Parallel Minimum Spanning Forest Computation using Sparse Matrix Kernels

Tim Baer
Department of Computer Science
University of Illinois at Urbana-Champaign
tbaer2@illinois.edu

Raghavendra Kanakagiri
Department of Computer Science
University of Illinois at Urbana-Champaign
raghaven@illinois.edu

Edgar Solomonik
Department of Computer Science
University of Illinois at Urbana-Champaign
solomon2@illinois.edu

***Abstract*—Formulations of graph algorithms using sparse linear algebra have yielded highly scalable distributed algorithms for problems such as connectivity and shortest path computation. We develop the first formulation of the Awerbuch-Shiloach parallel minimum spanning forest (MSF) algorithm using linear algebra primitives. We introduce a multilinear kernel that operates on an adjacency matrix and two vectors. This kernel updates graph vertices by simultaneously using information from both adjacent edges and vertices. In addition, we explore optimizations to accelerate the shortcutting step in the Awerbuch-Shiloach algorithm. We implement this MSF algorithm with Cyclops, a distributed-memory library for generalized sparse tensor algebra. We analyze the parallel scalability of our implementation on the Stampede2 supercomputer.**

I. Introduction

Graph computations are ubiquitous in many discplines with applicability to many real world problems. In recent years, there has been considerable interest in formulating graph algorithms via sparse linear algebra primitives. These primitives mask the underlying irregular communication patterns, lack of cache locality, and high synchronization costs of graph algorithms to achieve scalability. In this paper, we focus on the Minimum Spanning Forest (MSF) problem. When the graph is connected, the minimum spanning forest is a minimum spanning tree (MST). MST has many practical applications including network design for computers, transportation, telecommunication, and electrical grids [13], [16]. Approximation algorithms for traveling salesman, maximum flow, weighted perfect matching, invoke the computation of MST as subroutines.

Given an undirected weighted graph, a minimum spanning forest (MSF) is a subset of edges that connects all the vertices in each connected component with the minimum possible total edge weight. The minimum spanning forest is unique if each edge has a distinct weight. Boruvka, Prim, and Kruskal each proposed what has become three classic MSF algorithms in the literature. MSF algorithms generally rely on the cut property of minimum spanning trees: for any cut C of the graph, the minimum-weight cut edge $\operatorname{argmin}_{e \in C} w(e)$ belongs to all minimum spanning trees of that connected component.

Parallel MSF algorithms also have a long history. Among the classical algorithms, Boruvka's exhibits a high degree of parallelism. Essentially all parallel MSF algorithms including Boruvka's defines disjoint subgraphs of the MSF in some way and joins them together with minimum-weight cut edges. Awerbuch and Shiloach (AS) propose a MSF algorithm that has $O(\log n)$ depth on $n+m$ processors, where n is the number of vertices and m is the number of edges in the graph [27]. In this work, we develop a distributed-memory implementation of the AS algorithm based on sparse matrix algebra.

Many recent proposals on graph algorithms targeting scalablility and performance in distributed-memory implementations use linear algebraic primitives [4], [30], [36]. The plurality of algebraic graph algorithm libraries [10], [14], [34] has motivated standardization efforts such as the GraphBLAS [7]. The Combinatorial BLAS (CombBLAS) library [6] has been one of the first to leverage sparse matrix–vector and matrix–matrix primitives to implement graph algorithms. Their implementation of the Awerbuch-Shiloach connectivity algorithm as part of LACC [4] is perhaps the most closely related work to ours. We leverage the Cyclops library for our algebraic MSF implementation, which has previously been used to implement betweenness centrality using sparse matrix multiplication [30]. We overview the basics of algebraic graph algorithms, and describe the additional challenges in developing an algebraic implementation for the Awerbuch-Shiloach MSF algorithm relative to connectivity in Section II. We then contrast our work to other related efforts in Section V.

We propose a formulation of a variant of the AS MSF algorithm in terms of sparse matrix primitives in Section III. To achieve this, we introduce a multilinear kernel that updates graph vertices by simultaneously using information from both adjacent edges and vertices. This kernel is different from the

We are grateful to Zhaoyu Wu and David (Yunxin) Zhang for early contributions to this project. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. Via XSEDE, the authors made use of the TACC Stampede2 supercomputer. The research was supported by the US NSF OAC via award No. 1942995.

matrix-vector-product operations proposed in the existing literature including GraphBLAS. The multilinear kernel permits an algebraic implementation of the AS MSF algorithm that does not incur overhead in updating the adjacency matrix (for the AS connectivity algorithm, matrix–vector products and vector operations suffice). Further, we propose new optimizations for the shortcutting step in the AS algorithm, which transforms MSF trees into stars. The optimized algorithm shortcuts all trees into stars (height-1 trees) using one round of communication. We provide a cost analysis of the algorithm and optimizations in Section IV.

We implement this as part of a general distributed tensor library, Cyclops Tensor Framework (CTF) [31] that supports a variety of generalized vector/matrix/tensor operations and some all-at-once multi-tensor contraction kernels [29]. In Section VII, the algebraic formulation of MSF using the multilinear kernel coupled with optimizations achieves excellent strong and weak scaling for both synthetic and real world graphs. These real world graphs include some of the largest available graphs in the SNAP [20] and SuiteSparse datasets [11].

Overall, our paper makes the following novel contributions:

- we propose a new algebraic primitive that enables an efficient parallel implementation of the AS MSF algorithm [27],
- we provide the first distributed-memory implementation of the AS algorithm (the LACC implementation of the Awerbuch-Shiloach algorithm for connectivity [27] is the closest related work [4]),
- we propose a new optimization to the shortcutting procedure, which performs communication for all pointer-chasing rounds in a single stage,
- we demonstrate scalability of our algebraic implementation of the AS algorithm on up to 256 nodes (16K cores) of Stampede2 on graphs with up to 11 billion edges and 183 million vertices.

II. Background

We consider the case of an undirected weighted graph $G = (V, E, w : E \to \mathbb{R})$ with n vertices, m edges, and distinct edge weights. We label the vertices $\{1, \ldots, n\}$. A tree is an undirected graph in which any two vertices are connected by exactly one path. A directed rooted tree is a tree in which a single vertex is designated as the root and the edges of the tree are oriented toward the root. A star is a directed rooted tree of height at most 1. A forest is a disjoint union of trees. A directed rooted forest is a disjoint union of directed rooted trees. An outgoing edge from a tree T is an edge (i, j) such that $i \in T$ and $j \notin T$. An outgoing edge from a vertex i belonging to a tree T is an outgoing edge from T that is adjacent to i. A minimum outgoing edge from a tree is an outgoing edge with the smallest weight. A minimum outgoing edge from a vertex is an outgoing edge from that vertex with the smallest weight. The adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ of graph G is defined by $a_{ij} = w(i, j)$ if $(i, j) \in E$ and ∞ otherwise.

A. Basic Algebraic Structures

Monoids: A monoid $(\mathbb{S}, \oplus)$ is a set $\mathbb{S}$ equipped with an associative binary operation $\oplus : \mathbb{S} \times \mathbb{S} \to \mathbb{S}$ called addition and an identity element.

Semirings: A semiring is an extension of a monoid to two binary operations. Formally, a semiring $(\mathbb{S}, \oplus, \otimes)$ is a set $\mathbb{S}$ equipped with binary operations $\oplus, \otimes : \mathbb{S} \times \mathbb{S} \to \mathbb{S}$ called addition and multiplication respectively, satisfying

- additive associativity,
- additive commutativity,
- multiplicative associativity,
- left and right distributivity.

These conditions imply the existence of additive and multiplicative identities for semirings. A simple example of a monoid is the set of binary strings equipped with a concatenation operation $(\{0, 1\}^*, \cdot)$. The usual semiring in arithmetic is $(\mathbb{R}, +, *)$. Note that semirings do not guarantee the existence of additive inverses, so fast matrix multiplication algorithms like Strassen's algorithm [32] do not apply in general.

B. Algebraic Graph Algorithms

The formalism of algebraic structures such as monoids and semirings permit simple yet expressive graph algorithms. The key connection to linear algebra is the adjacency matrix representation of a graph. Many graph algorithms can be rewritten in terms of matrix-vector and matrix-matrix multiplications with the adjacency matrix on a certain monoid or semiring. Since fast matrix multiplications have been studied from a wide variety of algorithmic lenses, this paradigm allows such algorithms to leverage optimizations in settings such as asymptotic and parallel/distributed complexity. Specialized sparse matrix-vector (SpMV) and sparse matrix-matrix multiplication (SpMSpM) algorithms have been designed with asymptotic complexity depending on nnz, the number of nonzero entries in the matrix. In particular, a SpMV may be computed with $O(nnz)$ floating point additions and multiplications.

As a simple example of an algebraic graph algorithm, we review a linear algebraic interpretation of the Bellman-Ford algorithm for single source shortest paths (SSSP) based on discussion from [17]. Formally, given a graph $G = (V, E)$ with edge weights $w : E \to \mathbb{R}$ and a starting vertex s, the SSSP problem is to compute the length of a shortest path from s to j for all $j \in V$. The Bellman-Ford algorithm can be derived with dynamic programming on the number of hops on a shortest path from s to j for all $j \in V$. We store tentative shortest path distances in $\boldsymbol{d}^{(\ell)} \in \mathbb{R}^n$, where $d_j^{(\ell)}$ is the shortest path distance from s to j among paths with at most ℓ hops. The Bellman-Ford algorithm is based on the idea of *edge relaxations*: we call an edge (k, j) tense if $d_j^{(\ell)} > d_k^{(\ell)} + w(k, j)$. At each iteration, we *relax* all tense edges, meaning that we update $d_j^{(\ell)} \leftarrow d_k^{(\ell)} + w(k, j)$ if (k, j) is tense. After $n - 1$ iterations, we have converged to the length of a shortest path from s to j for all $j \in V$.

Algorithm: Now, consider an intepretation of the Bellman-Ford algorithm that implements each edge relaxation with

a SpMV of $\boldsymbol{A}$ and $\boldsymbol{d}^{(\ell)}$. Instead of the usual semiring, we perform operations on $\mathbb{R}$ over the tropical semiring $(\mathbb{R}, \oplus, \otimes)$ where $\oplus = \min$ and $\otimes = +$. We initialize $d_j^{(0)}$ to 0 if $j = s$ and ∞ otherwise. We iteratively compute $\boldsymbol{d}^{(\ell+1)} \leftarrow \boldsymbol{d}^{(\ell)}\boldsymbol{A}$ over the tropical semiring. After $n-1$ iterations, $d_j^{(n-1)}$ stores the length of a shortest path from s to j for all $j \in V$. Since we perform $n-1$ SpMVs with the adjacency matrix, the run time is $O(nm)$.

Intuition: Consider updating the tentative distances for a fixed j, $d_j^{(\ell+1)} \leftarrow (\boldsymbol{d}^{(\ell)}\boldsymbol{A})_j = \bigoplus_k d_k^{(\ell)} \otimes a_{kj}$. Replacing the generic semiring addition and multiplication symbols with $\min$ and $+$ respectively yields $d_j^{(\ell+1)} \leftarrow \min_k\{d_k^{(\ell)} + a_{kj}\}$. By introspection, we can interpret this expression as an edge relaxation.

C. Awerbuch-Shiloach Algorithm

Awerbuch and Shiloach [27] (AS) provide a classic parallel algorithm for computing the minimum spanning forest of an undirected graph $G = (V, E, w : E \to \mathbb{R})$ with distinct edge weights. In addition to the graph itself, the algorithm maintains a parent forest of directed rooted trees, each intuitively representing a part of the minimum spanning forest that has already been discovered. The algorithm grows the minimum spanning forest by joining trees together with minimum outgoing edges. The algorithm only computes minimum outgoing edges for trees that are stars. In such cases, we may find these edges with work proportional to the number of member vertices and in $O(1)$ depth.

We represent the minimum spanning forest with a set of edges F. In addition, we represent the parent forest with a parent vector $\boldsymbol{p} \in V^n$, where p_i stores the parent of vertex i. In the first iteration, $F \leftarrow \varnothing$, and the parent forest consists of n isolated vertices, each with a self-loop. At each iteration, the algorithm computes the minimum outgoing edge (i, j) for each star in parallel. We then join vertex i's parent to vertex j's parent and add edge (i, j) to F, while taking care to prevent cycles in the parent forest. Next, we shortcut trees to reduce their height by a factor of nearly two and possibly create many stars for the next iteration. The algorithm terminates when no more trees in the parent forest can be joined. If the input graph is connected, the algorithm will terminate when the parent forest converges to a single connected component and F contains the minimum spanning tree.

Let us assume that we have access to a routine that decides whether a given vertex belongs to a star. In summary, the algorithm performs the following three steps until convergence of the parent vector $\boldsymbol{p}$:

(i) Star hooking: joins two trees. For each star root i,

$$(i, j) \leftarrow \text{ minimum outgoing edge from } i\text{'s star},$$
$$p_i \leftarrow p_j.$$

(ii) Tie breaking: breaks cycles by detecting and removing star hookings that create cycles. For each star root i,

$$p_i \leftarrow \begin{cases} i & : i < p_i \text{ and } i = p_{p_i}, \\ p_i & : \text{otherwise.} \end{cases}$$

(iii) Shortcutting: reduces the height of trees. For each vertex i that does not belong to a star,

$$p_i \leftarrow p_{p_i}.$$

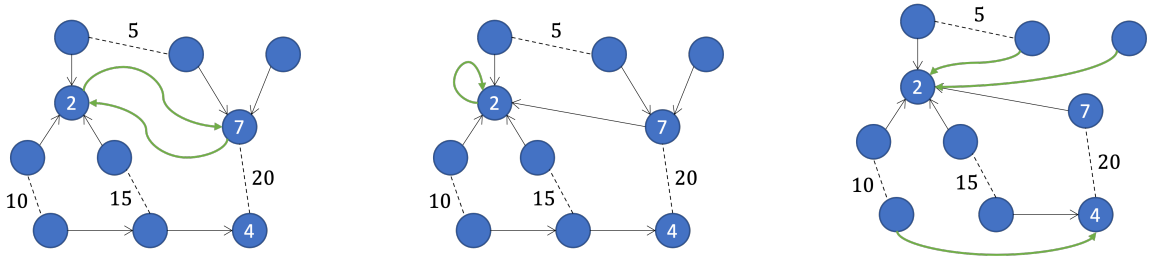
We visualize these steps in Figure 1. AS show that the sum of the heights of all the trees in the forest decreases by a factor of at least $3/2$ each iteration, resulting in convergence after $\log_{3/2} n$ iterations.

In [27], Awerbuch and Shiloach first present a parallel algorithm for computing the connected components (CC) of a graph that uses modified hooking and tie breaking steps, while reusing the shortcutting and starcheck steps from the MSF algorithm. For CC, it is sufficient to hook with any outgoing edge from a star instead of only the minimum such edge for the MSF. Computing the minimum outgoing edge from a star may not be expressed as a relaxation of the edges of a subset of vertices.

Star hooking: We consider the PRAM parallelization of a routine that computes the minimum outgoing edge from each star. In particular, we assume the concurrent read concurrent write (CRCW) PRAM with the conflict resolution that the write with the minimum value succeeds. Consider the edge $(i, j) \in E$ that is assigned to some processor. Suppose that the vertex i belongs to a star. Then, the edge (i, j) is outgoing from i's star if $p_i \neq p_j$. Notice that the condition that vertex i belongs to a star is necessary: if i does not belong to a star, (i, j) might not an outgoing edge but still have $p_i \neq p_j$. The processor that owns edge (i, j) continues only if vertex i belongs to a star. Next, the processor reads p_i and p_j to decide whether $p_i \neq p_j$. If $p_i \neq p_j$, then the processor writes $r_{p_i} \leftarrow (i, j)$. We have computed the minimum outgoing edge r_{p_i} from star root p_i with $r_{p_i} \leftarrow \operatorname{argmin}_{(i,j)\in E_i} w(i, j)$ where $E_i \subseteq E$ is the set of outgoing edges from vertex i. We see that this routine has work $O(m)$ and depth $O(1)$. By Brent's scheduling principle, we can compute this routine with p processors in $O(m/p)$ steps. For weaker PRAM models without concurrent write or minimum conflict resolution [3], we may simulate the update to r_{p_i} with a slowdown of $O(\log p)$.

Tie breaking: Step (ii) detects and breaks cycles of length 2. We show that after star hooking, there can not exist cycles of length greater than 2. Suppose for contradication that there exists a cycle of length $k > 2$. Given that the edge weights of the input graph are unique, some edge in the cycle has the largest weight. Since the input graph is undirected and the edges form a cycle of length k, then this edge is not the minimum outgoing edge from any star. We see that step (ii) is sufficient to remove any cycles introduced during star hooking.

Starcheck: Before each of the three steps, the algorithm requires vertices to determine whether or not they belong to a star. First, each vertex checks $p_i = p_{p_i}$ to decide whether its parent is a root. If not, the vertex does not belong to a star ($s_i =$ False) and the vertex informs its parent's parent that the tree is not a star ($s_{p_{p_i}} =$ False). After this step, it suffices for each vertex that has not yet determined that it is not in a star to query their parent to determine whether they belong to a star (if $s_i =$ True, $s_i = s_{p_i}$).



(a) Star hooking: star roots 2 and 7 hook with their minimum outgoing edges. The tree rooted at 4 is not a star so it does not hook.

(b) Tie breaking: breaks the cycle introduced in star hooking by setting 2 to be its own parent.

(c) Shortcutting: changes each vertex's parent to the parent of its parent.

Figure 1: Awerbuch-Shiloach algorithm steps where dotted black lines are outgoing edges, solid black lines are parent pointers, and solid green lines are the updated parent pointers.

D. Algebraic Connectivity Algorithms

Given a MSF, each tree corresponds to a connected component of the original graph. In [27], Awerbuch and Shiloach also present a parallel algorithm for connectivity based on the Shiloach-Vishkin algorithm [28]. Instead of hooking with only the minimum outgoing edge from a star, their connectivity variant hooks with *any* outgoing edge. In addition, Awerbuch and Shiloach replace the tie breaking step with two different kinds of hooking: conditional and unconditional. The shortcutting and starcheck steps remain the same across their MSF and CC algorithms. In this section, we review recent work [4] [36] on expressing the connectivity algorithm with linear algebra operations.

For conditional hooking, star roots are only allowed to hook onto other trees with a smaller parent index. Note that while this condition avoids cycles, it leads to $O(n)$ depth without the unconditional hooking step. In [4], Azad and Buluc rewrite the conditional hooking step as follows,

$$p_i^h \leftarrow \begin{cases} \text{argmin}_{p_j}\{a_{ij}\} & : i \text{ belongs to a star,} \\ 0 & : \text{ otherwise,} \end{cases}$$

$$p_{p_i} \underset{\min}{\leftarrow} \begin{cases} p_i^h & : p_i^h < p_i, \\ p_i & : \text{otherwise.} \end{cases}$$

First, we scan the neighbors of each vertex that belongs to a star and store the parent of a neighbor with the smallest index in $\boldsymbol{p}^h$. This step is implemented with a SpMV-like expression. Next, we only allow hooking onto other trees with a smaller parent index. The unconditional hooking step follows similarly, except without the check that $p_i^h < p_i$. The shortcutting and star checking steps can be implemented with reads/writes to vectors.

Performing conditional hooking followed by unconditional hooking is not applicable to the MSF algorithm. For example, the minimum outgoing edge (i, j) from a vertex i's star may not have $p_i^h < p_i$ so in that scenario, another edge would be used during conditional hooking. Instead, we require a function that operates on p_i, a_{ij}, p_j all-at-once, which we discuss on Section III-A.

III. Algebraic MSF

A. Multilinear Function to Find Outgoing Edges

We introduce a type of multilinear function that updates vertices by simultaneously using information from an edge and its two adjacent vertices. Consider the problem discussed in Section II-C of computing the weight of the minimum outgoing edge from a vertex i that belongs to a star. Let us denote the parent vector with $\boldsymbol{p} \in V^n$ where p_i is the parent of vertex i. Since we assume that vertex i belongs to a star, we decide whether an edge (i, j) is outgoing from i's star with $p_i \neq p_j$. We first seek to design a function f that outputs a_{ij} when edge (i, j) is outgoing from i's star and ∞ otherwise. Such functions are of the form $f : V \times \mathbb{R} \times V \to \mathbb{R}$, where

$$f(p_i, a_{ij}, p_j) = \begin{cases} a_{ij} & : p_i \neq p_j \text{ and } i \text{ belongs to a star,} \\ \infty & : \text{otherwise.} \end{cases}$$

Notice that if the vertex i did not belong to a star, then $f(p_i, a_{ij}, p_j) = \infty$ for any a_{ij} and p_j. We may compute the weight of the minimum outgoing edge from i's star with

$$q_i \underset{\min_j}{\leftarrow} f(p_i, a_{ij}, p_j).$$

More generally, this multilinear function is of the form $f : \mathbb{S}_{\boldsymbol{x}} \times \mathbb{S}_{\boldsymbol{A}} \times \mathbb{S}_{\boldsymbol{y}} \to \mathbb{S}_{\boldsymbol{w}}$, where

$$w_i \leftarrow \bigoplus_j f(x_i, a_{ij}, y_j).$$

with $\boldsymbol{w} \in \mathbb{S}_{\boldsymbol{w}}^n$, $\boldsymbol{A} \in \mathbb{S}_{\boldsymbol{A}}^{n \times n}$, $\boldsymbol{x} \in \mathbb{S}_{\boldsymbol{x}}^n$, $\boldsymbol{y} \in \mathbb{S}_{\boldsymbol{y}}^n$ and $\mathbb{S}_{\boldsymbol{w}}$ is equipped with a binary operation $\oplus$. In Section IV-A, we analyze the complexity of evaluating multilinear functions of this type in terms of PRAM complexity and communication cost.

B. Algebraic MSF Algorithm

In Algorithm 1, we reformulate the Awerbuch-Shiloach algorithm using linear algebraic primitives. Given an undirected graph $G = (V, E)$ equipped with distinct edge weights $w : E \to \mathbb{R}$, the algorithm computes the weight of the MSF. For clarity of presentation, we omit tracking the MSF itself but note that we may do so storing by (a_{ij}, i, j) in $\boldsymbol{A}$ and carrying this information throughout operations. We represent

the parent forest with a parent vector $\boldsymbol{p} \in V^n$, where p_i stores the parent of i. On line 1, we define a set $\mathbb{EDGE}$ which contains pairs consisting of an edge weight and an entry from the parent vector. The monoid $(\mathbb{EDGE}, \text{MINWEIGHT})$ outputs the pair with the least edge weight.

We leverage the multilinear function described in Section III-A to compute the minimum outgoing edge from each vertex that belongs to a star on line 9. Since we need the parent of the destination of the minimum outgoing edge to hook, we modify f to return the pair $(a_{ij}, p_j) \in \mathbb{EDGE}$. On line 10, we project the minimum outgoing edges of the children onto their star root and keep the smallest such edge. We can write this projection more verbosely as

$$r_i \leftarrow \text{MINWEIGHT}_j\{q_j : p_j = i\},$$

where $r \in \mathbb{EDGE}^n$. Intuitively, r_i stores the minimum outgoing edge from the star with root i. Next, stars hook on line 11. Note that only non-zero values of $\boldsymbol{r}$ are read, so $\boldsymbol{p}$ is unchanged for vertices that are not star roots. Since star hooking may create cycles in the parent forest, we detect which hooks must be removed to avoid cycles on line 12. We use $\boldsymbol{t}$ to fix the parent forest and update sum appropriately on lines 13 and 14. We finally shortcut on line 15.

IV. Parallel Analysis

A. Multilinear Kernel

We propose an all-at-once kernel to compute multilinear functions of the form introduced in Section III-A. We demonstrate that this multilinear kernel decreases the number of writes to remote data when compared to pairwise formulations. We implement a multilinear kernel as a part of the CTF library and optimize the vector distribution compared to the default implementation.

All-at-once: We count the number of writes to remote data and the communication cost of this all-at-once kernel. First, we map $\boldsymbol{A}$ to a 2D $\sqrt{p} \times \sqrt{p}$ processor grid and $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w}$ to a 1D p row-like processor grid. We denote processes on the 1D grid with (s) and on the 2D grid by (r, s). We use superscripts to denote the subset of an input assigned to each process. For example, $\boldsymbol{A}^{(r,s)}$ is the subset of $\boldsymbol{A}$ mapped to process (r, s). First, we redistribute $\boldsymbol{x}$ collect $\boldsymbol{x}^{(r)}$ along rows of the processor grid. Next, we broadcast $\boldsymbol{x}^{(r)}$ over all processes (r, t) and $\boldsymbol{y}^{(s)}$ over all processes (t, s), where t is variable. Each process (r, s) now owns $\boldsymbol{A}^{(r,s)}, \boldsymbol{x}^{(r)}, \boldsymbol{y}^{(s)}$ and locally computes $\boldsymbol{w}^{(r,s)} = f(\boldsymbol{x}^{(r)}, \boldsymbol{A}^{(r,s)}, \boldsymbol{y}^{(s)})$ locally. We then reduce over columns s to yield $\boldsymbol{w}^{(r)} = \bigoplus_s \boldsymbol{w}^{(r,s)}$. We see that this kernel requires $\frac{n}{\sqrt{p}}$ writes to main memory. The interprocess communication cost comprises of redistribution, broadcast, and reduction of a vector of local size $n/\sqrt{p}$. The amount of vector communication may in principle be reduced if the input vectors are sparse or if the ouput is sparse (or an output mask is supplied). We visualize the data distribution and communication pattern in Figure 2.

Pairwise: Alternatively, this multilinear kernel can be implemented with two SpMV-like operations. However, we show that this approach requires nnz more writes to remote data

Algorithm 1 Minimum Spanning Forest

Require: $G = (V, E, w : E \to \mathbb{R})$ where $V = \{1, \ldots, n\}$, is an undirected graph with distinct edge weights.

1: We use the monoid $(\mathbb{EDGE}, \text{MINWEIGHT})$, where $\mathbb{EDGE} = \underbrace{\{w(e) : e \in E\} \cup \{\infty\}}_{\text{weight}} \times \underbrace{\{1, \ldots, n\} \cup \{0\}}_{\text{parent}}$ and given edges $x_1, \ldots, x_k \in \mathbb{EDGE}$, $\text{MINWEIGHT}\{x_1, \ldots, x_k\}$ returns the edge x_i with the least weight or $(\infty, 0)$ if $k = 0$.

2: Let $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ be the adjacency matrix of G, where $a_{ij} = \begin{cases} w(i,j) & : (i,j) \in E \\ \infty & : \text{otherwise} \end{cases}$

3: Let $\boldsymbol{p}^{\text{old}} = \begin{bmatrix} 0 & \ldots & 0 \end{bmatrix}^T$ be the old parent vector.

4: Let $\boldsymbol{p} = \begin{bmatrix} 1 & \ldots & n \end{bmatrix}^T$ be the parent vector.

5: Let $f : V \times \mathbb{R} \times V \to \mathbb{EDGE}$, where $f(p_i, a_{ij}, p_j) = \begin{cases} (a_{ij}, p_j) & : p_i \neq p_j \\ (\infty, 0) & : \text{otherwise} \end{cases}$

6: $sum \leftarrow 0$

7: **while** $\boldsymbol{p} \neq \boldsymbol{p}^{\text{old}}$ **do**

8: $\quad \boldsymbol{p}^{\text{old}} = \boldsymbol{p}$

9: $\quad q_i \xleftarrow[\text{MINWEIGHT}_j]{} \begin{cases} f(p_i, a_{ij}, p_j) & : i \text{ belongs to a star} \\ (\infty, 0) & : \text{otherwise} \end{cases}$

10: $\quad r_{p_i} \xleftarrow[\text{MINWEIGHT}_i]{} q_i$

11: $\quad p_i \leftarrow r_i^{parent}$

12: $\quad t_i \leftarrow \begin{cases} \text{True} & : i \text{ is a star root and } i < p_i \text{ and } i = p_{p_i} \\ \text{False} & : \text{otherwise} \end{cases}$

13: $\quad p_i \leftarrow \begin{cases} i & : t_i = \text{True} \\ p_i & : \text{otherwise} \end{cases}$

14: $\quad sum \leftarrow sum + \begin{cases} r_i^{\text{weight}} & : t_i = \text{False} \\ 0 & : \text{otherwise} \end{cases}$

15: $\quad p_i = \begin{cases} p_{p_i} & : i \text{ does not belong to a star} \\ 0 & : \text{otherwise} \end{cases}$

16: **end while**

17: return sum

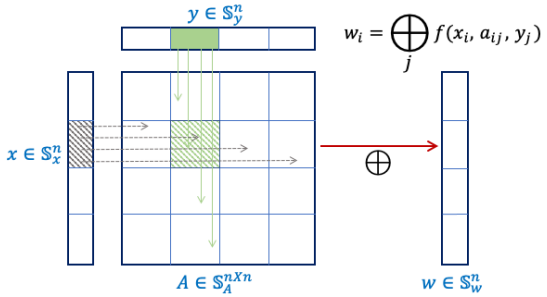


Figure 2: Multilinear kernel communication pattern.

than the all-at-once approach. As an illustrative example, let us consider again the motivating problem for defining this type of multilinear function: given a vertex i that belongs to a star, compute the weight of the minimum outgoing edge from i. Suppose we first compute a pairwise function $g(a_{ij}, p_j)$ which acts on $\boldsymbol{A}$ and $\boldsymbol{p}$. Since we have not yet determined which edges are outgoing from vertex i's star with $p_i \neq p_j$, we update $\boldsymbol{A}$ to contain the pair (a_{ij}, p_j). Now, we can redefine the function f from Section III-A as a pairwise function $f(p_i, (a_{ij}, p_j))$ which acts on $\boldsymbol{p}$ and $\boldsymbol{A}$. This approach has a comparable communication cost as the all-at-once approach, but requires nnz more writes to main memory.

B. Shortcutting

For real-world graphs, we observe that it often takes only a few shortcut iterations to turn all trees into stars. We introduce complete shortcutting: at the end of each iteration, we shortcut repeatedly until each tree in the forest is a star. Since complete shortcutting allows each tree to attempt to hook at every iteration, at least half of the trees will hook successfully after the tie breaking step. We see that complete shortcutting results in an algorithm that requires $\log_2 n$ iterations. However, complete shortcutting can perform at most $\log_{3/2} n$ shortcuts each iteration. While complete shortcutting requires fewer iterations than the original algorithm, the overall depth is worse by a $\log n$ factor. However, we observe in practice that the overhead of complete shortcutting is low and the tradeoff to converge faster is worthwhile. In addition, complete shortcutting simplifies the AS algorithm by removing the need to query whether a vertex belongs to a star. To reduce the communication cost of complete shortcutting, we introduce an optimization called Complete Shortcutting with Prefetching (CSP).

The AS algorithm proposes the following shortcut step: for each vertex i that does not belong to a star, update the parent vector $\boldsymbol{p}$ with

$$p_i \leftarrow p_{p_i}.$$

Assuming that $\boldsymbol{p}$ is distributed on a 1D processor grid, we consider a baseline implementation that reads p_{p_i} for each vertex i assigned to this process that does not belong to a star and then performs the update $p_i \leftarrow p_{p_i}$ locally. We collect duplicate remote reads from a single process to avoid reading p_{p_i} multiple times for vertices with the same parent.

If the number of vertices whose parent changed after hooking is small, we may collect them on all processes and do only local reads for complete shortcutting. Let us denote the parent vector before the hooking step with $\boldsymbol{p}^{prev}$. First, we compare our local data of $\boldsymbol{p}$ and $\boldsymbol{p}^{prev}$ to determine which vertices have a new parent after the hooking step. Notice that a vertex i has a new parent after the hooking step if and only if i is a star root that successfully hooked onto another star. We collect all such vertices and their new parents on all processes. We then implement complete shortcutting with only local reads. At any step of shortcutting, if the parent of a vertex was not a star root that hooked onto to another vertex, shortcutting is unnecessary. We provide a detailed description of SCP in Algorithm 2.

Algorithm 2 Complete Shortcutting with Prefetching

Require: $\boldsymbol{p} \in V^n$ is the parent vector,
$\boldsymbol{p}^{prev} \in V^n$ is the previous parent vector.
1: $changed \leftarrow \varnothing$ {$changed$ is a map}
2: **for** (i, p_i) in local data **do**
3: **if** $p_i \neq p_i^{prev}$ **then**
4: $changed \leftarrow \{(i, p_i)\} \cup changed$
5: **end if**
6: **end for**
7: $changed \leftarrow \mathrm{Allgather}(changed)$
8: **for** (i, p_i) in local data **do**
9: **while** p_i in $changed$ **do** {local read}
10: $p_i \leftarrow changed[p_i]$
11: **end while**
12: **end for**
13: return $\boldsymbol{p}$

V. Related Work

Randomized algorithms have been proposed to compute MST. The randomized linear time algorithm to compute MSTs by Klein et al. [18] inspired linear-work PRAM algorithms, first in CRCW (concurrent read concurrent write) then in EREW (exclusive read exclusive write) PRAM models. Cole et al. proposed the first linear-work parallel algorithm [9]. Pettie and Ramachandran proposed a logarithmic time and linear work algorithm for the EREW PRAM model [25].

Many shared and distributed memory graph frameworks provide an implementation for MST. Galois [23] provides a shared memory implementation of MST using constructs defined in its programming model. To process large graphs on a single machine (with enough memory), Dhulipala et al. develop various scalable graph algorithms [12] including MST. Their approach in many cases is shown to outperform the distributed memory implementations. GraphChi [19] is another single machine implementation that can process massive graphs from secondary storage.

STAPL [8] provides support for both shared and distributed memory parallelism in C++. MST can be implemented in STAPL using the framework's distributed data structures and parallel algorithms. Pregel [22] uses a Bulk Synchronous Parallel (BSP) model, and takes a vertex centric approach for graph computations. The computation is a sequence of iterations where at each step, the framework invokes a user-defined function for each vertex in parallel. The API provided by the framework can be used to program many graph algorithms in a distributed environment. Distribution, underlying message handlers, fault tolerance is invisible to the user. GPS [26] extends the Pregel API to incorporate dynamic repartitioning, and other optimizations. High degree vertices are partitioned across machines to achieve better performance.

One of the shortcomings of Pregel's vertex centric approach is the message load imbalance caused by few vertices that

communicate more messages than others. To address the shortcomings, Pregel+ [35] proposes two techniques, vertex-mirroring and request-respond paradigm. High degree vertices are mirrored on several machines to reduce the number of messages exchanged. The request-respond paradigm allows a vertex to query another vertex for data. All requests from a machine to the same target are merged into one request. Pregel+ is shown to outperform other distributed memory frameworks including GPS [26] and Powergraph [15]. An experimental evaluation to compare various graph frameworks (across different algorithm categories, graph characteristics, etc.) notes that there is no single system that has superior performance in all cases [21]. Panja et al. propose MND-MST to compute MST on heterogeneous systems that house both CPU and GPU compute capabilities [24]. We discuss Pregel+ and MND-MST in Section VII-D.

GraphBLAS [7] provides standardized linear-algebraic primitives for graph computations. LAGraph [33] builds of top of GraphBLAS to provide developers of graph algorithms a set of data structures and utility functions. They describe algebraic implementation of several graph algorithms using LAGraph's API.

We further compare and contrast our MST formulation with the previously proposed linear algebraic frameworks for graph connectivity, LACC [4] and FastSV [36]. Both perform similar hooking and shortcutting steps. The formulations identify sets of "inactive" vertices that do not contribute to the output of certain computations. For example, the vertices that belong to a star do not need to be shortcut. The "active" vertices are naturally represented with a sparse vector. Vertices that belong to converged components, or trees in which no new vertex is added in subsequent iterations, are "inactive" in the hooking, shortcutting, and $\mathrm{Starcheck}$ steps of both LACC and our MSF algorithm. LACC uses the following lemma to identify converged components: except in the first iteration, all remaining stars after hooking are converged components.

If the parent vector p has converged, we still need a last iteration to verify that. FastSV proposes a stronger termination condition: repeat until convergence of the grandparent vector p_{p_i}. Since both CC and MSF terminate when a spanning forest is found, this condition holds for MSF as well. For most real-world graphs, the last iteration does not perform any hooking and only shortcuts trees into stars. In these cases, the stronger termination condition identifies a spanning forest an iteration before all trees are shortcut into stars. We incorporate this optimization in our implementation.

FastSV proposes three hooking optimizations: hooking onto a grandparent, stochastic hooking, and aggressive hooking. Hooking onto a grandparent results in shorter trees after the hooking step. However, this optimization is not applicable with complete shortcutting. The second optimization, stochastic hooking, relaxes the hooking condition to allow hookings to happen more often. Intuitively, stochastic hooking allows a tree to be split into multiple parts, each of which hook independently. However, this highlights the main difference between CC and MSF variants of the AS algorithm. For CC, we can hook using *any* outgoing edge from a tree. For MSF, we must hook using the *minimum* outgoing edge from a star. Stochastic hooking with the MSF algorithm would violate this condition. The last hooking optimization, aggressive hooking, would also violate the requirement that the *minimum* outgoing edge is used to hook.

For routines like the tensor-times-tensor-product (TTTP) that arise in tensor completion, Singh et al. [29] propose a multi-tensor all-at-once contraction that replaces the standard approach of pairwise contraction of tensors. The multilinear kernel used in our MST formulation follows a similar approach.

VI. Experimental Setup

We evaluate the performance of our implementation on the Stampede2 supercomputer. Each node has an Intel Xeon Phi 7250 CPU ("Knights Landing") with 68 cores, 96GB of DDR4 RAM, and 16GB of high-speed on-chip MCDRAM memory (which operates as 16GB direct-mapped L3). In our experiments, we use 16 MPI processes each with 4 OMP threads per node. We use Cyclops Tensor Framework (CTF) for our implementation. We mainly rely on the data distribution and algebraic primitives supported by CTF. CTF supports a variety of generalized vector/matrix/tensor operations. We optimize vector distribution/transpose operations, and use them to implement the multilinear kernel. The implementation is generic, and can be used in the implementation of other algorithms that use multilinear primitives.

ID	Graph	Nodes	Edges
friendster	Friendster	65.6M	1.8B
orkut	Orkut social network	3.1M	117.2M
lj	LiveJournal social network	4M	34.7M
road-usa	Full USA road-network	23.9M	28.9M
road-central	Central USA road-network	14.1M	16.9M
agatha_2015	Deep-learning graph	183.9M	11.6B
moliere_2016	Hypothesis generation network	30.2M	6.7B

Table I: **Real-world undirected graphs**

VII. Evaluation

We present performance results for algebraic implementation of MST using the following classes of input graphs:

- road-network graphs from the DIMACS graph partitioning and clustering challenge [2],
- real-world social-network graphs from the SNAP dataset [20],
- two of the largest graphs in terms of number of edges from SuiteSparse matrix collection [11],
- synthetic uniform random and R-MAT graphs.

The various road and social network graphs, and graphs from SuiteSparse used in the evaluation are listed in Table I. We use both the R-MAT and real-word graphs to show strong

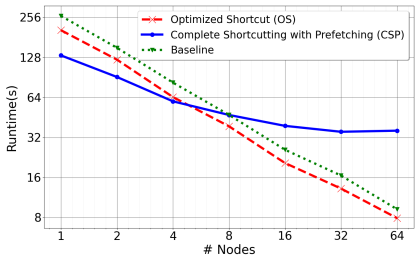


Figure 3: Performance of shortcut optimizations (Road USA graph).

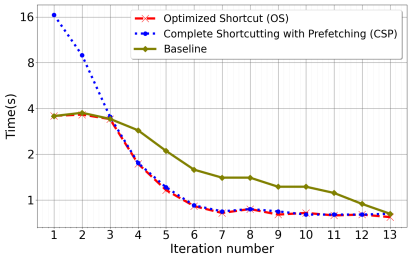


Figure 4: Performance of changed parent shortcut optimization (SCP) on 16 nodes (Road USA graph).

scaling results. We use uniform random graphs to present weak scaling results. For the unweighted graphs, we generate uniformly distributed integers from 1 through 255 as edge weights. This choice is consistent with previous performance studies of graph algorithms (GAP benchmark suite [5] and Graph 500's SSSP proposal [1]).

A. *Shortcut optimization*

In Figure 3, we compare the performance of the CSP optimization proposed in Section IV-B with no optimization in shortcutting. A single invocation of complete shortcutting performs several iterations to convert every tree in the forest to a star. We refer to these shortcutting iterations as sub-iterations.

In CSP, after the hooking step, all vertices that hooked onto a new parent are gathered on all processes. The gathered data comprises pairs of vertex IDs and their respective new parent IDs. This allows CSP to perform complete shortcutting without querying for remote data in every sub-iteration.

Many vertices in a process might have the same parent, and hence a process might send duplicate queries to the parent vector. In the baseline we do not optimize the shortcut function except for collating duplicate reads from a single process. The Optimized Shortcut (OS) is a combination of CSP and baseline. In OS, we invoke CSP only if the number of vertices to be gathered is below a fixed threshold. Otherwise, the unoptimized shortcut is invoked. We use an empirically determined threshold of 1310k, which translates to 20MB gathered on every process. We present data for runs where the respective optimizations are invoked in every iteration. We also present data for the baseline.

For up to 4 nodes, invoking CSP in all iterations performs the best. The predominant step in complete shortcutting is the parent vector read performed every sub-iteration. CSP on the other hand gathers data only once during complete shortcutting. The parent vector is distributed across nodes. With the increase in node count, since each process holds fewer vector elements, the read needs to fetch fewer elements per process. We see that it is beneficial to have no optimization in shortcut for the first few sub-iterations. If CSP has to gather substantial amount of data, the benefit of skipping the reads in every sub iteration is lost. CSP performs better at lower node counts or when the amount of data gathered and processed is not significant. This can also be inferred from the optimized shortcut performance where we invoke CSP based on a fixed threshold of 1310k. The algorithm computes MST for road_usa in 13 iterations. As the algorithm progresses, fewer vertices hook on to new parents. CSP gets invoked from the fourth iteration.

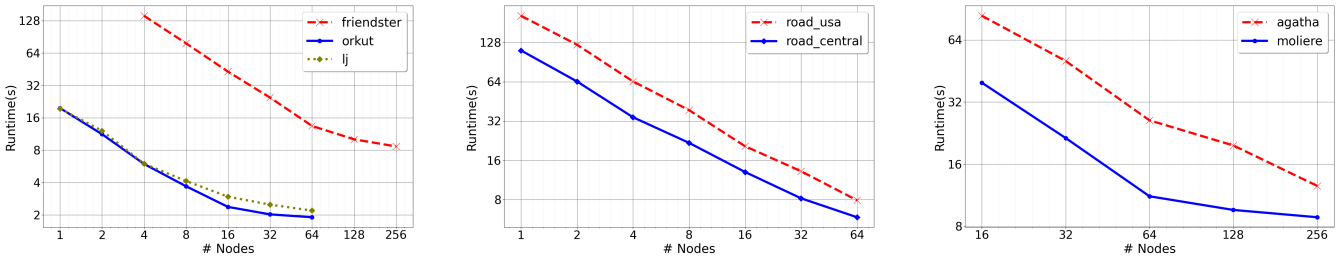
Figure 4 shows the time per iteration for complete shortcutting on 16 nodes. CSP outperforms no optimization in shortcut from the fourth iteration and is relatively expensive in the first two iterations. The optimized shortcut invokes CSP only after three iterations of the algorithm when fewer vertices hook on to a new parent. We keep the threshold fixed across node counts and graphs. For certain graphs we observe that the switch to CSP's shortcut is not always triggered at an optimal point, i.e., an earlier switch to CSP could have fetched better overall performance. A dynamic threshold would further improve performance.

B. *Strong Scaling*

We study the performance of the implementation to lower time to solution with an increase in node count. We consider three social-network graphs in Figure 5(a). Orkut and Livejournal show similar scaling results. Orkut is relatively denser but has a smaller diameter when compared to Friendster and Livejournal. The time per iteration is higher in Orkut, but the number of iterations for the algorithm to converge is higher (by one) for LiveJournal. On a single node, the multilinear kernel for Orkut takes 1.41s whereas for LiveJournal it is 0.65s, per iteration. Friendster is the largest graph available in the SNAP dataset.

The smallest number of nodes that can execute Friendster successfully is 4. Given the higher number of vertices and edges, both the computation of the multilinear kernel and the communication in the shortcut step per node are expensive. We note that the time per iteration for the multilinear kernel achieves good scalability for all the three graphs. For a fixed node count, the time per iteration for the multilinear kernel remains fairly constant across iterations. But, the overall time per iteration reduces with the progress in the algorithm steps as fewer vector elements need to be updated in the shortcut function.

From the DIMACS dataset we pick the road-network graphs Figure 5(b). These graphs have a larger diameter and are sparser. On a single node, though the computation time for



(a) Social network graphs (b) DIMACS road network graphs (c) SuiteSparse graphs

Figure 5: Strong scaling of real-world graphs.

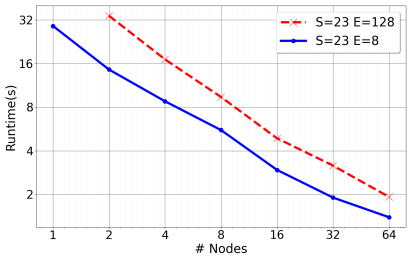


Figure 6: Strong scaling of R-MAT graph

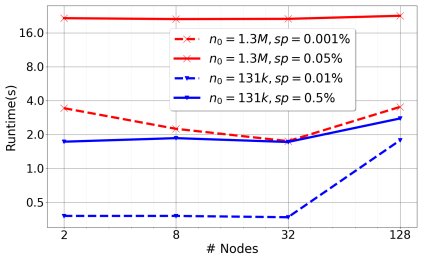


Figure 7: Edge weak scaling of uniform random graphs. Constant $n^2/p = n_0^2$ and edge percentage $f = 100 * m/n^2$.

the multilinear kernel is comparable with that of the Orkut graph, the number of iterations required to compute the MST is roughly doubled. The graphs also have 5X to 8X more vertices than Orkut, which translates to higher per iteration time because of the updates to the parent vector in the shortcut function. We achieve good strong scaling till 64 nodes with around 26X speedup when using 64X more nodes.

In Figure 5(c), we present strong scaling results for the two largest graphs from the SuiteSparse matrix collection. These graphs are dense, and the machine learning graph Agatha has around 700X more edges than the smallest graph considered until now (road-central). The minimum number of nodes required to compute is 16. We achieve good scaling till 256 nodes (the maximum number of nodes we tried).

R-MAT graphs: We use two R-MAT graphs with $\log_2(n) \approx S = 23$, and the average degree controlled by $k \approx E = 8, 128$. In figure 6 we show the strong scaling results for the two graphs. Both the graphs require only four iterations to compute MST. The computation time for shortcut is fairly comparable for both the graphs. Since the graph with $E = 8$ is sparse when compared to the graph with $E = 128$, the multilinear kernel is only a fraction of the iteration time. When $E = 128$ the multilinear kernel computation dominates the overall iteration time. Both graphs show good strong scalability until 64 nodes.

C. Weak Scaling

We use uniform random graphs, in which all nodes have the same expected degree to study weak scaling. We consider "edge weak scaling" where n^2/p is kept constant. We achieve good weak scaling as shown in Figure 7. The number of iterations is smaller when the sparsity is 0.05% or 0.5% when compared to sparsity of 0.001% or 0.01%, but the multilinear computation time per iteration is the most significant factor for the dense graphs.

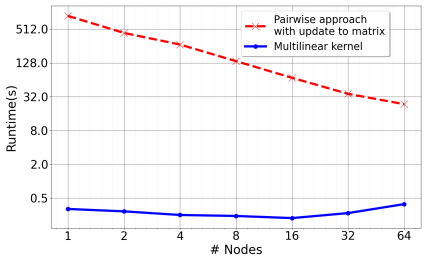


Figure 8: A case for multilinear kernel. R-MAT graph (S=17 E=8).

In Figure 8, we show the performance of an R-MAT graph with and without using the multilinear kernel. The multilinear kernel implementation outperforms the plain Awerbuch-Shiloach algebraic implementation by several orders of magnitude.

D. Comparison with Pregel+ and MND-MST

Yan et al. in Pregel+ [35] introduce techniques to optimize the implementation of the MSF algorithm proposed in Pregel [22]. They compare their implementation with several other parallel graph computing frameworks, and show that they are

competitive in terms of runtimes on various real world graphs. They do not discuss scalability results.

Panja et al. propose a multi-node multi-device algorithm for MST (MND-MST) [24]. They partition the input graph across multiple nodes and devices and compute local MSTs in parallel using Boruvka's algorithm. They employ 1D partitioning scheme to balance the number of edges across computing units. They present results for both CPU-only and multi-device (CPU-GPU) systems. Their results are shown to outperform Pregel+.

Both Pregel+ and MND-MST use 16 node cluster, and present results for the *road_usa* graph. We note that the computing platforms are different in each of the experimental setups. Pregel+ reports 19.95s to compute MST for *road_usa*. Panja et al. report a total runtime of 190s and 29.6s to compute MST using Pregel+ and MND-MST, respectively, on their platform. For *road_usa* graph, they report slowdowns beyond 16 nodes. They note that the communication overhead is higher at larger node counts. We show strong scalability for *road_usa* with an execution time of 20.6s and 7.9s on 16 and 64 nodes, respectively. While the architectures and backend systems are different (Pregel+ uses Hadoop, which is not a standard module in Stampede2), these results show that our algebraic MST implementation is roughly comparable in performance to hand-optimized MST codes.

VIII. Conclusions

Multi-tensor contraction kernels have shown to be advantageous in tensor completion operations such as TTTP and MTTKRP. We show that a similar all-at-once approach in the algebraic formulation for MST is efficient when compared to a pairwise approach which involves an expensive update. We suspect that such multilinear kernels or their simple variants can be leveraged to optimize existing algebraic algorithms and design new ones.

We observe that the number of shortcuts required for complete shortcutting in practice is much lower than upper bound discussed that leads to depth $O(\log^2 n)$. Complete shortcutting simplifies the Awerbuch-Shiloach algorithm by removing the need to query whether a vertex belongs to a star. We show that our algebraic formulation coupled with complete shortcutting and prefetch optimization achieves excellent strong and weak scaling for various graphs including some of the largest available real-world graphs.

References

[1] "Graph 500." [Online]. Available: https://graph500.org

[2] "DIMACS implementation challenge - graph partitioning and graph clustering," 2011. [Online]. Available: https://www.cc.gatech.edu/dimacs10/archive/clustering.shtml

[3] S. G. Akl and L. Chen, "On the power of some PRAM models," *Parallel Algorithms and Applications*, vol. 13, no. 4, pp. 307–319, 1999. [Online]. Available: https://doi.org/10.1080/01495739908947372

[4] A. Azad and A. Buluc, "LACC: A linear-algebraic algorithm for finding connected components in distributed memory," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 2–12.

[5] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[6] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[7] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "Design of the GraphBLAS API for C," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017, pp. 643–652.

[8] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger, "STAPL: Standard template adaptive parallel library," in *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, ser. SYSTOR '10. New York, NY, USA: Association for Computing Machinery, 2010. [Online]. Available: https://doi.org/10.1145/1815695.1815713

[9] R. Cole, P. N. Klein, and R. E. Tarjan, "A linear-work parallel algorithm for finding minimum spanning trees," 1994.

[10] T. A. Davis, "Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra," *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 4, pp. 1–25, 2019.

[11] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[12] L. Dhulipala, G. E. Blelloch, and J. Shun, "Theoretically efficient parallel graph algorithms can be fast and scalable," in *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[13] C. F. Bazlamaçcı and K. S. Hindi, "Minimum-weight spanning tree algorithms a survey and empirical study," *Computers and Operations Research*, vol. 28, no. 8, pp. 767–785, 2001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0305054800000071

[14] V. Gadepally, J. Bolewski, D. Hook, D. Hutchison, B. Miller, and J. Kepner, "Graphulo: Linear algebra graph kernels for nosql databases," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 2015, pp. 822–830.

[15] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. USA: USENIX Association, 2012, p. 17–30.

[16] R. Graham and P. Hell, "On the history of the minimum spanning tree problem," *Annals of the History of Computing*, vol. 7, pp. 43–57, 1985.

[17] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*. USA: Society for Industrial and Applied Mathematics, 2011.

[18] P. N. Klein and R. E. Tarjan, "A randomized linear-time algorithm for finding minimum spanning trees," in *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, ser. STOC '94. New York, NY, USA: Association for Computing Machinery, 1994, p. 9–15. [Online]. Available: https://doi.org/10.1145/195058.195084

[19] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-Scale graph computation on just a PC," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. USA: USENIX Association, 2012, p. 31–46.

[20] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[21] Y. Lu, J. Cheng, D. Yan, and H. Wu, "Large-scale distributed graph computing systems: An experimental evaluation," *Proc. VLDB Endow.*, vol. 8, no. 3, p. 281–292, Nov. 2014. [Online]. Available: https://doi.org/10.14778/2735508.2735517

[22] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 135–146. [Online]. Available: https://doi.org/10.1145/1807167.1807184

[23] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 456–471. [Online]. Available: https://doi.org/10.1145/2517349.2522739

[24] R. Panja and S. Vadhiyar, "MND-MST: A multi-node multi-device parallel boruvka's MST algorithm," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3225058.3225146

[25] S. Pettie and V. Ramachandran, "A randomized time-work optimal parallel algorithm for finding a minimum spanning forest," in *SIAM J. COMPUT*. Springer, 1999, pp. 233–244.

[26] S. Salihoglu and J. Widom, "GPS: A graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, ser. SSDBM. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2484838.2484843

[27] Y. Shiloach and B. Awerbuch, "New connectivity and MSF algorithms for shuffle-exchange network and PRAM," *IEEE Transactions on Computers*, vol. 36, no. 10, pp. 1258–1263, oct 1987.

[28] Y. Shiloach and U. Vishkin, "An $\mathcal{O}(\log n)$ parallel connectivity algorithm," *Journal of Algorithms*, vol. 3, pp. 57 – 67, 1982.

[29] N. Singh, Z. Zhang, X. Wu, N. Zhang, S. Zhang, and E. Solomonik, "Distributed-memory tensor completion for generalized loss functions in Python using new sparse tensor kernels," 2021.

[30] E. Solomonik, M. Besta, F. Vella, and T. Hoefler, "Scaling betweenness centrality using communication-efficient sparse matrix multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3126908.3126971

[31] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, and J. Demmel, "A massively parallel tensor contraction framework for coupled-cluster computations," *J. Parallel Distrib. Comput.*, vol. 74, no. 12, p. 3176–3190, Dec. 2014. [Online]. Available: https://doi.org/10.1016/j.jpdc.2014.06.002

[32] V. Strassen, "Gaussian elimination is not optimal," *Numerische mathematik*, vol. 13, no. 4, pp. 354–356, 1969.

[33] G. Szárnyas, D. A. Bader, T. A. Davis, J. Kitchen, T. G. Mattson, S. McMillan, and E. Welch, "LAGraph: Linear algebra, network analysis libraries, and the study of graph algorithms," *CoRR*, vol. abs/2104.01661, 2021. [Online]. Available: https://arxiv.org/abs/2104.01661

[34] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam, "Fast linear algebra-based triangle counting with KokkosKernels," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–7.

[35] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Effective techniques for message reduction and load balancing in distributed graph computation," in *Proceedings of the 24th International Conference on World Wide Web*, ser. WWW '15. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2015, p. 1307–1317. [Online]. Available: https://doi.org/10.1145/2736277.2741096

[36] Y. Zhang, A. Azad, and Z. Hu, "FastSV: A distributed-memory connected component algorithm with fast convergence," in *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing (PP20)*. SIAM, 2020, pp. 46–57.