stages* have the same asymptotic complexity when $qn = \Theta(m)$. *Stages* degenerates into *Borůvka* when $qn = m$. In general *Stages* has more operations than *Borůvka*.

Let us consider the impact of processing the edges in groups on locality. After E_1 is processed and the MSF is computed for the induced graph, some connected components are formed and then contracted into super-vertices. Updating the endpoints of edges in E_2 with their super-vertices increases the probability that either the two endpoints of an edge are within one component or multiple edges are incident to the same components. Thus we expect E_2 be processed much faster than E_1. The components subsume even more vertices after E_2 is processed, and are again contracted into super-vertices. As *Stages* progresses, more and more accesses to D, Min and I become regular (cache hits). The way the graph contracts is dependent on the input topology and weight distribution. Assuming all edges incident to a vertex (or super-vertex) have the same probability of being in the MSF, according to a theorem (see Theorem 2) of evolution random graph theory, a fairly large number of vertices will contract to a single super-vertex. As a result, *Stages* is expected to have better locality than *Borůvka* for many graphs.

Theorem 2. *Under the Erdös-Rényi model there is a unique giant component of order $f(c)n$ in the graph when $m \sim cn$ with $c > 1/2$. Function $f(c) = 1 - \frac{1}{2c}\sum_{k=1}^{\infty}\frac{k^{k-1}}{k!}(2ce^{-2c})^k$ approaches 1 as c increases [19].*

A routine similar to sample sort is used in our implementation to distribute E into g buckets. Note that a full sort is not necessary for our purpose. Due to limited space we do not present the details of the partitioning algorithm.

Figures 3 and 4 show the performance improvement of *Stages* over *Borůvka* on a random graph with 100M vertices, 400M edges and a scalefree graph with

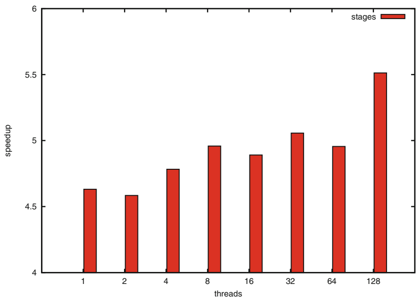

Fig. 3. Random graph

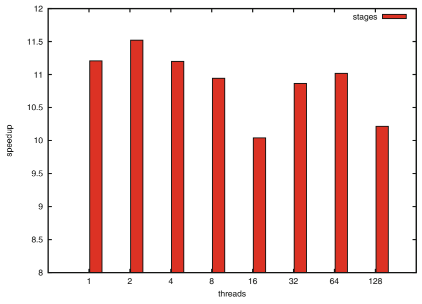

Fig. 4. Scalefree graph

100M vertices, 1B edges. The speedups achieved are between 4.5 to 5.5 for the random graph and between 10.2 to 11.5 for the scalefree graph.

5 PRAM Simulation

Locks in *Borůvka* (Algorithm 2) not only incur conflicts among processors but also exacerbate poor cache performance for large inputs as accesses to them are also random.

To reduce synchronization overhead, we adopt a PRAM simulation technique for simulating CRCW PRAM algorithms on EREW PRAM [13,23]. We cast *Borůvka* to the priority CRCW model where a priority function (*min*, in our case) resolves the conflict of concurrent writes. That is, when current writes to the same location occur, the one with the smallest value wins, and all others abort. The algorithm is then simulated on EREW PRAM. When implemented on multicore machines, all grafting actions on a vertex are grouped together and executed by one single processor.

ER implements concurrent reads, shown in Algorithm 5. Algorithm 5 implements indirect parallel accesses of D through R, that is, $C[i] \leftarrow D[R[i]], 1 \leq i \leq \bar{m}, |R| = \bar{m}, |D| = \bar{n}$. Lines 1–8 partition R and D into blocks (one per each processor), and group the access requests in R according to the target processor that owns the D block being accessed. At lines 9–13, each processor serves access requests to its block so that at any time there is only one processor accessing any element of D. At lines 14–19 each processor sends its retrieved D values to the requesting processors, and at lines 20–24 the D values are matched to the requests. In the algorithm, $\oplus$ is a concatenation operator.

Algorithm 5. ER (C, D, R, $\bar{n}$, $\bar{m}$, p)

```
1: divide $R$ and $D$ into $p$ blocks of size $s = \bar{m}/p$ and $w = \bar{n}/p$, respectively
2: for $1 \leq k \leq p$ in parallel do
3:     sort $R_k$ and store original location of $j^{th}$ element in $P_k[j]$, $1 \leq j \leq s$
4:     partition $R_k$ into $p$ blocks $R_k^j$, $1 \leq j \leq p$, such that $\forall r \in R_k^j, \frac{r}{s} = j$
5: end for
6: for $1 \leq j \leq p$ in parallel do
7:     $R'_j \leftarrow \oplus_{k=1}^{p} R_k^j$
8: end for
9: for $1 \leq k \leq p$ in parallel do
10:     for $1 \leq j \leq |R'_k|$ do
11:         $S_k[j] \leftarrow D_k[R'_k[j]]$
12:     end for
13: end for
14: for $1 \leq k \leq p$ in parallel do
15:     partition $S_k$ into $p$ consecutive blocks $S_k^j$, $1 \leq j \leq p$, such that $|S_k^j| = |R_k^j|$
16: end for
17: for $1 \leq k \leq p$ in parallel do
18:     $S'_k \leftarrow \oplus_{j=1}^{p} S_j^k$, $1 \leq k \leq p$
19: end for
20: for $1 \leq k \leq p$ in parallel do
21:     for $1 \leq j \leq s$ do
22:         $C_k[P_k[j]] \leftarrow S'_k[j]$
23:     end for
24: end for
25: $C \leftarrow \oplus_{k=1}^{p} C_k$
```

Algorithm 6. EW(W, D, R, $\bar{n}$, $\bar{m}$, p)

```
1: divide $R$, $W$, and $D$ into $p$ blocks of size $s = \bar{m}/p$, $s = \bar{m}/p$, and $w = \bar{n}/p$, respectively
2: for $1 \le k \le p$ in parallel do
3:     sort $R_k$ and $W_k$ and store original location of $j^{th}$ element in $Pr_k[j]$ and $Pw_k[j]$, $1 \le j \le s$, respectively
4:     partition $R_k$ and $W_k$ into $p$ blocks $R_k^j$, and $W_k^j$, $1 \le j \le p$, respectively, such that $\forall r \in R_k^j, \frac{r}{s} = j$, $\forall r \in W_k^j, \frac{r}{s} = j$
5: end for
6: for $1 \le j \le p$ in parallel do
7:     $R'_j \leftarrow \oplus_{k=1}^{p} R_k^j$
8:     $W'_j \leftarrow \oplus_{k=1}^{p} W_k^j$
9: end for
10: for $1 \le k \le p$ in parallel do
11:     for $1 \le j \le |R'_k|$ do
12:         $D_k[R'_k[j]] \leftarrow min(D_k[R'_k[j]], W'_k[j])$
13:     end for
14:     add edges for winning writes to $F$
15: end for
16: return F
```

The concurrent writes are done collectively through EW with the min priority function, shown in Algorithm 6. Similar to Algorithm 5, lines 1–9 partition the write requests (in R) and values (in D) into blocks, and group requests according to the target processor that owns the D block. At lines 10–15 a processor writes the data to the D location applying the min function. There are no concurrent writes to D at any time. With ER and EW, the *Borůvka* iteration is transformed into Algorithm 7. Note fine-grain synchronization is no longer needed.

Algorithm 7. *Simu* (E, D)

```
1: $F \leftarrow \emptyset$
2: for $1 \le i \le n$ in parallel do
3:     $I[i] \leftarrow \emptyset$
4: end for
5: for $1 \le i \le m$ in parallel do
6:     let $(u, v) = e_i \in E$
7:     $A[2*i-1] \leftarrow u$, $A[2*i] \leftarrow v$
8: end for
9: call $ER(C, A, D, n, 2*m, p)$
10: for $1 \le i \le m$ in parallel do
11:     $d_u \leftarrow C[2*i-1], d_v \leftarrow, C[2*i]$
12:     if $Min[D[d_u]] > w(e_i)$ then
13:         $R[i] \leftarrow d_v$, $W[i] \leftarrow w(e_i)$
14:     end if
15: end for
16: $F \leftarrow F \cup EW(W, D, R, min, n, m, p)$
17: for $1 \le i \le n$ in parallel do
18:     while $D[i] \ne D[D[i]]$ do
19:         $D[i] \leftarrow D[D[i]]$
20:     end while
21: end for
22: return F
```

Algorithm 7 (*Simu*) also has better locality than Algorithm 1 as random accesses to D are transformed into multiple random accesses to blocks of D. When these blocks fit in cache, performance can be improved.

The performance improvement for a random graph of 100M vertices, 400M edges and a scalefree graph of 100M vertices, 1B edges is shown in Fig. 5. The speedups are between 2.5 to 11 for the random graph and between 3 and 9 for the scalefree graph (Fig. 6).

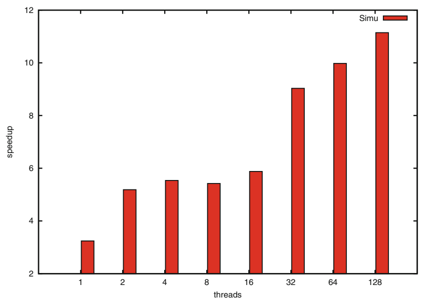

Fig. 5. Random graph

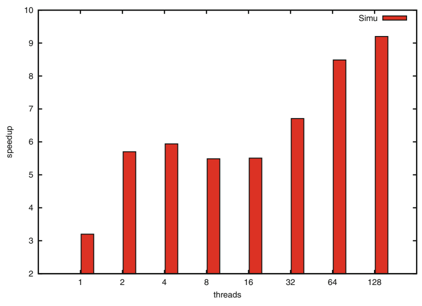

Fig. 6. Scalefree graph

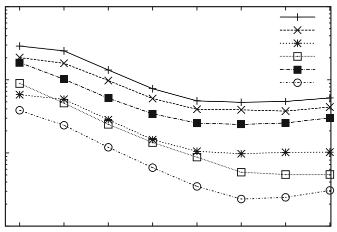

Fig. 7. In log − log plot

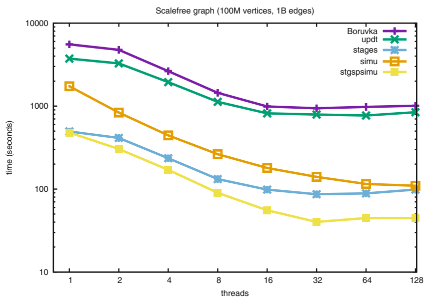

Fig. 8. In log − log plot

6 Combining Stages and PRAM Simulation

Both *Stages* and *Simu* are meta approaches that improve the performance of existing MSF algorithms. *Stages* exploits the properties of both the input and the *Borůvka* iteration to improve locality, while *Simu* reduces synchronization and improves cache performance through scheduling the memory accesses. *Stages* is specific to the "graft-and-shortcut" pattern, while *Simu* can be applied to many irregular algorithms. We combine the two approaches. That is, we use *Simu* as the base algorithm for *Stages*. In Algorithm 4, instead of calling Algorithm 2 at line 7, we call Algorithm 7. We call this approach *Stages+Simu*.

Figures 7 and 8 show the performance of *Borůvka*, *Borůvka-updt*, *Stages* with *Borůvka*, *Simu*, and *Stages+Simu* for a random graph with 100M vertices, 400M edges and a scalefree graph with 100M vertices, 1B edges. For both inputs, *Borůvka-updt* is faster than *Borůvka*. *Stages* and *Simu* are faster than *Borůvka-updt*. For the random graph, *Simu* is faster than *Stages*, while for the scalefree graph, *Stages* is faster than *Simu*. *Stages+Simu* is consistently the fastest among all implementations. *Stages+Simu* is more than an order of magnitude faster than the base implementation.

Table 1. networks

Network	Vertices	Edges
Bone	7798786	202895861
Adhead	12582914	327484556
Along	144441346	867447553
Journal	4846609	85702474
Phone	73037362	1248697024
Twitter	41652230	1468365181
Random	100M	400M
Scalefree	100M	1B

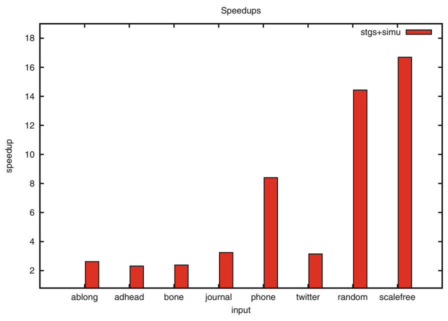

Fig. 9. Speedups

We next compare the performance of *Stages+Simu* with the best prior parallel MSF implementations on several networks. In addition to the synthetic graphs, we include two classes of real-world networks shown in Table 1. The first class contains three computer vision networks (*bone*, *adhead*, and *ablong*) constructed from the images from Siemmens Corporation Research and Robarts Research Institute [8]. A vertex is placed on a 2D or 3D grid corresponding to the pixels (or voxels). Edges connect the vertex to other vertices within the standard 8- (or 26-) neighborhood. These networks have regular structures and small weights. The second class of networks are social networks. These networks capture social relationships among entities. *journal* is a snapshot of the friendship network of the LiveJournal on-line blogging community [22]. *phone* records the phone calls whose origination or termination involve users in Cambridge, MA. *twitter* is a snap shot of the twitter networks [16]. The social networks are assigned random weights.

Figure 9 shows the speedups of *Stages+Simu* over the best prior parallel implementation (the fastest among Bor-AL, Bor-FAL, and *Borůvka*) at 32 threads. The range of speedups is between 2 to 17. The speedups are relatively modest for the vision networks (on average 2.43). This is largely due to the small weights and the regularity in the network. The speedups are larger for social networks. For *phone* the speedup is 8.4. The speedup for *twitter* is 3.1. Although *twitter* has more edges than *phone*, it has much fewer vertices. Recall that poor locality in *Borůvka* is associated with accessing D and Min with the vertices as indices. Similar networks with more vertices will likely see more performance improvement from *Stages+Simu*. Both *random* and *scalefree* have more vertices (100M), and for them the speedups are 14.4 and 16.7, respectively.

7 Conclusion and Future Work

We present accelerating minimum spanning forest computations through a series of meta algorithms. We improve locality and reduce synchronization for existing MSF implementations. The three approaches range from simple to sophisticated with different degrees of performance gain. *Stages+Simu* combines two different locality optimization approaches and can drastically improve the performance of MSF algorithms. *Stages+Simu* is up to 17 times faster than the base *Borůvka* implementation for synthetic graphs, and it is between 2 to 9 times faster for vision networks and social networks. As networks in applications become larger, locality optimization such as ours becomes even more critical to achieving high performance on current and future platforms.

In future work we will study optimization of graph algorithms on GPUs and a cluster of GPUs. We will evaluate the effectiveness of approaches presented in our study. We will also study architectural support for efficient execution of graph algorithms on current and emerging architectures.

References

1. Adler, M., Dittrich, W., Juurlink, B., Kutyłowski, M., Rieping, I.: Communication-optimal parallel minimum spanning tree algorithms (extended abstract). In: SPAA 1998: Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 27–36. ACM, New York (1998)
2. Agarwal, V., Petrini, F., Pasetto, D., Bader, D.: Scalable graph exploration on multicore processors. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010, pp. 1–11. IEEE Computer Society, Washington, DC (2010)
3. An, L., Xiang, Q., Chavez, S.: A fast implementation of the minimum spanning tree method for phase unwrapping. IEEE Trans. Med. Imaging **19**(8), 805–808 (2000)
4. Bader, D.A., Cong, G.: Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. In: Proceedings of the 18th International Parallel and Distributed Processing Symposium, IPDPS 2004, Santa Fe, New Mexico, April 2004
5. Banerjee, D., Sharma, S., Kothapalli, K.: Work efficient parallel algorithms for large graph exploration. In: 2013 20th International Conference on High Performance Computing (HiPC), pp. 433–442, December 2013
6. Barnat, J., Bauch, P., Brim, L., Ceska, M.: Computing strongly connected components in parallel on cuda. In: 2011 IEEE International Parallel Distributed Processing Symposium (IPDPS), pp. 544–555, May 2011
7. Beamer, S., Asanović, K., Patterson, D.: Direction-optimizing breadth-first search. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 12:1–12:10. IEEE Computer Society Press, Los Alamitos, CA, USA (2012)
8. Boykov, Y., Funka-Lea, G.: Graph cuts and efficient n-d image segmentation. Int. J. Comput. Vision **70**(2), 109–131 (2006). http://dx.doi.org/10.1007/s11263-006-7934-5
9. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: a recursive model for graph mining. In: Proceedings of the 4th SIAM International Conference on Data Mining, April 2004
10. Chen, C., Morris, S.: Visualizing evolving networks: minimum spanning trees versus pathfinder networks. In: IEEE Symposium on Information Visualization, Seattle, WA, October 2003
11. Chong, K.W., Han, Y., Lam, T.W.: Concurrent threads and optimal parallel minimum spanning tree algorithm. J. ACM **48**, 297–323 (2001)

12. Cong, G., Bader, D.A.: Lock-free parallel algorithms: an experimental study. In: Bougé, L., Prasanna, V.K. (eds.) HiPC 2004. LNCS, vol. 3296, pp. 516–527. Springer, Heidelberg (2004)
13. Fich, F., Ragde, P., Wigderson, A.: Simulations among concurrent-write prams. Algorithmica **3**(1–4), 43–51 (1988)
14. Goh, K.I., Oh, E., Jeong, H., Kahng, B., Kim, D.: Classification of scale-free networks. Proc. Natl. Acad. Sci. **99**, 12583 (2002). http://www.citebase.org/cgi-bin/citations?id=oai:arXiv.org:cond-mat/0205232
15. Hong, S., Oguntebi, T., Olukotun, K.: Efficient parallel graph exploration on multi-core CPU and GPU. In: 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 78–88, october 2011
16. Kunegis, J.: KONECT - The Koblenz network collection. In: Proceedings of the International Conference on World Wide Web Companion, pp. 1343–1350 (2013). http://userpages.uni-koblenz.de/kunegis/paper/kunegis-koblenz-network-collection.pdf
17. Meguerdichian, S., Koushanfar, F., Potkonjak, M., Srivastava, M.: Coverage problems in wireless ad-hoc sensor networks. In: Proceedings of the INFOCOM 2001, pp. 1380–1387. IEEE Press, Anchorage, April 2001
18. Olman, V., Xu, D., Xu, Y.: Identification of regulatory binding sites using minimum spanning trees. In: Proceedings of the 8th Pacific Symposium on Biocomputing (PSB 2003), pp. 327–338. World Scientific Pub., Hawaii (2003)
19. Palmer, E.: Graphical Evolution. Wiley-Interscience Series in Discrete Mathematic. Wiley, New York (1985)
20. Patwary, M., Ref, P., Manne, F.: Multi-core spanning forest algorithms using the disjoint-set data structure. In: Proceedings of the 2012 IEEE International Parallel & Distributed Processing Symposium, IPDPS 2012, pp. 827–835. IEEE Computer Society, Washington, DC (2012)
21. Pettie, S., Ramachandran, V.: A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. SIAM J. Comput. **31**(6), 1879–1895 (2002)
22. Stanford SNAP Large Network Dataset Collection. http://memetracker.org/data/index.html
23. Vishkin, U.: Implementation of simultaneous memory address access in models that forbid it. J. Algorithms **4**(1), 45–50 (1983). http://dblp.uni-trier.de/db/journals/jal/jal4.html#Vishkin83