Scaling Betweenness Centrality using Communication-Efficient Sparse Matrix Multiplication

Edgar Solomonik[1], Maciej Besta[2], Flavio Vella[3], Torsten Hoefler[2]
[1]Department of Computer Science, University of Illinois at Urbana-Champaign
[2]Department of Computer Science, ETH Zurich
[3]Sapienza University of Rome
solomon2@illinois.edu, maciej.besta@inf.ethz.ch, vella@di.uniroma1.it, htor@inf.ethz.ch

ABSTRACT

Betweenness centrality (BC) is a crucial graph problem that measures the significance of a vertex by the number of shortest paths leading through it. We propose Maximal Frontier Betweenness Centrality (MFBC): a succinct BC algorithm based on novel sparse matrix multiplication routines that performs a factor of $p^{1/3}$ less communication on p processors than the best known alternatives, for graphs with n vertices and average degree $k = n/p^{2/3}$. We formulate, implement, and prove the correctness of MFBC for weighted graphs by leveraging monoids instead of semirings, which enables a surprisingly succinct formulation. MFBC scales well for both extremely sparse and relatively dense graphs. It automatically searches a space of distributed data decompositions and sparse matrix multiplication algorithms for the most advantageous configuration. The MFBC implementation outperforms the well-known CombBLAS library by up to 8x and shows more robust performance. Our design methodology is readily extensible to other graph problems.

1 INTRODUCTION

Graph processing underlies many computational problems in machine learning, computational science, and other disciplines [32]. Yet, many parallel graph algorithms struggle to achieve scalability due to irregular communication patterns, high synchronization costs, and lack of spatial or temporal cache locality. To alleviate this, we pursue the methodology of formulating scalable graph algorithms via sparse linear algebra primitives.

In this paper, we focus on betweenness centrality (BC), an important graph problem that measures the significance of a vertex v based on the number of shortest paths leading through v. BC is used in analyzing various networks in biology, transportation, and terrorism prevention [4]. The Brandes BC algorithm [10, 36] provides a work-efficient way to obtain centrality scores without needing to store all shortest-paths simultaneously, achieving a quintessential reduction in the memory footprint. To date, most parallelizations of BC have leveraged the breadth first search (BFS) primitive [5, 18, 23, 33, 35, 42, 43], which can be used to implement Brandes algorithm on unweighted graphs [26].

Contrarily to these schemes, we propose to use the Bellman-Ford shortest path algorithm [7, 21] to compute shortest distances and multiplicities. This enables us to achieve maximal parallelism by simultaneously propagating centrality scores from all vertices that have determined their final score, starting with the leaves of the shortest path tree. We refer to this approach as the Maximal Frontier Betweenness Centrality (MFBC) algorithm, because the frontier of vertices whose edges are relaxed includes *all* vertices that could yield progress. We prove the correctness of the new scheme for computing shortest distances, multiplicities, and centrality scores with a succinct path-based argument applied to factors of partial centrality scores, simplifying the Brandes approach [10].

Each set of frontier relaxations in MFBC is done via multiplication of a pair of sparse matrices. These multiplications are defined to perform the desired relaxation via the use of monoids, monoid actions, and auxiliary functions. This algebraic formalism enables concise definition as well as implementation of MFBC via the Cyclops Tensor Framework (CTF) [41]. CTF is a distributed-memory library that supports tensor contraction and summation; operations that generalize the sparse matrix multiplications MFBC requires. By implementing a robust set of sparse matrix multiplication algorithms that are provably communication-efficient, our framework allows rapid implementation of bulk synchronous graph algorithms with regular communication patterns and low synchronization cost.

Aside from the need for transposition (data-reordering), sparse tensor contractions are equivalent to sparse matrix multiplication. We present a communication cost analysis of the sparse matrix multiplications we developed within CTF, and use it to derive the total MFBC cost. The theoretical scalability both of the sparse matrix multiplication, as well as of the MFBC algorithm surpasses the state-of-the-art. In our evaluation, the new algorithm obtains excellent strong and weak scaling for both synthetic and real-world power-law graphs. We compare our implementation to that of the Combinatorial BLAS (CombBLAS) library [11], a state-of-the-art matrix-based distributed-memory betweenness centrality code. We demonstrate that the CTF-MFBC code outperforms CombBLAS by up to 8x and measure a reduction in the runtime communication and synchronization costs. While CombBLAS is still faster in some cases, our implementation is general to weighted graphs and attains more consistent performance across different graphs.

SC17, Denver, CO, USA

978-1-4503-5114-0/17/11…$15.00
DOI: 10.1145/3126908.3126971

Graph structure	G	a given graph, $G = (V, E)$; V and E are sets of vertices and edges; if G is weighted, then $G = (V, E, w)$ where $w : E \to \mathbb{W}$.
	n, m	numbers of vertices and edges in G; $n = \|V\|$, $m = \|E\|$.
	$\rho(v)$	degree of v; $\overline{\rho}$ and $\widehat{\rho}$ are the average and maximum degree in G.
	d	diameter of a given graph G.
	A	adjacency matrix of G.
General BC	$\lambda(v)$	betweenness centrality of v.
	$\tau(s, t)$	shortest path distance between s, t.
	$\bar{\sigma}(s, t)$	number of shortest paths between s, t.
	$\sigma(s, t, v)$	number of shortest paths between s, t leading via v.
	$\delta(s, v)$	dependency of s on v; $\delta(s, v) = \sum_{t \in V} \sigma(s, t, v)/\bar{\sigma}(s, t)$.
	$\pi(s, v)$	set of immediate predecessors of v in shortest paths from s to v.

Table 1: Symbols used in the paper; v, s, $t \in V$ are vertices.

2 NOTATION, BACKGROUND, CONCEPTS

We first introduce the notation and basic concepts. The most important employed symbols are summarized in Table 1.

2.1 Basic Graph Notation

We start by presenting the used graph notation. We represent an undirected unweighted labeled graph G as a tuple (V, E); $V = 1 : n$ ($V = \{1, \ldots, n\}$) is a set of vertices and $E \subseteq V \times V$ is a set of edges; $|V| = n$ and $|E| = m$. We denote the set of possible weight values as $\mathbb{W} \subset \mathbb{R} \cup \{\infty\}$. If G is weighted, we have $G = (V, E, w)$ where $w : E \to \mathbb{W}$ is a weight function. We denote the adjacency matrix of G as A; $A(i, j) = w(i, j)$ if $(i, j) \in E$, otherwise $A(i, j) = \infty$.

2.2 Basic Algebraic Structures

We use heavily two structures: semirings and monoids.

Monoids A monoid $(S, \oplus)$ is a set S closed under an associative binary operation $\oplus$ with an identity element. A commutative monoid $(S, \oplus)$ is a monoid where $\oplus$ is commutative, and

$$\bigoplus_{i=j}^{k} s(i) = s(j) \oplus s(j+1) \oplus \cdots \oplus s(k)$$

for any $k \geq j$ with $s(i) \in S$ for each $i \in j : k$. We denote the elementwise application of a monoid operator to a pair of matrices as $A \oplus B$ for any $A, B \in S^{m \times n}$.

Semirings A semiring is defined as a tuple $S = (T, \oplus, \otimes)$; T is a set equipped with two binary operations $\oplus$, $\otimes$ such that $(T, \oplus)$ is a commutative monoid and $(T, \otimes)$ is a monoid. Semirings are often used to develop graph algorithms based on linear algebra primitives. In this work, we use monoids to enable a succinct formulation.

2.3 Algebraic Graph Algorithms

Most graph algorithms can be expressed via matrix-vector or matrix-matrix products. As an introductory example, we consider BFS [16]. BFS starts at a root vertex r and traverses all nodes connected to r by one edge, then the set of nodes two edges away from r, etc. BFS can be used to compute shortest paths in an unweighted graph, which we can represent by an adjacency matrix with elements $A_{ij} \in \{1, \infty\}$. In this case, BFS would visit each vertex v and derive its distance $\tau(r, v)$ from the root vertex r.

Algebraically, BFS can be expressed as iterative multiplication of the sparse adjacency matrix A with a sparse vector x_i over the tropical semiring, (i denotes the iteration number). The tropical semiring is a commutative monoid $(\mathbb{W}, \min)$ combined with the addition operator (replacing the monoid $(\mathbb{R}, +)$ and multiplication operator that are usually used for the matrix-vector product). The BFS algorithm would initialize $x_0^r = 0$ (the initial distance to r is 0) and any other element of x_0 is ∞ (i.e., the initial distance to any other element is ∞). Each BFS iteration computes $x_{i+1} = x_i \bullet_{\langle \min, + \rangle} A$, then screens x_{i+1} retaining only elements that were ∞ in all x_j for $j < i$. The sparsity of the vector is given by all entries which are not equal to ∞, the additive identity of the tropical semiring.

2.4 Brandes' Algorithm

BC derives the importance of a vertex v using the number of shortest paths that pass through v. Let $\bar{\sigma}(s, t)$ be the number of shortest paths between vertices s, t, and let $\sigma(s, t, v)$ be the number of such paths that pass via v, $\sigma(s, s, v) = \sigma(s, t, s) = \sigma(s, t, t) = 0$. The centrality score of v is defined as $\lambda(v) = \sum_{s,t \in V} \frac{\sigma(s,t,v)}{\bar{\sigma}(s,t)}$. Define the dependency of a source vertex s on v as: $\delta(s, v) = \sum_{t \in V} \frac{\sigma(s,t,v)}{\bar{\sigma}(s,t)}$. Then, $\lambda(v) = \sum_{s \in V} \delta(s, v)$ where $\delta(s, v)$ satisfies the recurrence: $\delta(s, v) = \sum_{w \in \pi(s,w)} \frac{\bar{\sigma}(s,v)}{\bar{\sigma}(s,w)}(1 + \delta(s, w))$; $\pi(s, w)$ is a list of immediate predecessors of w in the shortest paths from s to w. Brandes' scheme [10] uses this recurrence to compute $\lambda(v)$ in two parts. First, multiple concurrent BFS traversals compute $\pi(s, v)$ and $\bar{\sigma}(s, v)$, $\forall s, v \in V$, obtaining a predecessor tree over G. Second, the tree is traversed backwards (from the highest to the lowest distance) to compute $\delta(s, v)$ and $\lambda(v)$ based on the equations above.

Brandes' algorithm has been a subject of many previous efficiency studies [5, 15, 23, 24, 28, 29, 33, 36–38, 42, 43, 46, 47]. Some efforts considered distributed memory parallelization [8, 19, 20]. The only distributed memory BC implementation done using matrix primitives we are aware of exists in CombBLAS [11]. To the best of our knowledge, we provide the first communication cost analysis of a BC algorithm, and the first implementation leveraging 3D sparse matrix multiplication. A mix of graph replication and blocking have been previously used for BC computation [8], but the communication complexity of the scheme was not analyzed. Furthermore, previous parallel codes and algebraic BC formulations have largely been limited to unweighted graphs.

3 MONOIDS FOR SUCCINCT FORMULATION

Our first idea is to employ commutative monoids for describing MFBC. Semirings permit multiplicative operators only on elements within the same set, while our algorithms require operators on members of different sets. We use monoids to define generalized matrix-vector and matrix-matrix multiplication operators.

To define a suitable matrix multiplication primitive for our algorithms, we permit different domains for the matrices and replace elementwise multiplication with an arbitrary function that is a suitable map between the domains. Specifically, consider two input matrices $A \in D_A^{m \times k}$ and $B \in D_B^{k \times n}$, a bivariate function $f : D_A \times D_B \to D_C$, and a commutative monoid $(D_C, \oplus)$. Then, we denote matrix multiplication (MM) as $C = A \bullet_{\langle \oplus, f \rangle} B$, where each element of the output matrix, $C \in D_C^{m \times n}$, is $C(i, j) = \bigoplus_{k=1}^{n} f(A(i, k), B(k, j))$, $\forall i \in 1 : m, j \in 1 : n$. This MM notation enables a unified description of the main steps of MFBC.

4 MAXIMAL FRONTIER ALGORITHM

We now describe our algebraic maximal frontier BC algorithm (MFBC), which uses the introduced algebraic formulation based on monoids for a succinct description. MFBC consists of two parts. First, it enhances Bellman-Ford to compute distances between vertices ($\tau(s,t)$) and the multiplicities of shortest paths ($\bar{\sigma}(s,t)$); we refer to this part as Maximal Frontier Bellman Ford (MFBF; see Section 4.1). Second, it computes partial centrality factors ($\delta(s,v)$) with a strategy that extends Brandes' algorithm; we thus refer to it as Maximal Frontier Brandes (MFBr; see Section 4.2).

Both MFBF and MFBr use generalized matrix products of: (1) the adjacency matrix A, and (2) a sparse rectangular $n \times n_b$ matrix with elements of one of the two specially defined monoids: *multpaths* (elements from the *multpath monoid* associated with MFBF) and *centpaths* (elements from the *centpath monoid* associated with MFBr). Here, n_b is the *batch size*: the number of vertices for which we solve the final BC score $\lambda(v)$. It constitutes a tradeoff between the time and the storage complexity: MFBC takes n/n_b iterations but must maintain an $n \times n_b$ matrix. Finally, these two combined schemes give MFBC (i.e., informally MFBC = MFBF + MFBr).

4.1 MFBF: Computing Shortest Paths

The core idea behind MFBF is to use and extend Bellman-Ford so that it computes not only shortest paths but also their multiplicities. To achieve this, rather than working only with weights, we define MFBF in terms of multpaths: tuples (that belong to the multpath monoid) which carry both path weight and multiplicity.

4.1.1 The Multpath Monoid.

Intuition: To express Bellman-Ford with multiplicities algebraically, MFBF uses *the multpath monoid* $(\mathbb{M}, \oplus)$. The elements of $\mathbb{M}$ are *multpaths*: tuples that model a weighted path with a multiplicity. The $\oplus$ operator acts on any two multpaths x and y and returns the one with lower weight; if the weights of x and y are equal, then their multiplicities are summed.

Formalism: A multpath $x = (x.w, x.m) \in \mathbb{M} = \mathbb{W} \times \mathbb{N}$ is a path in G with a weight $x.w$ and a multiplicity $x.m$. Then, we have

$$\forall_{x,y \in \mathbb{M}}, \quad x \oplus y = \begin{cases} x & : x.w < y.w \\ y & : x.w > y.w \\ (x.w,\ x.m + y.m) & : x.w = y.w. \end{cases}$$

4.1.2 The Bellman-Ford Action.

Intuition: In each MFBF iteration, we multiply A with a sparse tensor ($\mathcal{T}$) that constitutes the *multpath frontier*: it contains the multpaths of the nodes whose multiplicity changed in the previous iteration. Here, each element-wise operation acts on a multpath (an element from $\mathcal{T}$) and an edge weight (an element from A); we refer to this operation as the *Bellman-Ford Action*.

Formalism: Our MFBF algorithm (Algorithm 1) iteratively updates a matrix T of multpaths via forward traversals from each source vertex. This is done in the inner loop (lines 3-7) where the $\mathcal{T}$ tensor with partial multiplicity scores is updated in each iteration using Bellman-Ford Action $\bullet_{\langle \oplus, f \rangle}$, where the function f is defined as

$$f : \mathbb{M} \times \mathbb{W} \to \mathbb{M}, \qquad f(a, w) = (a.w + w,\ a.m).$$

f is interpreted as an action of the monoid $(\mathbb{W}, +)$ on the set $\mathbb{M}$. This concept generalizes to $n \times k$ matrices, where we have a monoid action $\bullet_{\langle \oplus, f \rangle}$ with monoid $(\mathbb{W}^{n \times n}, \bullet_{\langle \min, + \rangle})$ on the set $\mathbb{M}^{n \times k}$.

4.1.3 Algorithm and Correctness.

Intuition: We obtain shortest path distances and multiplicities via MFBF (Algorithm 1): a Bellman-Ford variant that relaxes all edges adjacent to vertices whose path information changed in the previous iteration. The edge relaxation is done via matrix multiplication of the adjacency matrix and a multpath matrix, which appends edges to the existing frontier of vertices via function f, then uses the multpath operator $\oplus$ to select the minimum distance new set of paths, along with the number of such new paths. This partial multiplicity score is subsequently accumulated to T if it corresponds to a minimum distance path from the starting vertex. Note that the multiplicity is set to 1 (line 1) even if the corresponding weight equals ∞. Thus, such edges are considered in the main loop (line 3). When a path to v with a finite distance is found, it replaces such a multiplicity.

Algorithm 1 $[T] = \text{MFBF}(A, \vec{s})$

Input A: $n \times n$ adjacency matrix, $\vec{s}$: list of starting n_b vertices
Output T: multpath matrix of distances and multiplicities from vertices $\vec{s}$

∇ Existential qualifiers $\forall s \in 1 : n_b$ (denoting starting vertices) and $\forall v \in 1 : n$ (denoting destination vertices) are implicit. ∇

1: $T(s, v) := (A(\vec{s}(s), v), 1)$ ◃ Initialize multpaths.
2: $\mathcal{T} := T$ ◃ Initialize multpath frontier
3: **while** $\mathcal{T}(s, v) \neq (\infty, 0)$ for some s, v **do**
4: $\quad \mathcal{T} := \mathcal{T} \bullet_{\langle \oplus, f \rangle} A$ ◃ Explore nodes adjacent to frontier
5: $\quad T := T \oplus \mathcal{T}$ ◃ Accumulate multiplicities
$\quad \nabla$ Determine new frontier based on updates to the vertex path information ∇
6: $\quad$ **if** $\mathcal{T}(s, v).m = 0 \vee \mathcal{T}(s, v).w > T(s, v).w$ **then** $\mathcal{T}(s, v) := (\infty, 0)$
7: **end while**

Formalism: We can prove that Algorithm 1 will output the correct shortest distances and multiplicites.

Lemma 4.1. *For any adjacency matrix A and vertex set $\vec{s}$, $T = MFBF(A, \vec{s})$ satisfies $T(s, v) = (\tau(s, v), \bar{\sigma}(s, v))$.*

Proof. Let the maximum number of edges in any shortest path from node s be d. For $j \in 1 : d$, $v \in V \setminus \{s\}$, let each $h_j(s, v) \in \mathbb{M}$ be a multpath corresponding to the weight and multiplicity of all shortest paths from vertex s to vertex v containing *up to* j edges (if there are no such paths, $h_j(s, v) = (\infty, 0)$. Further, let each $\hat{h}_j(s, v) \in \mathbb{M}$ be a multpath corresponding to the weight and multiplicity of all shortest paths from vertex s to vertex v containing *exactly* j edges (if there are no such paths, $\hat{h}_j(s, v) = (\infty, 0)$). Note that $h_d(s, v)$ contains the weight and multiplicity of all shortest paths from vertex s to vertex v, since no shortest path can contain more than d edges, therefore $h_d(s, v) = (\tau(s, v), \bar{\sigma}(s, v))$. Further, by the definition of $\oplus$, we have $h_j(s, v) = \bigoplus_{q=1}^{j} \hat{h}_j(s, v)$.

Let $T_j(s, v)$ be the state of $T(s, v)$ after the completion of $j - 1$ iterations of the loop from line 3. We show by induction on j that $T_j(s, v) = h_j(s, v)$ and $\mathcal{T}_j(s, v) = \hat{h}_j(s, v)$, and subsequently that after $d - 1$ loop iterations, $T(s, v) = T_d(s, v) = h_d(s, v)$. For $j = 1$, no iterations have completed and we have $T(s, v) = (A(s, v), 1)$, as desired. For the inductive step, we show that given $T_j(s, v) = h_j(s, v)$, one iteration of the loop on line 3 yields $T_{j+1}(s, v) = h_{j+1}(v)$. We note that by definition of $\oplus$, only paths with a minimal weight

$\mathcal{T}_j(s,u).w$ contribute to $\mathcal{T}_{j+1}(s,v)$, and (again by definition of $\oplus$),

$$\mathcal{T}_{j+1}(s,v).m = \sum_{w \in P} w.m, \quad \text{where}$$
$$P = \{\mathcal{T}_j(s,u) : \mathcal{T}_j(s,u).w + A(u,v) = \mathcal{T}_{j+1}(s,v).w\},$$

i.e., $\mathcal{T}_{j+1}(s,v).m$ is the sum of the multiplicities of all the minimal weight paths from vertex s to v consisting of $j+1$ edges. Our expression for P is valid, since each must consist of a minimal weight path of k edges from vertex s to some vertex u, which is given by $\mathcal{T}_j(s,u)$ and another edge (u,v) with weight $A(u,v)$. □

4.2 MFBr: Computing Centrality Scores

Once we have obtained the distances and multiplicities of shortest paths from a set of starting vertices via MFBF, we can begin computing the partial centrality scores. We perform this by traversing the shortest-path tree from the leaves to the root. This time, the maximal frontier is composed of all vertices whose leaves have just reported their centrality scores.

4.2.1 Centpath Monoid.

Intuition: To propagate partial centrality scores, we use centpaths, which store a distance, a contribution to the centrality score, and a counter. Similarly to the multpath monoid, we define a centpath monoid $(\mathbb{C}, \otimes)$ with an operator that acts on any two centpaths x and y, and returns the one with lower weight; if the weights of x and y are equal, then the partial centrality factors and counter values of the two centpaths are summed.

Formalism: Instead of working with partial centrality scores $\delta(s,v)$ (defined in Section 2.4) we work with partial centrality factors (as in [37]):

$$\zeta(s,v) = \delta(s,v)/\bar{\sigma}(s,v) = \sum_{w \in \pi(s,v)} \left(\frac{1}{\bar{\sigma}(s,w)} + \zeta(s,w)\right).$$

Computing ζ rather than δ simplifies the algebraic steps done by the algorithm and leads to a simpler proof of correctness. Once we have computed ζ, we can construct δ simply via multiplication by elements of $\bar{\sigma}$, which we have already computed via MFBF.

A centpath $x = (x.w, x.p, x.c) \in \mathbb{C} = \mathbb{W} \times \mathbb{R} \times \mathbb{Z}$ is a path with a weight $x.w \in \mathbb{W}$, partial centrality score $x.p \in \mathbb{R}$, and a counter $x.c \in \mathbb{Z}$. Our algorithm will converge to a centpath x for each pair of starting and destination nodes s, v, where the partial dependency factor $x.p = \zeta(s,v)$. The counter $x.c$ is used to keep track of the number of predecessors who have not propagated a partial dependency factor up to the node v in a previous iteration. The counter is decremented until reaching zero, at which point the final centrality scores of all predecessors have been integrated into $x.p$ and it is then propagated from v up to the root s.

The centpath monoid operator $\otimes$ is defined as

$$\forall_{x,y \in \mathbb{C}}, \quad x \otimes y = \begin{cases} x & : x.w > y.w \\ y & : x.w < y.w \\ (x.w,\ x.p + y.p,\ x.c + y.c) & : x.w = y.w. \end{cases}$$

4.2.2 Brandes Action.

Intuition: Our MFBr algorithm (Algorithm 2) iteratively updates a matrix Z of centpaths via backward propagation of partial centrality factors from the leaves of the shortest path tree.

Algorithm 2 $[Z]$ = MFBr(A, T)

Input A: $n \times n$ adjacency matrix, T: matrix of distances and multiplicities
Output Z: centpath matrix of partial centrality factors ζ

```
    ▽ Existential qualifiers ∀s ∈ 1 : n_b (denoting starting vertices) are implicit and
    ∀v ∈ 1 : n (denoting intermediate vertices). ▽
    ▽ Initialize centpaths by finding counting predecessors ▽
 1: Z(s, v) := (T(s, v).w, 0, 1)
 2: Z := Z ⊗ (Z •⟨⊗,g⟩ A^T)
    ▽ Initialize centpath frontier ▽
 3: if Z(s, v).c = 0 then Z(s, v) := (T(s, v).w, 1/T(s, v).m, −1)
 4: else Z(s, v) = (∞, 0, 0)
 5: while Z(s, v) ≠ (∞, 0, 0) for some s, v do
 6:   Z := Z •⟨⊗,g⟩ A^T                ◃ Back-propagate frontier of centralities
    ▽ Turn off counters for nodes that already appeared in a frontier ▽
 7:   if Z(s, v).c = 0 then Z(s, v).c = −1
 8:   Z := Z ⊗ Z                      ◃ Accumulate centralities and increment counters
    ▽ Determine new frontier based on counters ▽
 9:   if Z(s, v).c = 0 then
10:     Z(s, v) := (T(s, v).w, Z(s, v).p + 1/T(s, v).m, −1)
11:   else Z(s, v) = (∞, 0, 0)
12: end while
```

Formalism: In the inner loop (lines 5-12), Z is computed with $\bullet_{\langle \otimes, g \rangle}$, where function g is defined as

$$g : \mathbb{C} \times \mathbb{W} \to \mathbb{C}, \qquad g(a, w) = (a.w - w,\ a.p,\ a.c).$$

g may be interpreted as an action of the monoid $(\mathbb{W}, +)$ on set $\mathbb{C}$.

4.2.3 Algorithm and Correctness.

Intuition: For weighted graphs, a single vertex may appear many times in the frontier as its shortest path information and multiplicity is corrected, unlike in traversals in BFS or Dijkstra's algorithms, where the total number of nonzeros in the matrix multiplication operand $\mathcal{T}$ sums to $(n-1)n_b$ over all iterations. For the Brandes step, we can avoid propagating unfinalized information as we already know the structure of the shortest path trees.

MFBr (Algorithm 2) propagates centrality factors optimally via the counter kept by each centpath, putting vertices in the frontier only when all of their predecessors have already appeared in previous frontiers. The counter is initialized to the number of predecessors, is decremented until reaching 0, added to a frontier and set to -1 to avoid re-adding the vertex to another frontier. This approach is strictly better than propagating partial centrality scores, which does not contribute to overall progress. Moreover, this scheme is much faster than using Dijkstra's algorithm to compute shortest-paths, since it requires the same number of iterations as Bellman Ford (Dijkstra's algorithm requires $n-1$ matrix multiplications).

Formalism: We can demonstrate correctness of the algorithm by showing that the counter mechanism serves to correctly define each frontier in the shortest path tree.

Lemma 4.2. *For any adjacency matrix A and a multpath matrix T containing shortest path distances and multiplicities, $Z = MFBr(A, T)$ satisfies $Z(s,v).p = \zeta(s,v)$.*

Proof. We prove that the partial BC scores are computed correctly after $d-1$ iterations of the loop in line 5 if all shortest paths from $\vec{s}$ in G consist of at most d edges. As before, we denote the shortest distance from node s to v as $\tau(s,v)$ and the multiplicity as $\bar{\omega}(s,v)$. We define $k_j(s,v) \in \mathcal{Z}$ as the sum of all minimal distance paths of at most $j-1$ edges from s ending at u that are on the

minimal distance path between 1 and v,

$$k_j(s,v) = \sum_{(u,\star)\in P_j(s,v)} \frac{1}{\bar{\sigma}(s,u)}, \text{ where}$$
$$P_j(s,v) = \{(u,\vec{w}) | l \in 1:j-1, \vec{w} \in (1:n)^l,$$
$$\tau(s,u) + A(u,w_1) + A(w_1,w_2) + \ldots + A(w_l,v) = \tau(s,v)\}$$

Since $P_d(s,v)$ is the set of all shortest paths between s and v that are parts of shortest paths between s and u, for each u there are $\sigma(s,u,v)/\bar{\sigma}(u,v)$ such paths, and therefore,

$$k_d(s,v) = \sum_{(u,\star)\in P_d(s,v)} \frac{1}{\bar{\sigma}(s,u)} = \sum_{u=1}^{n} \frac{\sigma(s,u,v)}{\bar{\sigma}(s,u)\bar{\sigma}(u,v)}.$$

We now show that $k_j(s,v)$ can be expressed in terms of $k_{j-1}(s,u)$ for all $u \in P_1(v)$ (the 1-edge shortest-path neighborhood of v from s). We accomplish this by disjointly partitioning $P_j(s,v)$ into $P_1(s,v)$ and $\bigcup_{u\in P_1(s,v)} P_{j-1}(s,u)$, which yields,

$$k_j(s,v) = \sum_{(u,\star)\in P_1(s,v)} \left(\frac{1}{\bar{\sigma}(s,u)} + \sum_{(w,\star)\in P_{j-1}(s,u)} \frac{1}{\bar{\sigma}(s,w)}\right)$$
$$= \sum_{(u,\star)\in P_1(s,v)} \left(\frac{1}{\bar{\sigma}(s,u)} + k_{j-1}(s,u)\right)$$

Let $Z_j(s,v)$ be the state of $Z(s,v)$ and $\mathcal{Z}_j(s,v)$ be the state of $\mathcal{Z}(s,v)$ after the completion of $j-1$ iterations of the loop on line 5. We argue by induction on j, that for all $j \in 1:d$, $Z_j(s,v).p = k_j(s,v) = k_d(s,v)$ and

$$\mathcal{Z}_j(s,v) = (\tau(s,v), 1/\bar{\sigma}(s,v) + \zeta(s,v), -1)$$

if and only if the largest number of edges in any shortest path from any node u to v, such that $\tau(s,v) = \tau(s,u) + \tau(u,v)$, is $j-1$. In the base case, $j = 1$ and thus $Z_1(s,v).p = k_j(s,v) = k_d(s,v) = \zeta(s,v) = 0$ for all vertices v with no predecessors (leaves in the shortest path tree); these vertices are set appropriately in $\mathcal{Z}_1$.

For the inductive step, the update on line 5 contributes the appropriate factor of $\frac{1}{\bar{\sigma}(s,u)} + k_{j-1}(u)$ from each predecessor vertex u. Next, each such predecessor vertex u must have been a member of a single frontier by iteration j, since the larger number of edges in any shortest path from u to any node v must be no greater than $j-1$. Therefore, the counter $Z_j(s,v).c = 0$, which means $\mathcal{Z}_j(s,v)$ is set appropriately (for subsequent iterations $k > j$, $\mathcal{Z}_k(s,v) = (\infty,0,0)$ since we set $Z_j(s,v).c = -1$ at iteration $j+1$). □

4.3 Combined BC Algorithm

To obtain a complete algorithm for BC, we combine MFBF and MFBr into MFBC (Algorithm 3). MFBC is parametrized with a batch size n_b and proceeds by computing MFBF and MFBr to obtain partial centrality factors for n_b starting vertices at a time. These factors are then appropriately scaled by multiplicities ($\bar{\sigma}(s,v)$) and accumulated into a vector of total centrality scores.

Theorem 4.3. *For any adjacency matrix A and $n_b \in 1:n$, $\lambda = MFBC(A,n_b)$ satisfies $\lambda(v) = \sum_{s,t\in V} \frac{\sigma(s,t,v)}{\bar{\sigma}(s,t)}$.*

Proof. We assume $n \mod n_b = 0$, if it does not hold then $n \mod n_b$ disconnected vertices can be added to G without changing λ. For each vertex batch, MFBF computes the correct shortest distances and multiplicities T by Lemma 4.1. For each T, MFBr computes the correct partial centrality scores Z by Lemma 4.2.

Algorithm 3 $[\lambda] = \text{MFBC}(A)$

Input A: $n \times n$ adjacency matrix, n_b: the batch size
Output λ: a vector of BC scores
1: $\forall v \in 1:n, \quad \lambda(v) := 0$ ◃ Initialize the BC scores
2: **for** $i \in 1 : n/n_b$ **do**
3: $\quad [T] = \text{MFBF}(A, (i-1)n_b + 1 : in_b)$
4: $\quad [Z] = \text{MFBr}(A, T)$
$\quad \nabla$ Accumulate partial centralities: $\delta(s,v) = \zeta(s,v) \cdot \bar{\sigma}(s,v)$ ∇
5: $\quad \forall v \in 1:n, \quad \lambda(v) := \lambda(v) + \sum_{s=1}^{n_b} Z(s,v).p \cdot T(s,v).m$
6: **end for**

Thus, at iteration i, $T(s,v).m = \bar{\sigma}((i-1)n_b + s, v)$ and $Z(s,v).p = \zeta((i-1)n_b + s, v)$. Next, over all iterations, line 5 expands to

$$\lambda(v) = \sum_{s=1}^{n} Z(s,v).p \cdot T(s,v).m = \sum_{s\in V} \zeta(s,v) \cdot \bar{\sigma}(s,v)$$
$$= \sum_{s\in V} \delta(s,v) = \sum_{s\in V} \sum_{t\in V} \frac{\sigma(s,t,v)}{\bar{\sigma}(s,t)}.$$

□

5 COMMUNICATION COMPLEXITY

MFBC leverages all the available parallelism in the problem to accelerate overall progress. We now formally study its scalability by bounding its communication complexity. We first present a cost model (Section 5.1). In Section 5.2 we derive the communication costs for sparse matrix multiplication, by far the most expensive operation in MFBC. We assume sparse matrices with arbitrary dimensions and nonzero count. In several cases, to concretize the model and derive tighter bounds, we use matrix multiplication (tensor contraction) routines in CTF. However, this does not limit our analysis as CTF employs a larger space of sparse matrix multiplication variants than any previous work. Our results provide a communication bound that is substantially lower than previous results for sparse matrix multiplication when the number of nonzeros is imbalanced between matrices. As this scheme is a critical primitive not only in graph algorithms, but in numerical algorithms such as multigrid, this theoretical result is of stand-alone importance.

Finally, in Section 5.3, we express the cost of MFBC in terms of the communication cost of the sparse matrix multiplications it executes. Our analysis shows that the latency (synchronization) cost of MFBC may in several cases be higher than the best-known all-pairs shortest-path algorithms by a factor proportional to the number of batches n/n_b, which in turn depends on the available memory. Simultaneously, MFBC can operate with $O(m/p)$ memory per processor, while all-pairs shortest-paths algorithm require $\Omega(n^2/p)$ memory per processor. Finally, the communication bandwidth cost of MFBC is identical or better than all the used comparison targets.

5.1 Cost Model

We use a parallel execution model where we count the number of messages and amount of data communicated by any processor. We do not keep track of the number of CPU operations because, for sparse matrix multiplication, all the considered algorithms have an optimal computation cost, and for BC our algorithm is work-optimal in the unweighted case. The computation cost in the weighted case depends on the number of times each vertex appears in a frontier during the MFBC execution, which depends on the graph connectivity as well as the edge weights.

We use the $\alpha - \beta$ model [39] where the latency of sending a message is α and the inverse bandwidth is β. We assume that $\alpha \geq \beta$. There are p processes and M (number of words) is the size of a local memory at every processor. Next, the cost of collective communication routines (scatter, gather, broadcast, reduction, and allreduction) on p processors in the $\alpha - \beta$ model is $O(\beta \cdot x + \alpha \cdot \log p)$ [3]; x is the maximum number of words that each processor owns at the start or end of the collective. Furthermore, the cost of a *sparse reduction* where each processor inputs a sparse array and the resulting array has x nonzeros is also $O(\beta \cdot x + \alpha \cdot \log p)$. Finally, we use $\text{nnz}(X)$ to denote the number of non-zeros in any matrix X and $\text{flops}(X, Y)$ to denote the number of nonzero products when multiplying sparse matrices X and Y.

5.2 Parallel Sparse Matrix Multiplication

We first analyze the product of sparse matrices $A^{m\times k}$ and $B^{k\times n}$ that produce matrix $C^{m\times n}$. We use algorithms based on 1-, 2-, and 3-dimensional matrix decomposition. They all have a computation cost of $O(\text{flops}(A, B)/p)$, which we omit.

All the considered algorithms and implementations use matrix blocks that correspond to the cross product of a subset of columns and a subset of rows of the matrices. The blocks are chosen obliviously of the matrix structure. For sparse matrices with a sufficiently large number of nonzeros, randomizing the row and column order implies that the number of nonzeros of each such block is proportional to the block size. Thus, we assume that the number of nonzeros in any block of dimensions $b_1 \times b_2$ of a sparse matrix $A^{m\times k}$ has $O(\text{nnz}(A)b_1b_2/(mk))$ nonzeros so long as $mk/(b_1b_2) \leq p$. When $\text{nnz}(A) \gg p$, the assumption holds true with high probability based on a balls-into-bins argument [41].

We also assume that multiplying any two blocks of equal dimensions yields about the same number of nonzero products and output nonzeros. This assumption does not impact our communication-cost analysis, but allows us to assert that the computational work is asymptotically load-balanced. Thus, when multiplying blocks of size $b_1 \times b_2$ of matrix A and $b_2 \times b_3$ of matrix B, the number of nonzero operations is $O(\text{flops}(A, B)b_1b_2b_3/(mnk))$ and the number of nonzeros in the output block contribution to matrix C is

$$O\left(\min\left[\frac{\text{nnz}(C)}{mn}b_1b_3, \frac{\text{flops}(A, B)}{mnk}b_1b_2b_3\right]\right).$$

For a sparse matrix corresponding to uniform random graphs, the respective numbers are

$$\text{flops}(A, B) \approx \frac{\text{nnz}(A)}{mk}\frac{\text{nnz}(B)}{kn}mnk = \text{nnz}(A)\,\text{nnz}(B)/k$$

and $\text{nnz}(C) \approx \min(mn, \text{flops}(A, B))$. We can invoke the same balls-into-bins argument [41] to argue that the work load-balance assumption holds, by arguing about the induced layout of the $\text{flops}(A, B)$ operations as a third-order cyclically distributed tensor.

5.2.1 1D Algorithms. 1D decompositions are the simplest way to parallelize a matrix multiplication. There are three variants, each of which replicates one of the matrices and blocks the others into p pieces. Variant A replicates A via broadcast and assigns each processor a set of columns of B and C. Variant B broadcasts B and assigns each processor a set of rows of A and C. Variant C assigns each processor a set of columns of A and rows of B, computes their product, and uses a reduce to obtain C. The communication cost of version X of a 1D algorithm for $\text{X} \in \{\text{A}, \text{B}, \text{C}\}$ is

$$W_{\text{X}}(X, p) = O(\alpha \cdot \log p + \beta \cdot \text{nnz}(X)).$$

5.2.2 2D Algorithms. 2D algorithms [1, 13, 45] block all matrices on a grid of $p_r \times p_c$ processors and move the data in steps to multiply matrices. 2D algorithms can be based on point-to-point or collective communication. The former are up to $O(\log p)$ faster in latency, but the latter generalize easier to rectangular processor grids. The algorithms are naturally extended to handle sparse matrices by treating the matrix blocks as sparse [6, 12]. One of the simplest 2D algorithms is Cannon's algorithm [13], which shifts blocks of A and B on a square processor grid, achieving a communication cost of

$$O\left(\alpha \cdot \sqrt{p} + \beta \cdot \frac{\text{nnz}(A) + \text{nnz}(B)}{\sqrt{p}}\right).$$

The algorithm is optimal for square matrices, but other variants achieve lower communication cost when the number of nonzeros in the two operand matrices are different.

Our implementation uses three variants of 2D algorithms using broadcasts and (sparse) reductions. The variant AB broadcasts blocks of A and B along processor grid rows and columns, while the variants AC and BC reduce C and broadcast A and B, respectively. CTF uses $\text{lcm}(p_r, p_c)$ (lcm is the least common multiple) broadcasts/reductions and adjusts p_r and p_c so that $\text{lcm}(p_r, p_c) \approx \max(p_r, p_c)$ steps of collective communication are performed. When each matrix block is sparse with the specified load balance assumptions, the costs achieved by these 2D algorithms are given in general by W_{YZ} for variants $\text{YZ} \in \{\text{AB}, \text{AC}, \text{BC}\}$ as $W_{\text{YZ}}(Y, Z, p_r, p_c) =$

$$O\left(\alpha \cdot \max(p_r, p_c)\log(p) + \beta \cdot \left(\frac{\text{nnz}(Y)}{p_r} + \frac{\text{nnz}(Z)}{p_c}\right)\right)$$

5.2.3 3D Algorithms. While 2D algorithms are natural from a matrix-distribution perspective, the dimensionality of the computation suggests the use of 3D decompositions [1, 2, 9, 17, 25, 34], where each processor computes a subvolume of the mnk dense products. 3D algorithms have been adapted to sparse matrices, in particular by the Split-3D-SpGEMM scheme [3] that costs

$$O\left(\alpha \cdot \sqrt{cp}\log p + \beta \cdot \left(\frac{\text{nnz}(A) + \text{nnz}(B)}{\sqrt{cp}} + \frac{\text{flops}(A, B)}{p}\right)\right)$$

by using a the grid of processes that is $\sqrt{p/c} \times \sqrt{p/c} \times c$.

We derive 3D algorithms (and implement in CTF) by nesting the three 1D algorithm variants with the three 2D algorithm variants. The cost of the resulting nine 3D variants on a $p_1 \times p_2 \times p_3$ processor grid with the 1D algorithm applied over the first dimension is

$$W_{\text{X,YZ}}(X, Y, Z, p_1, p_2, p_3) = W_{\text{X}}(X[p_2, p_3]) + \begin{cases} W_{\text{YZ}}(Y, Z[p_1], p_2, p_3) & : X = Y, \\ W_{\text{YZ}}(Y[p_1], Z, p_2, p_3) & : X = Z, \\ W_{\text{YZ}}(Y[p_1], Z[p_1], p_2, p_3) & : X \notin \{Y, Z\}, \end{cases}$$

for $(\text{X}, \text{YZ}) \in \{\text{A}, \text{B}, \text{C}\} \times \{\text{AB}, \text{AC}, \text{BC}\}$, with notation $X[p_2, p_3]$ denoting that the 1D algorithm operates on blocks of X given from a $p_2 \times p_3$ distribution, while $Y[p_1]$ and $Z[p_1]$ refer to 1D distributions.

This cost simplifies to

$$W_{\text{X,YZ}}(X, Y, Z, p_1, p_2, p_3) =$$
$$O\left(\alpha \cdot \max(p_1, p_2)\log(\min(p_1, p_2)) + \beta \cdot \frac{\text{nnz}(X)}{p_2 p_3}\right)$$
$$+ \begin{cases} O\left(\beta \cdot (\frac{\text{nnz}(X)}{p_2} + \frac{\text{nnz}(Z)}{p_1 p_3})\right) & : X = Y, \\ O\left(\beta \cdot (\frac{\text{nnz}(Y)}{p_1 p_2} + \frac{\text{nnz}(X)}{p_3})\right) & : X = Z, \\ O\left(\beta \cdot (\frac{\text{nnz}(Y)}{p_1 p_2} + \frac{\text{nnz}(Z)}{p_2 p_3})\right) & : X \notin \{Y, Z\}. \end{cases}$$

The amount of memory used by this algorithm is

$$M_{\text{X,YZ}}(X, Y, Z, p, p_1) = O\left(\frac{\text{nnz}(X)p_1}{p} + \frac{\text{nnz}(Y) + \text{nnz}(Z)}{p}\right).$$

As we additionally consider pure 1D and 2D algorithms, then pick the 1D, 2D, or 3D variant of least cost. Provided unlimited memory, the execution time of our sparse matrix multiplication scheme is no greater than

$$W_{\text{MM}}(A, B, C, p) = O\Bigg(\min_{\substack{p_1,p_2,p_3\in\mathbb{N} \\ p_1p_2p_3=p}} \Bigg[\alpha \cdot \max(p_1, p_2, p_3)\log p$$
$$+ \beta \cdot \left(\frac{\text{nnz}(A)}{p_1 p_2}\delta(p_3) + \frac{\text{nnz}(B)}{p_2 p_3}\delta(p_1) + \frac{\text{nnz}(C)}{p_1 p_3}\delta(p_2)\right)\Bigg]\Bigg),$$

where $\delta(x) = 1$ when $x \geq 1$ and $\delta(1) = 0$.

5.3 Parallel Betweenness Centrality

We finally use the matrix multiplication analysis to ascertain a communication cost bound for MFBC, which performs the bulk of the computation via generalized matrix multiplication. We focus on unweighted graphs; the associated proofs heavily use the fact that each vertex appears in a unique frontier. We then (Section 5.3.1) discuss weighted graphs; in this case we cannot anymore use the same technique to ascertain bounds on the size of each frontier.

Theorem 5.1. *For any unweighted n-node m-edge graph G with adjacency matrix A and diameter d, on a machine with p processors with $M = \Omega(cm/p)$ words of memory each, for any $c \in [1, p]$, MFBC (Algorithm 3) can execute with communication cost*

$$W_{MFBC}(n, m, p, c) = O\left(\alpha \cdot \frac{dn^2}{m}\sqrt{p/c^3}\log p + \beta \cdot \left(\frac{n^2}{\sqrt{cp}} + \frac{n\sqrt{m}}{p^{2/3}}\right)\right).$$

Proof. MFBC is dominated in computation and communication cost by multiplications of sparse matrices, triggered by the operator $\bullet_{\langle\oplus,f\rangle}$ within MFBF and $\bullet_{\langle\otimes,g\rangle}$ within MFBr. There are up to $2d + 1$ such matrix multiplications in total. Without loss of generality, we consider only the d within MFBF, letting F_i be the frontier ($\mathcal{T}$) at iteration i and G_i be the output of $\mathcal{T} \bullet_{\langle\oplus,f\rangle} A$, which include F_{i+1} but can be much denser. We can then bound the cost of MFBC as

$$W_{\text{MFBC}} = O\left(\min_{n_b\in 1:n}\left[\frac{n}{n_b}\sum_{i=1}^{d} W_{\text{MM}}(A, F_i, G_i, p)\right]\right)$$
$$= O\Bigg(\min_{n_b\in 1:n}\frac{n}{n_b}\sum_{i=1}^{d}\min_{\substack{p_1,p_2,p_3\in\mathbb{N} \\ p_1p_2p_3=p}}\Bigg[\alpha \cdot \max(p_1, p_2, p_3)\log p$$
$$+ \beta \cdot \left(\frac{m}{p_1 p_2}\delta(p_3) + \frac{\text{nnz}(F_i)}{p_2 p_3}\delta(p_1) + \frac{\text{nnz}(G_i)}{p_1 p_3}\delta(p_2)\right)\Bigg]\Bigg).$$

The MFBC algorithm requires $O(nn_b/p)$ memory to store T, therefore, we have $nn_b/p = O(M)$, and select $n_b = cm/n$,

$$W_{\text{MFBC}} = O\Bigg(\frac{n^2}{cm}\sum_{i=1}^{d}\min_{\substack{p_1,p_2,p_3\in\mathbb{N} \\ p_1p_2p_3=p}}\Bigg[\alpha \cdot \max(p_1, p_2, p_3)\log p$$
$$+ \beta \cdot \left(\frac{m}{p_1 p_2}\delta(p_3) + \frac{\text{nnz}(F_i)}{p_2 p_3}\delta(p_1) + \frac{\text{nnz}(G_i)}{p_1 p_3}\delta(p_2)\right)\Bigg]\Bigg).$$

We use a 3D algorithm with $p_1 = p_2 = \sqrt{p/c}$, $p_3 = c$, which replicates A via a 1D algorithm, then employs the BC variant of a 2D algorithm, using $O(cm/p)$ memory. A's replication can be amortized over (up to d) sparse matrix multiplications and over the $\frac{n^2}{cm}$ batches, since A is always the same adjacency matrix. Thus,

$$W_{\text{MFBC}} = O\Bigg(\beta \cdot \frac{cm}{p} + \frac{n^2}{cm}\Bigg(\Big(\sum_{i=1}^{d}\Big[\alpha \cdot \sqrt{p/c}\log p$$
$$+ \beta \cdot \left(\frac{\text{nnz}(F_i)}{\sqrt{pc}} + \frac{\text{nnz}(G_i)}{\sqrt{pc}}\right)\Big]\Bigg)\Bigg),$$

and furthermore, over all n_b batches the total cost is

$$W_{\text{MFBC}} = O\left(\alpha\frac{dn^2}{m}\sqrt{\frac{p}{c^3}}\log p + \beta\left[\frac{cm}{p} + \sum_{i=1}^{d}\frac{n^2(\text{nnz}(F_i) + \text{nnz}(G_i))}{m\sqrt{pc^3}}\right]\right).$$

Now, since the graph is unweighted, we know that each vertex appears in a unique frontier, so $\sum_{i-1}^{d}\text{nnz}(F_i) \leq nn_b = cm$. Therefore, each node can be reached from 3 frontiers (the one it is a part of, the previous one, and the subsequent one), therefore $\sum_{i-1}^{d}\text{nnz}(G_i) \leq 3cm$. Then, the total bandwidth cost over all d iterations and cm/n batches is $O(\beta \cdot (n^2/\sqrt{cp} + \frac{cm}{p}))$. This cost is minimized for $c = p^{1/3}n^2/m$, so with $M = \Omega(n^2/p^{2/3})$ memory, the cost $O(\beta \cdot n\sqrt{m}/p^{2/3})$ can be achieved. □

5.3.1 Discussion on Weighted Graphs. Our communication cost analysis can be extended to weighted graphs, provided bounds on $\sum_i \text{nnz}(F_i)$ and $\sum_i \text{nnz}(G_i)$ for each batch. The quantity $\sum_i \text{nnz}(F_i)$ can be bounded given an amplification factor bounding the number of Bellman-Ford iterations in which the shortest path distance between any given pair of source and destination vertices is changed. However, we do not see a clear way to bound $\sum_i \text{nnz}(G_i)$ for weighted graphs. We evaluate MFBC for weighted graphs in the subsequent section, observing a slowdown proportional to the factor of increase in the number of iterations with respect to the unweighted case (in the unweighted case it is the diameter d).

5.3.2 Comparison to Other Analyses. We are not aware of other communication cost studies of BC, but we can compare our approach to those computing the full distance matrix via all-pairs shortest-paths (APSP) algorithms, requiring at least n^2/p memory, regardless of m. The best-known APSP algorithms leverage 3D matrix multiplication to obtain a bandwidth cost of $O(\beta \cdot n^2/\sqrt{cp})$ using $O(cn^2/p)$ memory for any $c \in [1, p^{1/3}]$ [44]. MFBC matches this bandwidth cost, while using only $O(cm/p)$ memory. Further, given sufficient memory $M = \Omega(n^2/p^{2/3})$, our algorithm is up to $\min(n/\sqrt{m}, p^{2/3})$ faster. When also considering an algorithm that replicates the graph as an alternative, the best speed-up achievable by MFBC is for $M = \Theta(n^2/p^{2/3})$ memory with $n/\sqrt{m} = p^{1/3}$, and when $\beta \gg \alpha$, in which case $W_{\text{MFBC}}(n, n^2/p^{2/3}, p, p^{2/3}) =$

$O(\beta \cdot n^2/p)$ is $p^{1/3}$ times faster than Floyd-Warshall, path doubling, or Dijkstra with a replicated graph.

5.3.3 Discussion on Latency. The Floyd-Warshall APSP algorithm has latency cost $O(\alpha \cdot \sqrt{cp})$, but a path-doubling scheme can achieve $O(\alpha \cdot \log p)$ [44] using $O(n^2/p^{2/3})$ memory. Given this amount of memory, MFBC can achieve a latency cost of

$$O\left(\alpha \cdot d \log p\left(\frac{n^2}{\sqrt{p}m} + \sqrt{n/\sqrt{m}}\right)\right).$$

It might be possible to improve this latency cost by using different sparse matrix multiplication algorithms.

5.3.4 Discussion on Scalability. The capability of our algorithm to employ large replication factors c gives it good strong scalability properties. If each processor has $M = O(m/p_0)$ memory, it is possible to achieve perfect strong scalability in bandwidth cost using up to $p_0^{3/2}n^3/m^{3/2}$ processors, while for up to $p_0^{3/2}n^2/m$,

$$W_{\text{MFBC}}(n, m, cp_0, c) = \frac{1}{c}W_{\text{MFBC}}(n, m, p_0, 1)$$

is satisfied, so strong scalability is achieved in all costs from p_0 to $p_0^{3/2}n^2/m$ processors. This range in strong scalability is better than that achieved by the best known square dense matrix multiplication algorithms, p_0 to $p_0^{3/2}$ [40].

6 IMPLEMENTATION

We implement two parallel versions of MFBC using CTF. The first, CTF-MFBC, uses CTF to dynamically select data layouts without guidance from the developer. The second, CA-MFBC, predefines the 3D processor grid layout that we used to minimize theoretical communication cost in the proof of Theorem 5.1. We first summarize the functionality of CTF and explain how it provides the sparse matrix operations necessary for MFBC. We then give more details on how CTF handles data distribution and communication.

6.1 From Algebra to Code

CTF permits definition of all well-known algebraic structures and implements tensor contractions with user-defined addition and multiplication operators [41]. Matrices can encode graphs and subgraphs (frontiers); tensors of order higher than two can represent hypergraphs. As graphs are sufficient for the purposes of this paper, we refer only to CTF matrix operations. An $n \times n$ CTF matrix is distributed across a `World` (an MPI communicator), and has attributes for symmetry, sparsity, and the algebraic structure of its elements. We work with adjacency matrices with weights in a set `W`

```
Matrix<W> A(n,n,SP,D,Y);
```

where `D` is a `World` and `Y` defines the `Monoid<W>` of weights with minimum as the operator.

CTF permits operations on one, two, or three matrices at a time, each of which is executed bulk synchronously. To define an operation, the user assigns a pair of indices (character labels) to each matrix (generally, an index for each mode of the tensor). An example function inverting all elements of a matrix `A` and storing them in `B` is expressed as

```
Function<int,float>([](int x){ return 1./x; })
B["ij"] = f(A["ij"]);
```

All CTF operations may be interpreted as nested loops, where one operation is performed on elements of multidimensional arrays in the innermost loop. For instance, in terms of loops on arrays `A` and `B`, the above example is

```
for (int i=0; i<n; i++)
  for (int j=0; j<n; j++) {B[i,j] = 1./A[i,j];}
```

For contractions, we can define functions with two operands.

We express $\bullet_{\langle\oplus,f\rangle}$ from Section 4.1.2 by defining functions `u` for operation $\oplus$ and `f` for f, and then a `Kernel` corresponding to $\bullet_{\langle\oplus,f\rangle}$

```
Kernel<W,M,M,u,f> BF;
Z["ij"]=BF(A["ik"],Z["kj"]);
```

If `Z` is a matrix with each element in `M` and `A` is the adjacency matrix with elements in `W`, the above CTF operation executes $Z = A\bullet_{\langle\oplus,f\rangle}Z$. One could supply the algebraic structure in a `Monoid` when defining the matrix, then use `Function` in place of a `Kernel`. However, the latter construct parses the needed user-defined functions as template arguments rather than function arguments, enabling generation of more efficient (sparse) matrix multiplication kernels for blocks. Having these alternatives enables the user to specify which kernels are intensive and should be optimized thoroughly at compile time, while avoiding unnecessary additional template instantiations.

Other CTF constructs employed by our MFBC code are

- `Tensor::write()` to input graphs bulk synchronously,
- `Tensor::slice()` to extract subgraphs,
- `Tensor::sparsify()` to filter the next frontier,
- `Transform` to modify matrix elements with a function.

More information on the scope of operations provided by CTF is detailed by Solomonik and Hoefler [41].

6.2 Data Distribution Management

CTF enables the user to work obliviously of the data distribution of matrices. Each created matrix is distributed over all processors using a processor grid that makes the block dimensions owned by each processor as close to a square as possible. For each operation (e.g., sparse matrix multiplication), CTF seeks an optimal processor grid, considering the space of algorithms described in Section 5.2 as well as overheads, such as redistributing the matrices.

Transitioning between processor grids and other data distributions are achieved using three kernels: (1) block-to-block redistribution, (2) dense-to-dense redistribution, (3) sparse-to-sparse redistribution. Kernel (1) is used for reassigning blocks of a dense matrix to processors on a new grid, (2) is used for redistributing dense matrices between any pair of distributions, and (3) is used for reshuffling sparse matrices and data input. After redistribution, the matrix/tensor data is transformed to a format suitable for summation, multiplication, or contraction. For dense matrices, this involves only a transposition, but for sparse matrices, CTF additionally converts data stored as index–value pairs (coordinate format) to a compressed-sparse-row (CSR) matrix format.

CTF uses BLAS [30] routines for blockwise operations whenever possible (for the datatypes and algebraic operations provided by BLAS). We additionally use the Intel MKL library to multiply sparse matrices, with three variants: one sparse operand, two sparse operands, and two sparse operands with a sparse output. Substitutes for these routines are provided in case MKL is not available.

Further, for special algebraic structures or mixed-type contractions, general unoptimized blockwise multiplication and summation routines are used. Users can also provide manually-optimized routines for blockwise operations, which we do not leverage for MFBC.

CTF predicts the cost of communication routines, redistributions, and blockwise operations based on linear cost models. Besides latency α and bandwidth β, CTF also considers the memory bandwidth cost and computation cost of redistribution and blockwise operations. The dimensions of the submatrices on which all kernels are executed for a given mapping can be derived at low cost a priori. To determine sparsity of blocks, we scale by either the nonzero fraction of the operand matrix or the estimated nonzero fraction of the output matrix. Automatic model tuning allows the cost expressions of different kernels to be comparable on any given architecture. CTF employs a model tuner that executes a wide set of benchmarks on a range of processors, designed to make use of all kernels for various input sizes. Tuning is done once per architecture or whenever a kernel is added or significantly modified.

7 EVALUATION

We present performance results for the the CTF (unmodified v1.4.2) implementation of MFBC (CTF-MFBC) and for CombBLAS. Our benchmarks use the CombBLAS betweenness centrality code and benchmark included in the CombBLAS v1.5.0 distribution. on the Blue Waters Cray XE6 supercomputer. We provide a mix of performance benchmark tests, working with different types of graphs:

(1) real-world social-network and citation graphs [31],
(2) synthetic weighted and unweighted R-MAT graphs [14],
(3) Erdős-Rényi random graphs [22].

Real-world graphs and R-MAT graphs serve to provide effective strong scaling experiments. We consider different types of weak scaling experiments using uniform graphs. For *edge-weak scaling*, we keep the number of edges per processor and the nonzero fraction constant. For *vertex-weak scaling*, we keep the number of vertices per processor and the average degree constant.

CTF-MFBC achieves consistent scalability patterns for all types of graphs. It outperforms CombBLAS on various tests, but not uniformly so. Large speed-ups are achieved for real, R-MAT, and uniform random graphs with the average vertex degree above 100.

7.1 Experimental Setup

To debug and tune our code, we used the NERSC Edison supercomputer, a Cray XC30 as well as the CSCS Piz Dora machine, a Cray XC40. Each Edison compute node has two 12-core HT-enabled Intel Ivy Bridge sockets with 64 GiB DDR3-1866 RAM. Each node of Piz Dora has two 18-core Intel Broadwell CPUs (Intel® Xeon® E5-2695 v4). The network is the same on both of these machines, a Cray's Aries implementation of the Dragonfly topology [27]. We then collected our final set of benchmarks on Blue Waters, a Cray XE6 supercomputer. Each Blue Waters XE node has two 16-core AMD 6276 Interlagos sockets; there are 22,500 nodes in total. The network is a Cray Gemini torus. The performance-portability of CTF made it easy to transition the code between these machines. Our choice of machines was based on resource availability.

For both CTF and CombBLAS, we benchmarked a range of batch-sizes for each graph and processor count. We show the highest performance rate over all batch sizes, which was usually achieved by the largest batch-size that still fit in memory. Such batches run for on the order of minutes, so we executed each batch only once, rather than testing many iterations. Due to the large overall time-granularity of the benchmarks, we observed that system noise did not effect the runtime in the first 2 significant digits.

We use the metric of edge traversals per second (TEPS) to quantify performance. The number of edge traversals scales with the size of the graph. For betweenness centrality on a connected unweighted graph, each edge is traversed to consider shortest paths from every starting node. We use all one MPI process per node and benchmark on core counts that are powers of four, as CombBLAS requires square processor grids. CTF can leverage threading and execute efficiently on most core-counts, but we maintain powers of four for all experiments for consistency and simplicity.

Considered graphs We used two classes of synthetic graphs: R-MAT (power-law) and random-uniform. We varied the density for both types of synthetic graphs. Generally these graphs have a low diameter, which roughly reflects social-network graphs, a key application domain of betweenness centrality. We also use real-world SNAP [31] graphs (Table 2) of various sparsities and diameters. Our CTF-MFBC code preprocessed all graphs to remove completely disconnected vertices.

ID	Name	directed?	n	m	d	$\bar{d}$
frd	Friendster	undirected	65.6M	1.8B	32	5.8
ork	Orkut social network	undirected	3.1M	117M	9	4.8
ljm	LiveJournal membership	directed	4.8M	70M	16	6.5
cit	Patent citation graph	directed	3.8M	16.5M	22	9.4

Table 2: The analyzed real-world graphs (sorted by m). All graphs are unweighted and have diameter d, with 90-percentile effective diameter $\bar{d}$ [31].

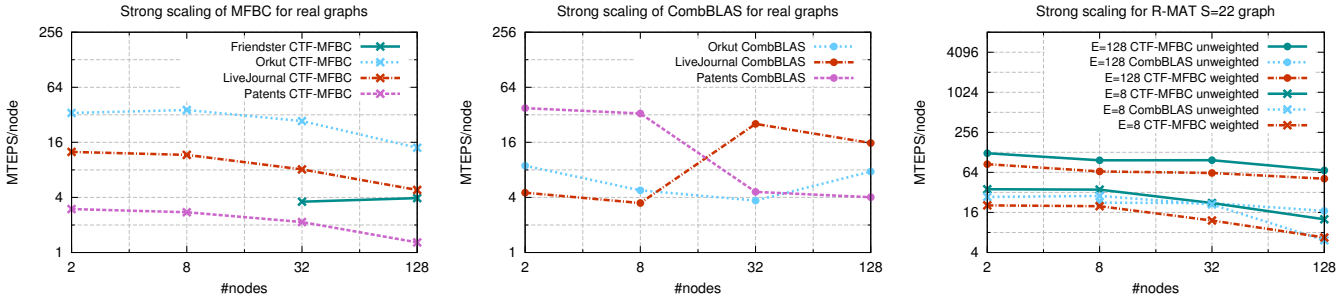
7.2 Strong Scaling

We begin our performance study by testing the ability of MFBC to lower time to solution by using extra nodes (strong scaling). Our strong scaling experiments evaluate MFBC with respect to CombBLAS on both real-world and synthetic graphs.

Real-world graphs Our selection of graphs (Table 2) considers three social-network graphs and a patent citation graph. Two of the graphs are directed and two are undirected. The diameter of the Orkut and LiveJournal graphs is relatively small, but Orkut is significantly denser. The patent citation graph has the largest diameter, posing the biggest challenge to betweenness centrality computation. The Friendster graph has roughly 15X more vertices and edges than any of the other graphs, and a fairly large diameter.

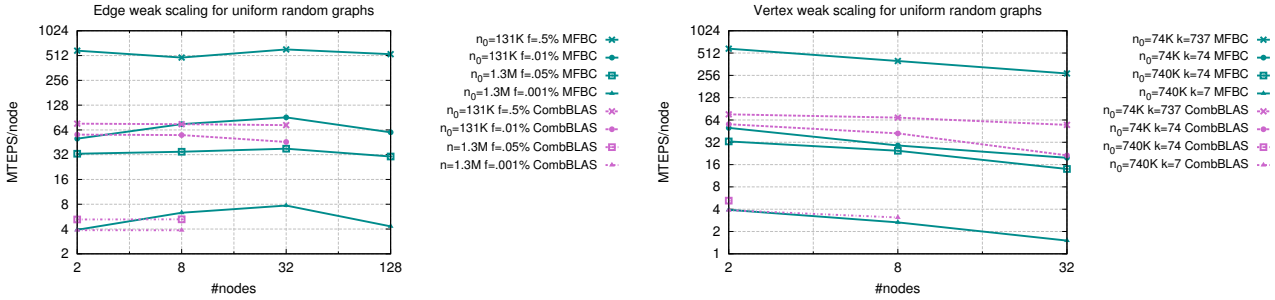
CTF-MFBC performs best for the Orkut graph, as the sparse matrix multiplications are more computation-intensive for denser graphs with low diameter (the latter implying denser frontiers). The somewhat larger diameter of LiveJournal and the significantly larger diameter of the patent citation graph take a toll on the absolute performance. However, the strong scalability is reasonable for all three of these graphs, as speed-ups of around 30X are achieved for each, when using 64X more nodes. The smallest number of nodes on which CTF-MFBC successfully executes the Friendster graph is 32, with good scalability to 128 nodes. The graph is the largest social network available in the SNAP dataset.

On the other hand, we observed relatively volatile performance for these graphs for CombBLAS, shown in Figure 1(b). In absolute



(a) MFBC on largest SNAP graphs (b) CombBLAS on largest SNAP graphs (c) Weighted and unweighted R-MAT graphs

Figure 1: Strong scaling of MFBC and CombBLAS real graphs (Table 2) and for R-MAT graphs, which have roughly 2^S vertices and average degree E. Weights are selected randomly between 1 and 100 for weighted R-MAT graphs in Figure 1(c)



(a) Constant $n^2/p = n_0^2$ and edge percentage $f = 100 \cdot m/n^2$ (b) Constant $n/p = n_0$ and vertex degree $k = m/n$

Figure 2: Weak scaling of MFBC and CombBLAS for uniform random graphs

terms the performance of CombBLAS on LiveJournal and the patent citation graph compare well to CTF-MFBC. On the other hand, we were unable to successfully execute the Friendster graph with CombBLAS. Further, CTF-MFBC is up to 7.6X faster for the Orkut graph. We don't have a good understanding of the radical change in performance of the CombBLAS benchmark (in different directions for different graphs) when transitioning from 8 to 32 nodes. On 32 nodes, CombBLAS reports a 10X improvement performance for LiveJournal by using a batch size of 8192 rather than 512, while for Orkut the corresponding improvement is only 3X.

R-MAT graphs We work with two R-MAT graphs, for both of which $\log_2(n) \approx S = 22$, while the average degree is controlled by $k \approx E \in \{8, 128\}$. Disconnected vertices are removed in CTF-MFBC and skipped for consideration as starting vertices in CombBLAS. R-MAT graphs have a low diameter, so a small number of matrix products is done in the unweighted case.

Figure 1(c) compares CTF-MFBC with CombBLAS for strong scaling on these R-MAT graphs. We omit points where we were not able to get CombBLAS to successfully execute betweenness centrality on the graph. Overall, the performance is roughly the same for the case with small average degree E=8. However, CTF-MFBC performs significantly better when E=128.

Figure 1(c) also compares CTF-MFBC performance for R-MAT graphs with edge weights randomly selected as integers in the range [1, 100] versus unweighted R-MAT. In these tests, the number of sparse matrix multiplications doubles and the frontier stays relatively dense for several steps of Algorithm 1, thus the overall performance of MFBC decreases by more than a factor of two with the inclusion of weights.

7.3 Weak Scaling

We now test CTF-MFBC's parallel scalability, while keeping m/p constant (weak scaling). We use uniform random graphs, in which all nodes have the same expected vertex degree, and every edge exists with a uniform probability. We consider "edge weak scaling" where n^2/p is kept constant and "vertex weak scaling" where n/p is kept constant. CTF-MFBC achieves good edge weak scaling, but deteriorates in efficiency for vertex weak scaling, a discrepancy justified by our theoretical analysis.

Figure 2(a) provides "edge weak scaling" results, in which the sparsity percentage of the adjacency matrix, $f = 100 \cdot m/n^2$, stays constant. The data confirms the observation that MFBC performs best for denser graphs. CTF-MFBC scales well, which is expected, since the communication cost term $O(\beta \cdot n^2/\sqrt{cp})$ grows in proportion with $\sqrt{p}$, while the amount of computation per node $O(mn/p)$ also grows in proportion with $\sqrt{p}$.

Figure 2(b) provides "vertex weak scaling" results, in which the the vertex degree k stays constant. We were unable to get CombBLAS to execute successfully on 64 nodes for the graphs with $n = 740K$ vertices. CTF-MFBC performs again better than CombBLAS when the average degree of the graph is large, but both implementations deteriorate in performance rate with increasing node count. This deterioration is predicted by our communication cost analysis, since in this weak scaling mode, the term $O(\beta \cdot n^2/\sqrt{cp})$ grows in proportion with $p^{3/2}$, while the amount of work per node $O(mn/p)$ grows in proportion with p. Therefore, unlike edge weak scaling, vertex weak scaling is not sustainable, the number of words communicated per unit of work grows with $\sqrt{p}$.

7.4 Communication Cost

We experimentally measured the communication complexity incurred by CTF-MFBC and CombBLAS. Communication in both codes is dominated by collective communication routines. We profiled the time spent in communication (excluding time to synchronize) as well as used the parameters passed to MPI routines to build an analytical model. To get the critical path costs, we follow the communication pattern: for each collective over a set of processors, we maximize the critical path costs incurred by those processors so far. Broadcast and reduce of a message of size n over p processors have the cost $2n \cdot \beta + 2\log_2(p) \cdot \alpha$ (twice that of scatter and allgather). At the end of execution, we consider the maximum over all processors for each cost, thus obtaining the greatest amount of data communicated along any dependent sequence of collectives, as well as the greatest amount of messages communicated along any (possibly different) dependent sequence of collectives.

graph	code	W (GB)	S (#msgs)	comm (sec)	total (sec)
Orkut	CombBLAS	19.71	546.9K	48.81	233.7
Orkut	CTF-MFBC	7.010	184.6K	46.95	111.6
LiveJournal	CombBLAS	10.21	190.6K	54.03	238.5
LiveJournal	CTF-MFBC	8.794	94.69K	54.00	100.2
Patents	CombBLAS	1.026	202.8K	4.084	6.422
Patents	CTF-MFBC	3.900	35.32K	24.05	60.53

Table 3: Critical path times and costs collected on 4096 cores of Blue Waters, all for a single batch of 512 starting vertices. Total time includes overhead of additional profiling barriers.

Table 3 shows that CTF-MFBC uses fewer messages and performs less collective communication than CombBLAS in some cases. In various other cases, CombBLAS performed less communication. For the patent citations graph, CombBLAS performs significantly faster than CTF. The blocked layout and choice of starting vertices likely permits CombBLAS to exploit locality for this directed graph. Further, the back-propagation stage for CTF-MFBC (which is dynamically computed, rather than stored from BFS) takes considerably longer (performing more work in the directed case).

Overall, we can conclude that either the CombBLAS or CTF-MFBC implementation may incur more communication, depending on the type of graph. CTF-MFBC seems to require less for denser graphs (e.g. Orkut and uniform random). Persistence of layout and more accurate performance models would further reduce communication costs.

8 CONCLUSION

Our new maximal frontier algorithm for betweenness centrality achieves good parallel scaling due to its low theoretical communication complexity and a robust implementation of its primitive operations. The algebraic formalism we use for propagating information through graphs enables intuitive expression of frontiers and edge relaxations, making it extensible to other graph problems such as maximum flow. We expect that the approach of selecting frontiers to maximize overall progress also leads to good parallel algorithms for other graph computations.

By implementing MFBC on top of Cyclops Tensor Framework (CTF), we have introduced the first application case-study of CTF for a non-numerical problem. MFBC with CTF shows substantial improvements in performance over CombBLAS for some graphs, while additionally being general to weighted graphs. Automatic parallelism for sparse tensor contractions with arbitrary algebraic structures is useful in many other application contexts. The communication-efficiency achieved by sparse matrix multiplication routines in CombBLAS and CTF has a promising potential for changing the way massively-parallel graph computations are done.

9 ACKNOWLEDGEMENTS

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications. We thank Hussein Harake, Colin McMurtrie, and the whole CSCS team granting access to the Piz Dora machine and for their excellent technical support. Maciej Besta was supported by Google European Doctoral Fellowship and Edgar Solomonik was supported by an ETH Zurich Postdoctoral Fellowship.

REFERENCES

[1] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. 1995. A three-dimensional approach to parallel matrix multiplication. *IBM J. Res. Dev.* 39 (September 1995), 575–582. Issue 5.

[2] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. 1990. Communication complexity of PRAMs. *Theoretical Computer Science* 71, 1 (1990), 3 – 28.

[3] Ariful Azad, Grey Ballard, Aydin Buluc, James Demmel, Laura Grigori, Oded Schwartz, Sivan Toledo, and Samuel Williams. 2015. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *arXiv preprint arXiv:1510.00844* (2015).

[4] David A Bader, Shiva Kintali, Kamesh Madduri, and Milena Mihail. 2007. Approximating betweenness centrality. In *Algorithms and Models for the Web-Graph*. Springer, 124–137.

[5] David A. Bader and Kamesh Madduri. 2006. Parallel Algorithms for Evaluating Centrality Indices in Real-world Networks. In *Proceedings of the 2006 International Conference on Parallel Processing (ICPP '06)*. IEEE Computer Society, Washington, DC, USA, 539–550. https://doi.org/10.1109/ICPP.2006.57

[6] Grey Ballard, Aydin Buluc, James Demmel, Laura Grigori, Benjamin Lipshitz, Oded Schwartz, and Sivan Toledo. 2013. Communication Optimal Parallel Multiplication of Sparse Random Matrices. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '13)*. ACM, New York, NY, USA, 222–231. https://doi.org/10.1145/2486159.2486196

[7] Richard Bellman. 1956. *On a routing problem.* Technical Report. DTIC Document.

[8] Massimo Bernaschi, Giancarlo Carbone, and Flavio Vella. 2016. Scalable Betweenness Centrality on Multi-GPU Systems. In *Proceedings of the ACM International Conference on Computing Frontiers (CF '16)*. ACM, New York, NY, USA, 29–36. https://doi.org/10.1145/2903150.2903153

[9] Jarle Berntsen. 1989. Communication efficient matrix multiplication on hypercubes. *Parallel Comput.* 12, 3 (1989), 335–342.

[10] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology* 25, 2 (2001), 163–177.

[11] Aydın Buluç and John R Gilbert. 2011. The Combinatorial BLAS: Design, implementation, and applications. *International Journal of High Performance Computing Applications* (2011), 1094342011403516.

[12] Aydin Buluç and John R. Gilbert. 2012. Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments. *SIAM Journal on Scientific Computing* 34, 4 (2012), C170–C191. https://doi.org/10.1137/110848244 arXiv:http://dx.doi.org/10.1137/110848244

[13] Lynn Elliot Cannon. 1969. *A cellular computer to implement the Kalman filter algorithm.* Ph.D. Dissertation. Bozeman, MT, USA.

[14] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining.. In *SDM*, Vol. 4. SIAM, 442–446.

[15] Guojing Cong and Konstantin Makarychev. 2012. Optimizing large-scale graph analysis on multithreaded, multicore platforms. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 414–425.

[16] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. *Introduction to Algorithms* (2nd ed.). McGraw-Hill Higher Education.

[17] Eliezer Dekel, David Nassimi, and Sartaj Sahni. 1981. Parallel Matrix and Graph Algorithms. *SIAM J. Comput.* 10, 4 (1981), 657–675.

[18] Yangdong Steve Deng, Bo David Wang, and Shuai Mu. 2009. Taming irregular EDA applications on GPUs. In *Proceedings of the 2009 International Conference*

on Computer-Aided Design. ACM, 539–546.

[19] Nick Edmonds, Torsten Hoefler, and Andrew Lumsdaine. 2010. A space-efficient parallel algorithm for computing betweenness centrality in distributed memory. In *2010 International Conference on High Performance Computing*. IEEE, 1–10.

[20] Rui Fan, Ke Xu, and Jichang Zhao. 2017. A GPU-Based Solution to Fast Calculation of Betweenness Centrality on Large Weighted Networks. *arXiv preprint arXiv:1701.05975* (2017).

[21] Lester Randolph Ford. 1956. Network Flow Theory. (1956).

[22] Edgar N Gilbert. 1959. Random graphs. *The Annals of Mathematical Statistics* 30, 4 (1959), 1141–1144.

[23] Oded Green and David A Bader. 2013. Faster betweenness centrality based on data structure experimentation. *Procedia Computer Science* 18 (2013), 399–408.

[24] Shuangshuang Jin, Zhenyu Huang, Yousu Chen, Daniel Chavarría-Miranda, John Feo, and Pak Chung Wong. 2010. A novel application of parallel betweenness centrality to power grid contingency analysis. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 1–7.

[25] S. Lennart Johnsson. 1993. Minimizing the communication time for matrix multiplication on multiprocessors. *Parallel Comput.* 19 (November 1993), 1235–1257. Issue 11.

[26] Jeremy Kepner and John Gilbert. 2011. *Graph algorithms in the language of linear algebra*. Vol. 22. SIAM.

[27] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. 2008. Technology-Driven, Highly-Scalable Dragonfly Topology. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE Computer Society, Washington, DC, USA, 77–88. https://doi.org/10.1109/ISCA.2008.19

[28] S. Kintali. 2008. Betweenness Centrality : Algorithms and Lower Bounds. *ArXiv e-prints* (Sept. 2008). arXiv:cs.DS/0809.1906

[29] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic parallelism requires abstractions. In *ACM SIGPLAN Conf. on Prog. Lang. Des. and Impl. (PLDI '07)*. ACM, New York, NY, USA, 211–222. https://doi.org/10.1145/1250734.1250759

[30] Chuck L. Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. 1979. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)* 5, 3 (1979), 308–323.

[31] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (June 2014).

[32] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan W. Berry. 2007. Challenges in Parallel Graph Processing. *Par. Proc. Let.* 17, 1 (2007), 5–20.

[33] Kamesh Madduri, David Ediger, Karl Jiang, David Bader, Daniel Chavarria-Miranda, et al. 2009. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 1–8.

[34] W. F. McColl and A. Tiskin. 1999. Memory-Efficient Matrix Multiplication in the BSP Model. *Algorithmica* 24 (1999), 287–297. Issue 3.

[35] Adam McLaughlin and David A Bader. 2014. Scalable and high performance betweenness centrality on the GPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 572–583.

[36] Dimitrios Prountzos and Keshav Pingali. 2013. Betweenness centrality: algorithms and implementations. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 35–46.

[37] Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit V Çatalyürek. 2013. Betweenness centrality on GPUs and heterogeneous architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. ACM, 76–85.

[38] Zhiao Shi and Bing Zhang. 2011. Fast network centrality analysis using GPUs. *BMC bioinformatics* 12, 1 (2011), 1.

[39] Edgar Solomonik, Erin Carson, Nicholas Knight, and James Demmel. 2014. Tradeoffs between synchronization, communication, and computation in parallel linear algebra computations. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*. ACM, 307–318.

[40] Edgar Solomonik and James Demmel. 2011. Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms. In *Euro-Par 2011 Parallel Processing*, Emmanuel Jeannot, Raymond Namyst, and Jean Roman (Eds.). Lecture Notes in Computer Science, Vol. 6853. Springer Berlin Heidelberg, 90–109. https://doi.org/10.1007/978-3-642-23397-5_10

[41] E. Solomonik and T. Hoefler. 2015. Sparse Tensor Algebra as a Parallel Programming Model. *ArXiv e-prints* (Nov. 2015). arXiv:cs.MS/1512.00066

[42] Guangming Tan, Vugranam C Sreedhar, and Guang R Gao. 2011. Analysis and performance results of computing betweenness centrality on IBM Cyclops64. *The Journal of Supercomputing* 56, 1 (2011), 1–24.

[43] Guangming Tan, Dengbiao Tu, and Ninghui Sun. 2009. A parallel algorithm for computing betweenness centrality. In *2009 International Conference on Parallel Processing*. IEEE, 340–347.

[44] Alexander Tiskin. 2001. All-Pairs Shortest Paths Computation in the BSP Model. In *Automata, Languages and Programming*, Fernando Orejas, Paul Spirakis, and Jan van Leeuwen (Eds.). Lecture Notes in Computer Science, Vol. 2076. Springer Berlin / Heidelberg, 178–189.

[45] R. A. Van De Geijn and J. Watts. 1997. SUMMA: Scalable Universal Matrix Multiplication Algorithm. *Concurrency: Practice and Experience* 9, 4 (1997), 255–274.

[46] Jing Yang and Yingwu Chen. 2011. Fast computing betweenness centrality with virtual nodes on large sparse networks. *PloS one* 6, 7 (2011), e22557.

[47] Qiaofeng Yang and Stefano Lonardi. 2005. A parallel algorithm for clustering protein-protein interaction networks. In *2005 IEEE Computational Systems Bioinformatics Conference-Workshops (CSBW'05)*. IEEE, 174–177.

10 REPRODUCIBILITY APPENDIX

We describe our benchmarking methodology in Section 7. Here, we provide some more technical details to allow reproduction of our results on the Blue Waters machine or a similar architecture. We make our anonymized code for betweenness centrality available in the repository https://github.com/solomonik/graph_centrality. The repository also includes a modified CombBLAS benchmark that generates uniform random graphs.

Compilation

We built CTF using the Intel compiler module v5.2.82 with an additional gcc v5.3.0 module loaded. We built CombBLAS using purely the gcc v5.2.82 module. These modules were the latest available at the start of April 2017.

We built CTF with flags '-fast' and -no-ipo' as well as weith profiling output on. We built CombBLAS with the default optimization flags '-O2'. No changes were made to the contents of either software internally to the aforementioned versions (CTF v1.4.1, ComBBLAS v1.5.0).

Execution

We provide our submission job generation scripts below. In some cases, we rerun parts of scripts if they did not complete in the expected time.

Real-World graphs

```
#!/bin/bash

snp=$1
nodes=$((2*$snp*$snp))
p=$((32*$nodes))

cat > real.script$nodes.sh <<EOF
#!/bin/bash
#PBS -l nodes=$nodes:ppn=32:xe
#PBS -l walltime=03:00:00
#PBS -N ctf_btwn_central
#PBS -e \$PBS_JOBID.err
#PBS -o \$PBS_JOBID.out
#PBS -W umask=0027

cd \$PBS_O_WORKDIR

for bsize in 128 512 2048 8192
do
  aprun -n $p ./betwcent ../graphs/patents
      1 \$bsize >> patents.p$p.\$PBS_JOBID
  aprun -n $p ./betwcent ../graphs/livejournal
      1 \$bsize >> livejournal.p$p.\$PBS_JOBID
  aprun -n $p ./betwcent ../graphs/orkut
      1 \$bsize >> orkut.p$p.\$PBS_JOBID
 aprun -n $p ./betwcent ../graphs/friendster
      1 \$bsize >> friendster.p$p.\$PBS_JOBID

  aprun -n $p ./test_btwn_central -prep 1 -test 0
      -bsize \$bsize -mwht 1 -nbatches 1 -n 6009554
      -f ../graphs/patents/patents.txt
      >> patents.p$p.\$PBS_JOBID
  aprun -n $p ./test_btwn_central -prep 1 -test 0
      -bsize \$bsize -mwht 1 -nbatches 1 -n 4847571
      -f ../graphs/livejournal/livejournal.txt
      >> livejournal.p$p.\$PBS_JOBID
  aprun -n $p ./test_btwn_central -prep 1 -test 0
      -bsize \$bsize -mwht 1 -nbatches 1 -n 3072441
      -f ../graphs/orkut/orkut.txt
      >> orkut.p$p.\$PBS_JOBID
  aprun -n $p ./test_btwn_central -prep 1 -test 0
      -bsize \$bsize -mwht 1 -nbatches 1 -n 65608366
      -f ../graphs/friendster/friendster.txt
      >> friendster.p$p.\$PBS_JOBID
done
EOF
```

R-MAT graphs

```
#!/bin/bash

snp=$1
nodes=$((2*$snp*$snp))
p=$((32*$nodes))
sqrt_p=$((8*$snp))
cat > rmat.script$nodes.sh <<EOF
#!/bin/bash
#PBS -l nodes=$nodes:ppn=32:xe
#PBS -l walltime=02:00:00
#PBS -N ctf_btwn_central
#PBS -e \$PBS_JOBID.err
#PBS -o \$PBS_JOBID.out
#PBS -W umask=0027

cd \$PBS_O_WORKDIR

for bsize in 128 512 2048 8192
do
  aprun -n $p ./betwcent ../graphs/rmat_e8  1 \$bsize
      >> rmat.S22.E8.p$p.\$PBS_JOBID
  aprun -n $p ./test_btwn_central -prep 1 -test 0
      -bsize \$bsize -mwht 1 -nbatches 1 -S 22 -E 8
      >> rmat.S22.E8.p$p.\$PBS_JOBID
  aprun -n $p ./test_btwn_central -prep 1 -test 0
      -bsize \$bsize -mwht 1 -nbatches 1 -S 22 -E 8
      -adapt 0 >> rmat.S22.E8.p$p.\$PBS_JOBID
  aprun -n $p ./test_btwn_central -prep 1 -test 0
      -bsize \$bsize -mwht 1 -nbatches 1 -S 22 -E 8
      -adapt 0 -sp_C 0 >> rmat.S22.E8.p$p.\$PBS_JOBID
  aprun -n $p ./test_btwn_central -prep 1 -test 0
      -bsize \$bsize -mwht 1 -nbatches 1 -S 22 -E 8
      -adapt 0 -sp_C 0 -sp_B 0
      >> rmat.S22.E8.p$p.\$PBS_JOBID
  aprun -n $p ./test_btwn_central -prep 1 -test 0
      -bsize \$bsize -mwht 100 -nbatches 1 -S 22 -E 8
      >> rmat.S22.E8.p$p.\$PBS_JOBID

  aprun -n $p ./betwcent ../graphs/rmat_e128  1
      \$bsize >> rmat.S22.E128.p$p.\$PBS_JOBID
  aprun -n $p ./test_btwn_central -prep 1 -test 0
      -bsize \$bsize -mwht 1 -nbatches 1 -S 22 -E 128
```

```
      >> rmat.S22.E128.p$p.\$PBS_JOBID
  aprun -n $p ./test_btwn_central -prep 1 -test 0
      -bsize \$bsize -mwht 1 -nbatches 1 -S 22 -E 128
      -adapt 0 >> rmat.S22.E128.p$p.\$PBS_JOBID
  aprun -n $p ./test_btwn_central -prep 1 -test 0
      -bsize \$bsize -mwht 1 -nbatches 1 -S 22 -E 128
     -adapt 0 -sp_C 0 >> rmat.S22.E128.p$p.\$PBS_JOBID
  aprun -n $p ./test_btwn_central -prep 1 -test 0
      -bsize \$bsize -mwht 1 -nbatches 1 -S 22 -E 128
      -adapt 0 -sp_C 0 -sp_B 0
      >> rmat.S22.E128.p$p.\$PBS_JOBID
  aprun -n $p ./test_btwn_central -prep 1 -test 0
     -bsize \$bsize -mwht 100 -nbatches 1 -S 22 -E 128
      >> rmat.S22.E128.p$p.\$PBS_JOBID
done
EOF
```

Random-uniform graphs

```
#!/bin/bash

snp=$1
nodes=$((2*$snp*$snp))
p=$((32*$nodes))
sqrt_p=$((8*$snp))
n0=131072
n1=1310720
n0_escl=$(($n0*$snp))
n0_vscl=$(($n0*$snp*$snp))
n1_escl=$(($n1*$snp))
n1_vscl=$(($n1*$snp*$snp))
f_nnz00=.005
f_nnz01=.0001
f_nnz10=.0005
f_nnz11=.00001
f_vscl_nnz00=$(echo "$f_nnz00/($snp*$snp)" | bc -l)
f_vscl_nnz01=$(echo "$f_nnz01/($snp*$snp)" | bc -l)
f_vscl_nnz10=$(echo "$f_nnz10/($snp*$snp)" | bc -l)
f_vscl_nnz11=$(echo "$f_nnz11/($snp*$snp)" | bc -l)

cat > unif.script$nodes.sh <<EOF
#!/bin/bash
#PBS -l nodes=$nodes:ppn=32:xe
#PBS -l walltime=02:00:00
#PBS -N ctf_btwn_central
#PBS -e \$PBS_JOBID.err
#PBS -o \$PBS_JOBID.out
#PBS -W umask=0027

cd \$PBS_O_WORKDIR

for bsize in 128 512 2048 8192
do
  aprun -n $p ./betwcent_unif $n0_escl $f_nnz00 1 \$bsize
    >> unif.n$n0_escl.sp$f_nnz00.p$p.\$PBS_JOBID
  aprun -n $p ./betwcent_unif $n0_escl $f_nnz01 1 \$bsize
    >> unif.n$n0_escl.sp$f_nnz01.p$p.\$PBS_JOBID
  aprun -n $p ./betwcent_unif $n1_escl $f_nnz10 1 \$bsize
    >> unif.n$n1_escl.sp$f_nnz10.p$p.\$PBS_JOBID
  aprun -n $p ./betwcent_unif $n1_escl $f_nnz11 1 \$bsize
    >> unif.n$n1_escl.sp$f_nnz11.p$p.\$PBS_JOBID

  aprun -n $p ./test_btwn_central -prep 1 -test 0 -bsize \$bsize
    -mwht 1 -nbatches 1 -n $n0_escl -sp $f_nnz00
    >> unif.n$n0_escl.sp$f_nnz00.p$p.\$PBS_JOBID
  aprun -n $p ./test_btwn_central -prep 1 -test 0 -bsize \$bsize
    -mwht 1 -nbatches 1 -n $n0_escl -sp $f_nnz01
    >> unif.n$n0_escl.sp$f_nnz01.p$p.\$PBS_JOBID
  aprun -n $p ./test_btwn_central -prep 1 -test 0 -bsize \$bsize
    -mwht 1 -nbatches 1 -n $n1_escl -sp $f_nnz10
    >> unif.n$n1_escl.sp$f_nnz10.p$p.\$PBS_JOBID
  aprun -n $p ./test_btwn_central -prep 1 -test 0 -bsize \$bsize
    -mwht 1 -nbatches 1 -n $n1_escl -sp $f_nnz11
    >> unif.n$n1_escl.sp$f_nnz11.p$p.\$PBS_JOBID

  aprun -n $p ./betwcent_unif $n0_vscl $f_vscl_nnz00 1
    \$bsize >> unif.n$n0_vscl.sp0.p$p.\$PBS_JOBID
  aprun -n $p ./betwcent_unif $n0_vscl $f_vscl_nnz01 1
    \$bsize >> unif.n$n0_vscl.sp1.p$p.\$PBS_JOBID
  aprun -n $p ./betwcent_unif $n1_vscl $f_vscl_nnz10 1
    \$bsize >> unif.n$n1_vscl.sp0.p$p.\$PBS_JOBID
  aprun -n $p ./betwcent_unif $n1_vscl $f_vscl_nnz11 1
    \$bsize >> unif.n$n1_vscl.sp1.p$p.\$PBS_JOBID

  aprun -n $p ./test_btwn_central -prep 1 -test 0 -bsize \$bsize
    -mwht 1 -nbatches 1 -n $n0_vscl
   -sp $f_vscl_nnz00 >> unif.n$n0_vscl.sp0.p$p.\$PBS_JOBID
  aprun -n $p ./test_btwn_central -prep 1 -test 0 -bsize \$bsize
    -mwht 1 -nbatches 1 -n $n0_vscl -sp $f_vscl_nnz01
    >> unif.n$n0_vscl.sp1.p$p.\$PBS_JOBID
  aprun -n $p ./test_btwn_central -prep 1 -test 0 -bsize \$bsize
    -mwht 1 -nbatches 1 -n $n1_vscl -sp $f_vscl_nnz10
    >> unif.n$n1_vscl.sp0.p$p.\$PBS_JOBID
  aprun -n $p ./test_btwn_central -prep 1 -test 0 -bsize \$bsize
    -mwht 1 -nbatches 1 -n $n1_vscl -sp $f_vscl_nnz11
    >> unif.n$n1_vscl.sp1.p$p.\$PBS_JOBID
done
EOF
```