Betweenness Centrality: Algorithms and Implementations [*]

Dimitrios Prountzos   Keshav Pingali
The University of Texas at Austin, Texas, USA
dprountz@cs.utexas.edu, pingali@cs.utexas.edu

Abstract

Betweenness centrality is an important metric in the study of social networks, and several algorithms for computing this metric exist in the literature. This paper makes three contributions. First, we show that the problem of computing betweenness centrality can be formulated abstractly in terms of a small set of *operators* that update the graph. Second, we show that existing parallel algorithms for computing betweenness centrality can be viewed as implementations of different schedules for these operators, permitting all these algorithms to be formulated in a single framework. Third, we derive a new asynchronous parallel algorithm for betweenness centrality that (i) works seamlessly for both weighted and unweighted graphs, (ii) can be applied to large graphs, and (iii) is able to extract large amounts of parallelism. We implemented this algorithm and compared it against a number of publicly available implementations of previous algorithms on two different multicore architectures. Our results show that the new algorithm is the best performing one in most cases, particularly for large graphs and large thread counts, and is always competitive against other algorithms.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming; G.2.2 [*Discrete Mathematics*]: Graph Theory—Graph algorithms

General Terms Algorithms, Performance

Keywords Concurrency, Parallelism, Amorphous Data-parallelism, Irregular Programs, Optimistic Parallelization, Betweenness Centrality.

1. Betweenness Centrality

Centrality metrics are essential in understanding network structure, since they capture the relative importance of individual nodes in the overall network. In this paper, we examine Betweenness Centrality (BC) [20], a commonly used metric that is based on shortest path computation. If $G = (V, E)$ is a graph and s, t are a fixed pair of graph nodes, the betweenness score of a node v for this node pair is the fraction of shortest paths between s and t that include v. The betweenness centrality of v is the sum of its betweenness scores for all possible pairs of s and t in the graph. More formally, let σ_{st} be the number of shortest paths between s and t, and let $\sigma_{st}(v)$ be the number of those shortest paths that pass through v. The betweenness centrality of node v is defined as: $BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$.

Applications of BC include study of sexual networks and AIDS [28], lethality in biological networks [17, 25], identifying key actors in terrorist networks [15, 27], organizational behavior [11], contingency analysis for power grid component failures [26], and analysis of transportational networks [23]. BC is also used as a heuristic in other algorithms; for example [22] proposes an algorithm for community detection and clustering in large networks, based on the BC of the network edges.

1.1 Brandes' Algorithm: a Basis for Parallel BC Algorithms

An efficient sequential algorithm for computing BC was proposed by Brandes [9], and it has been the basis for many parallelization approaches [5, 16, 19, 29, 36]. Below, we outline the main ideas behind Brandes' algorithm. We define the *dependency* of a source vertex s on a vertex v as: $\delta_s(v) = \sum_{t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$. The betweenness centrality of a vertex v is then expressed using Eq. (1). The key insight is that $\delta_s(v)$ satisfies the recurrence (2), where $pred(s, w)$ is a list of immediate predecessors of w in the shortest paths from s to w.

$$BC(v) = \sum_{s \neq v \in V} \delta_s(v) \tag{1}$$

$$\delta_s(v) = \sum_{w:\ v \in pred(s,w)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_s(w)) \tag{2}$$

Brandes' algorithm uses this insight and it works as follows. Each $s \in V$ is considered as a source of shortest-paths and the contribution of s to $BC(v)$, for all $v \neq s$ is computed in two phases. In the first phase, a shortest-path computation is performed from s, that computes $pred(s, v)$ and σ_{sv} for all nodes. The predecessor lists induce a DAG D over the graph G. In a second phase, D is traversed backward (in non-increasing distance order) and for each $v \in V$, $\delta_s(v)$ is computed based on (2), and the contribution to $BC(v)$ is computed based on (1). The process is described in Fig. 1. Between processing successive sources, node and edge attributes are reset.

1.2 Understanding Parallelism in BC

This algorithm has parallelism at multiple levels. First, we can process multiple source nodes in parallel (loop in line 3 in Fig. 1). In this coarse-grained parallelization strategy, each thread picks an arbitrary graph node s and computes its contribution to the betweenness values of other nodes. Each of these computations is independent, and the updates on each $BC(v)$ form a simple reduction. This parallelization strategy is simple and effective, but each outer loop iteration that is performed in parallel requires its own storage, so the space overhead of this scheme can be substantial. Therefore, it is used only for relatively small graphs. We will refer to approaches

[*] This work was supported by NSF grants 0833162, 1062335, 1111766, 1218568, and 1216701 as well as grants and equipment donations from NVIDIA, IBM, Intel and Qualcomm Corporations.

```
1  Graph G = /* read input graph */
2  Worklist wl = $\{v : v \in G.nodes\}$
3  foreach $s$ : Node $\in$ wl {
4    forall $v$ : Node $\in G$: // Compute shortest−path DAG D
5      compute $\sigma_{sv}$
6      compute $pred(s, v)$
7    forall $v$ : Node $\in D$: // Traverse DAG D backward
8      compute $\delta_s(v)$
9      $BC(v)$ += $\delta_s(v)$
10   forall $(u, v)$ : Edge $\in G$: // Reset graph attributes
11     reset (u,v)
12 }
```

Figure 1. Pseudocode for Brandes' algorithm.

using this strategy as *outer-level* schemes. Alternatively, we can expose parallelism by focusing on processing a single source node and performing each of the computation steps in parallel (loops in lines $4, 7, 10$ in Fig. 1). This fine-grained, *inner level* approach is more space-efficient since we only need to maintain a single graph instance, but poses a more challenging goal for parallelization due to non-trivial data dependencies. Finally one can combine the two techniques by processing several source nodes in parallel and performing the per-node computations in parallel.

1.3 Previous Work

Examples of the outer-level approach are [12, 26, 41]. Parallel performance is excellent, as expected, but the size of the input graph is very small or a big distributed cluster is used [12]. Bader et al. in [5] were the first to present an approach that targets both outer-level and inner-level parallelism. This work focuses on unweighted graphs, where the shortest path exploration can be performed by a breadth-first-search (BFS) exploration. Both of the main phases of the algorithm are performed in a level-parallel manner. Within level i all nodes are processed in parallel but only edges between nodes in levels i and $i + 1$ are allowed to be processed. Similarly, in the backward DAG traversal, only nodes between levels i and $i - 1$ are processed in parallel. The strong ordering between levels is achieved by using barriers. Subsequent work by Madduri et al. [29], improves the algorithm to use successors instead of predecessors in the computation of the DAG D, which produces a more efficient, locality-friendly algorithm. [16] targets outer-level parallelism and also performs prefetching and appropriate re-layout of the graph nodes to improve locality. [39] presents a variation of [5] where the graph is logically partitioned among processors, locking is coarsened to a lock per partition, and the predecessor lists are distributed across partitions; [38] extends this work with architecture specific optimizations for the IBM Cyclops64 processor. Similarly, in [36] a GPU level-synchronous parallelization is presented, where graph edges are partitioned among the threads.

Edmonds et al. [19] present an approach that targets fine-grained parallelism and focuses on a distributed memory environment. This work deals with both weighted and unweighted graphs. In the case of weighted graphs, the level-parallel BFS approach is not applicable. Their solution breaks up the DAG construction phase into a number of sub-phases, separated again by barriers. Initially a label-correcting single-source shortest-path (SSSP) algorithm is employed [30, 33] to compute the shortest path distances and predecessor lists. Then, using the predecessor lists the node successors are computed. Finally, a third sub-phase computes σ_{sv} in a level-parallel BFS style, using the node successors. The backward traversal of the DAG is performed without using barriers by using the predecessor lists. [40] presents a serial adaption of Brandes that deals with weighted graphs by adding virtual nodes to turn them into unweighted graphs, where a BFS exploration can be performed. These algorithms compute the exact value of BC. To reduce the computational cost, a number of approximation algorithms have been proposed [4, 10, 21]. For example, instead of computing the contribution of all source nodes $s \neq v$ to $BC(v)$ in Eq. (1), we can compute the contributions of a subset of source nodes.

1.4 Goals and Contributions

This paper makes three contributions.

- We show that the problem of computing BC can be formulated abstractly in terms of the *operator formulation* of algorithms [34].
- We show that existing parallel BC algorithms can be viewed as implementations of different schedules for applying the operators to the graph, permitting all these algorithms to be formulated in a single framework.
- The full set of our operators can correctly compute BC *under any arbitrary schedule* and can therefore be the basis for a new class of algorithms that can potentially mine more parallelism. This is especially true for the case of weighted graphs, where a purely level-synchronous approach cannot be used. Using these operators, we derive a new parallel algorithm by carefully controlling and optimizing the scheduling of operators. It is space-efficient because it targets fine-grained parallelism. It is able to expose a lot of parallelism because it breaks away from the level parallel mode of execution. It deals with both weighted and unweighted graphs, and, as we show experimentally, has good scalability. Our current implementation targets multicore systems; it is straightforward to adopt it to a distributed setting.

The rest of the paper is organized as follows. § 2 presents the operator formulation of BC and describes how existing algorithms for BC can be viewed as implementations of different schedules for applying the operators to the graph. § 3 describes how to derive and optimize a new asynchronous algorithm for BC by appropriately controlling operator scheduling. § 4 presents our experimental evaluation on two multicore architectures using inputs from multiple graph classes. § 5 concludes the paper.

2. A Framework for Expressing BC Algorithms

In this section, we formulate the computation of betweenness centrality in terms of the *operator formulation* of algorithms [34]. Operators act on graph nodes and edges and update their attributes.We describe a generic algorithm that computes BC by repeatedly applying operators in an unspecified order to the graph until a fixpoint is reached (that is, when no new operator applications can happen). We also show that existing algorithms for BC can be viewed as particular schedules for applying these operators.

To introduce the notion of operators, we consider the simpler problem of computing the breadth-first-search level of nodes in a directed graph. Each node u has a field $l(u)$ (for level), initialized to 0 for *Root* , and to ∞ for all other nodes. When the algorithm terminates, the level of a node will be equal to the length of the shortest path from *Root* to that node. The algorithm discovers paths from *Root* incrementally, so during the execution of the algorithm, the level of a node v is equal to the length of the shortest path to v that has been discovered so far.

Fig. 2 shows the operator formulation for BFS. Like all the operators we introduce in this paper, this operator is applied to a single edge of the graph and to the two nodes that are its end-points. An operator has a left hand side that specifies the precondition (predicate) under which it can be applied; an edge that satisfies this precondition is called an *active edge*. An active edge can be updated as shown in the right-hand side of the operator.

Initial state: $\forall u \neq \mathit{Root}\colon\ l(u) = \infty, l(\mathit{Root}) = 0$

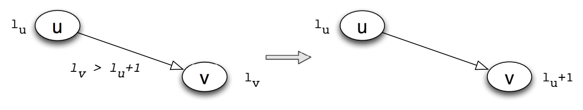


Figure 2. BFS expressed using a single operator for computing shortest paths.

The operator of Figure 2 can be described in words as follows: an edge (u, v) is active if $l(v) > l(u) + 1$; such an edge can be updated by setting $l(v)$ to $l(u) + 1$. If there are several active edges in a graph, an implementation is allowed to update them concurrently, provided that active edge attributes are updated atomically. It can be shown that as long as the scheduling of active edges is fair (that is, the selection of an active edge is not postponed indefinitely), (i) the computation will terminate within some finite number of steps, and (ii) upon termination, for each node u, $l(u)$ will be its BFS level.

2.1 Operators for BC

We now show how to express BC using a small set of operators. To keep the presentation simple, we focus first on unweighted graphs, and then extend our approach to the case of weighted graphs. Our solution operates in two phases. In the first phase, it builds the BFS DAG. As in the BFS computation described above, the first phase discovers and records paths from *Root* to other nodes incrementally. The second phase performs a bottom-up walk of the DAG to compute betweenness centrality.

For each node u we maintain a number of attributes:

- the shortest path distance ($l(u)$) of u,
- the number of shortest paths ($\sigma(u)$) of length $l(u)$ from *Root* to u,
- a list of nodes ($\mathit{preds}(u)$), each of which is the predecessor of u on a shortest path from *Root* to u, and
- a list of nodes ($\mathit{succs}(u)$), each of which is a successor of u on a shortest path from *Root*. Our implementation actually maintains only the number of the successors of a node and not the full list; for expository purposes we describe the operators using the successor list attribute.

Additionally, we associate with each edge (u, v) a level ($l(uv)$) and a path-count ($\sigma(uv)$) attribute; these are used for book-keeping during the algorithm execution, as explained below.

2.1.1 Operators for the DAG Construction Phase

The goal of the first phase is to construct the shortest-path DAG D and also compute the path-count $\sigma(u)$ for each node u. There are four operators, shown in Figure 3. Below, we discuss each in detail.

Shortest Path (*SP*): This is the same as the BFS operator except that it also resets $\sigma(v)$, $\mathit{preds}(v)$ and $\mathit{succs}(v)$ to their default values. This is the only operator that modifies levels of nodes. It is enabled when the following *guard* predicate $g_{SP}(uv)$ is true:

$$g_{SP}(uv) := l(v) > l(u) + 1$$

First Update (*FU*): This operator is applied to an edge connecting nodes at successive levels. It updates $\sigma(v)$ with the current value of $\sigma(u)$ and it also updates the predecessor and successor lists of v and u. The operator is enabled when g_{FU} holds:

$$g_{FU}(uv) := l(v) = l(u) + 1 \wedge l(uv) \neq l(u)$$

The second constraint ensures that the operator is applied only once, since after the operator application, $l(uv) = l(u)$ as long as $l(u)$ is stable. There may be several incoming edges to node v at level $l + 1$ from nodes at level l, and this operator will be applied once for each such edge. These applications will be preceded by an application of the SP operator that brings node v to level $l + 1$.

Update Sigma (*US*): This operator is applied on an edge connecting nodes at successive levels, and it propagates changes to the path-count ($\sigma(u)$) of u to v. The previous update of $\sigma(v)$ from $\sigma(u)$ is stored in $\sigma(uv)$, which is used to compute the correct incremental update. The operator is enabled when g_{US} holds:

$$g_{US}(uv) := l(u) = l(uv) \wedge l(v) = l(u) + 1 \wedge \sigma(uv) \neq \sigma(u)$$

Correct Node (*CN*): This operator corrects the successor list of u in case its neighbor v moves to a lower level after having received some updates from u. Note that it is unnecessary to remove u from $\mathit{preds}(v)$ since $\mathit{preds}(v)$ would have been set to $\emptyset$ when v moved to a lower level as a result of applying *SP*. The operator is enabled when g_{CN} holds:

$$g_{CN}(uv) := l(u) \geq l(v) \wedge l(uv) = l(u) \wedge l(u) \neq \infty$$

Example 1 (Sample execution of the first phase)**.** *In Fig. 4 we show a sample execution of the operators for the DAG construction phase of the algorithm. The Root node is s. Then, nodes a, b, c are at distance 1 and node d is at distance 2. Initially, all nodes other than s have distance ∞. First, we explore all nodes across the path (s, a, c, d). This results in a sequence of SP and FU applications that set $l(a) = 1, l(c) = 2, l(d) = 3$, update all path-counts to 1, and set successors and predecessors accordingly. Subsequently, we explore the path (s, b, c, d) and perform a similar sequence of operator applications. Note that, when we process (c, d), we apply US(cd) to correct c's contribution to d's path-count. When we explore path (s, c, d), the levels of c and d are lowered. Finally, we must also update the information of a and b to correctly reflect that c is no longer their successor. This is done by applying CN(ac), CN(bc). At this point, no more operators can be applied and thus we have reached the fixpoint.*

2.1.2 Operators for Backward DAG Traversal

The goal of the second phase is to update the dependency $\delta(u)$ and the contribution to the centrality $BC(u)$ for each node u, based on equations (2) and (1), respectively. This is achieved by applying the single operator in Fig. 5(a) until we reach a fixpoint. The operator is applied on a single edge (u, v) of the graph, such that $u \in \mathit{preds}(v)$. Hence, (u, v) is also an edge of the shortest-path DAG D. The operator guard is:

$$g := \mathit{succs}(v) = \emptyset \wedge u \in \mathit{preds}(v)$$

When the operator is applied, the value of $\delta(u)$ is updated based on the value of $\delta(v)$ as specified by Eq.(2), that is:

$$\delta(u) \stackrel{+}{=} \frac{\sigma_{su}}{\sigma_{sv}}(1 + \delta(v))$$

Additionally, v is removed from the successors of u, and u from the predecessors of v. Finally, $BC(v)$ is updated conditionally, based on $\delta(v)$. This happens during the update of the last predecessor u of v. The operator applies when $\mathit{succs}(v) = \emptyset$, that is, when v has no successors, or has received updates from all its successors. This way, the backward traversal of the DAG D is performed in a data driven manner, breaking away from a level-parallel implementation. In Fig. 5(b) we present a composite operator that is produced by merging together a number instances of the above backward traversal operator. This is an instance of an optimization we call *operator merging*, discussed in detail in § 3.3.

2.2 Operators for Weighted Graphs

In Fig. 6 we show the operators for the weighted case. We assume that each edge (u, v) has a strictly positive weight $w(uv)$. *SP*

Initial state: $\forall u \in V \setminus \mathit{Root}$: $[\sigma, l, \mathit{preds}, \mathit{succs}](u) = (0, \infty, \emptyset, 0)$, $[\sigma, l, \mathit{preds}, \mathit{succs}](\mathit{Root}) = (1, 0, \emptyset, 0)$
$\forall (u, v) \in E$: $[l, \sigma](u, v) = (\infty, 0)$

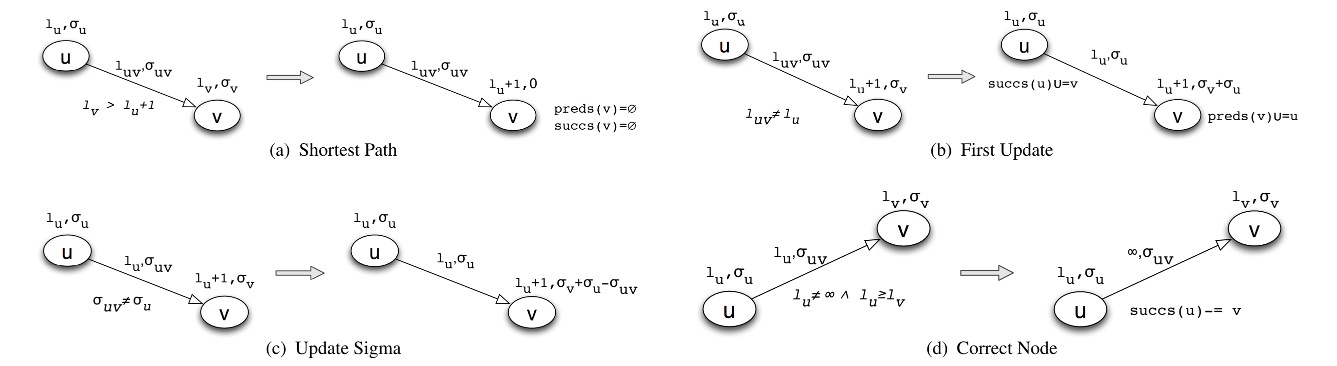


Figure 3. Operators for shortest-path DAG construction phase for unweighted graphs.

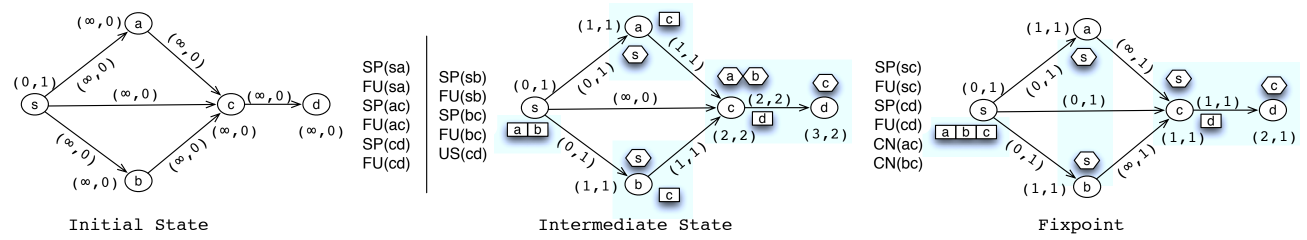
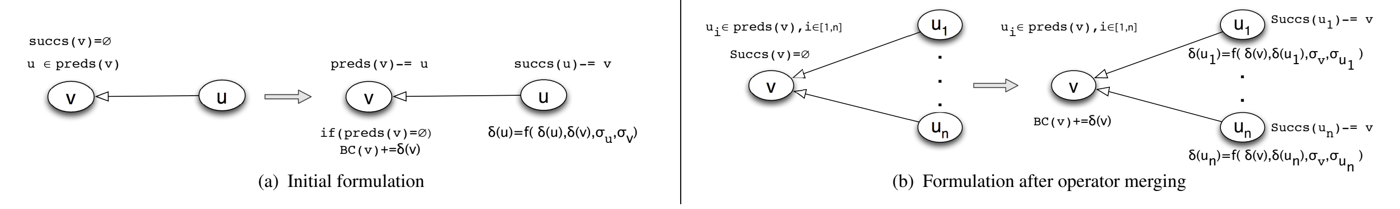


Figure 4. Sample sequential execution of the algorithm. The graph state is shown before, during and after the algorithm execution. Each node u is decorated with $(l(u), \sigma(u))$. Square boxes represent $\mathit{succs}(u)$ and hexagon boxes represent $\mathit{preds}(u)$. In the first fragment, operators in the left column execute before the ones in the right column.

Initial state: $\forall u \in V$: $\delta_s(u) = 0$ | **Initial state:** $\forall u \in V$: $\delta_s(u) = 0$



succs(v)=∅
u ∈ preds(v)
preds(v)−= u
succs(u)−= v
if(preds(v)=∅) BC(v)+=δ(v)
δ(u)=f(δ(u),δ(v),σ_u,σ_v)
(a) Initial formulation
u_i∈ preds(v),i∈[1,n]
Succs(v)=∅
BC(v)+=δ(v)
Succs(u_1)−= v
δ(u_1)=f(δ(v),δ(u_1),σ_v,σ_u1)
Succs(u_n)−= v
δ(u_n)=f(δ(v),δ(u_n),σ_v,σ_un)
(b) Formulation after operator merging

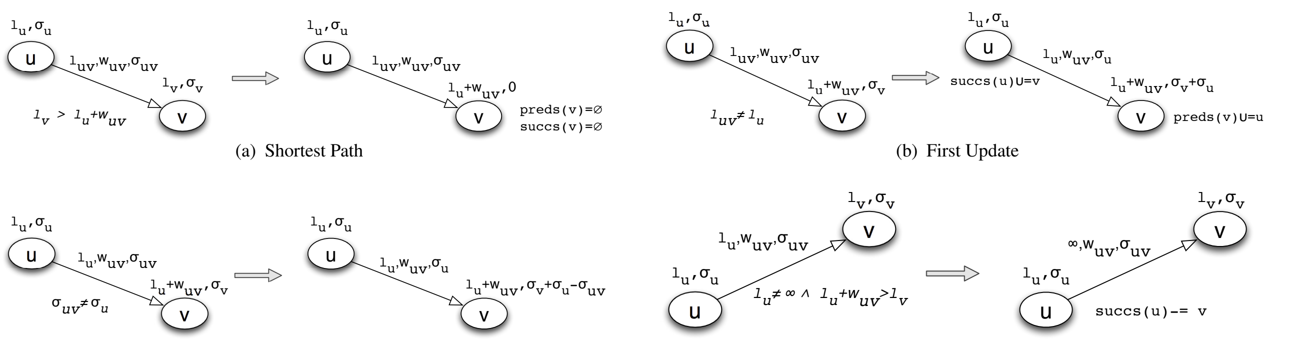
Figure 5. Operator for backward DAG traversal phase

discovers a new shortest path from *Root* to v through u when $l(u) + w(uv) < l(v)$ and sets $l(v) = l(u) + w(uv)$. Similarly, g_{FU}, g_{US} are changed to properly identify successive nodes on shortest paths from *Root*. Finally, g_{CN} is modified to check for $l(u) + w(uv) > l(v)$ in order to capture the case when a shorter path v has been discovered after v has received an update from u. It is easy to see that the operators in Fig. 3 are derived from the ones in Fig. 6 by setting $w_e = 1$ for each edge e in the graph. The second phase is the same as in the unweighted case.

2.3 Characterizing BC Algorithms

Algorithms for BC in the literature can be viewed as implementations of particular schedules for the operators of Figures 3, 5 and 6. Some of the operators are not required for certain schedules.

Unweighted graphs: We first describe algorithms for unweighted graphs. The algorithms by Bader [5] and Madduri [29] build the BFS DAG level by level; each level is built in parallel, with barrier synchronization between levels. The construction of each level essentially involves executions of the *SP* and *FU* operators of Fig. 3; *US* and *CN* are unnecessary in such level-by-level algorithms since nodes reach their final levels in a single step rather than being lowered to that level by relaxation, and the path-counts of nodes at the current level are finalized before moving to the next level. The second phase of [5], which traverses the DAG backward to compute betweenness centrality, can be described by applications of the operator in Fig. 5(b); once again, these applications are not performed in a data-driven manner but in a level-parallel manner using an auxiliary stack data structure populated in the first phase. The second phase of [29] operates in a similar manner except that each node “pulls” information from all its successors instead of “pushing” information to its predecessors, and can be described by a similar operator. Approaches that exploit just coarse grained [16, 26] or both coarse and fine-grained parallelism [5] are also straightforward to describe with the same operators. An open question is what is the best policy for mapping operators from multiple iterations to the available computational resources. Approaches such as [36, 39] perform the level synchornous approach using the op-



(a) Shortest Path (b) First Update (c) Update Sigma (d) Correct Node

Figure 6. Operators for shortest-path DAG construction phase for weighted graphs. Initialization same as in Fig. 3

erators of Fig. 3. Logically partitioning the graph among threads and having each thread be responsible for its partition, is a different way of scheduling the operators and of achieving their atomic execution. Similarly, [12] can be seen as a SIMDization of the operators of Fig. 3. Data structure optimizations in the implementation of the graph abstract data type, such as the re-layout of graph nodes [16] or the distributed storage of the predecessor list [39], are orthogonal to our description of the algorithm and their effect on improving performance is complementary.

Weighted graphs: The algorithm for weighted graphs by Edmonds et al. [19] breaks down the forward pass in a number of sub-phases. It performs a first phase where applications of *SP* and *FU* are applied asynchronously in order to update the distance of nodes and the predecessor lists. Then in a subsequent phase, the path-counts are updated by using applications of the *US* operator in a level-synchronous manner. The *US* operator in that phase is simplified in that no corrections in the estimation of σ are necessary, due to the constraint on the schedule. Essentially, [19] acts as a label-correcting algorithm for only a subset of the node attributes $(l(u), preds(u))$ and as a label-setting algorithm for the rest $(\sigma(u), succs(u))$. Therefore, it is restricted to level-synchronous schedules for the computation of the latter. Our operators, on the other hand, provide a label-correcting capability for all node attributes and are able to merge the above two phases in a single, fully asynchronous phase, potentially exposing more parallelism. Finally, the backward DAG traversal is performed asynchronously using the operator in Fig. 5(b).

2.4 Correctness of BC Operators

In this section, we state the main correctness results of our operator formulation of BC.

Forward Pass We start with the operators for the forward pass, presented in Fig. 3. We prove that the first phase terminates, and that upon termination all node attributes have correct values. We consider the most general setting where operators are allowed to execute in any order. The computation is modeled by a *history*, which is a sequence of operator applications, each of which is considered to be an instantaneous event that changes the state of the graph. We denote an operator application on an edge (u, v) by $op(uv), op \in \{SP, FU, US, CN\}$. To capture meaningful computations, we restrict attention to *well-formed histories*, which are histories where each time an operator is enabled on an edge, its execution cannot be postponed indefinitely. Correctness follows from the following two theorems.

Theorem 1. *(Termination) Any well-formed history H of events $op(uv), op \in \{SP, FU, US, CN\}$ of the operators in Fig. 3 to a graph $G = (V, E)$ has finite length.*

Theorem 2. *At the fixpoint, the following facts hold for an arbitrary node v:*

***(a)** $l(v)$ is equal to the length of the shortest path to v from Root. **(b)** u is the predecessor of v in a shortest path to v from Root $\iff u \in preds(v)$ and $v \in succs(u)$. **(c)** $\sigma(v)$ is the number of shortest paths from Root to v.*

Due to lack of space, the proofs are presented in [35]. Here we briefly state the main ideas. Theorem 1 is proven by showing that each operator can appear only a finite number of times in an arbitrary history H. We prove that the *SP* operator appears only a finite number of times, and that successive *SP* applications partition H into "windows", within which we can only have a finite number of *FU*, *US*, *CN* applications. Theorem 2(a): We consider the partitioning of nodes $P = \{P_0 \dots P_n\}$, where P_0 contains the *Root* and P_i contains nodes at shortest path distance i, that is, nodes directly connected to nodes in P_{i-1} but not to nodes in $P_0, \dots, P_{i-2}$. We show that at the fixpoint the partitioning induced by our algorithm is equivalent to P. Theorem 2(b): ($\Rightarrow$) We examine an arbitrary history, and show that for an arbitrary edge (u, v) there always exists an *FU* application that updates appropriately $succs(v), preds(v)$ after u, v finally settle as successive nodes on a shortest path from *Root*. ($\Leftarrow$) Considering $u \in preds(v)$ and $v \in succs(u)$ at the fixpoint we show by induction on $l(u)$ that there exists a shortest path to v through u. Theorem 2(c): By induction on the shortest path length, using the fact that at the fixpoint $\sigma(v) = \sum_{u \in preds(v)} \sigma(u)$.

Backward Pass We now discuss the correctness of the operator in Fig. 5(a) for the backward pass. Proving termination is straightforward. Initially there is a fixed number of predecessor edges between the nodes comprising the shortest-path DAG. Each operator application depends on finding one such predecessor edge (u, v) and removes it from the graph. Therefore, the number of predecessor edges decreases monotonically and eventually becomes zero. At that point no more applications are enabled and the algorithm terminates. Correctness at the fixpoint follows from the theorem below (the proof is available in [35]):

Theorem 3. *Let $preds_{init}(v), BC_{init}(v)$ denote the values of the respective node attributes at the beginning of the backward pass. At the fixpoint, the following facts hold for an arbitrary node v: **(a)** $\delta(v) = \sum_{w:\ v \in preds_{init}(w)} \frac{\sigma_{sv}}{\sigma_{sw}}(1 + \delta_s(w))$ **(b)** if $v \neq Root$ then $BC(v) = BC_{init}(v) + \delta(v)$, else $BC(v) = BC_{init}(v)$.*

3. Derivation of New Asynchronous Algorithms

The operators in Figures 3, 6, 5(a) can be applied in any order, and as long as no enabled operator application is postponed indefinitely, the implementation will terminate and produce the right BC values. However, the number of operator applications, which is one measure of work-efficiency, may be very different for different orders. In addition, some orders may exploit locality better than others. Getting a scalable solution greatly depends on performing the right operator scheduling. Exploiting scheduling to improve parallelism is a well-studied area [6–8, 14]. Here, we derive asynchronous algorithms by choosing particular scheduling policies for operator execution.

A simple approach is to repeatedly scan all the edges of the graph in some order, applying all applicable operators to an edge when it is visited. This is inefficient since most edges will not have any operators that can be applied to them. Instead, we use a worklist-based approach. The unit of work in our setting is processing a single edge, which may trigger applications at neighboring edges. Therefore, we maintain a multiset of edges, implemented as a dynamic worklist *Wl*. At each step, we pick an edge e from *Wl*, apply an operator to it, and add to *Wl* new edges that may need processing. In a concurrent setting, *Wl* must support concurrent `add` and `poll` operations. Atomic operator execution is achieved by acquiring locks on the edge end-points during the operator execution.

In § 3.1 we discuss the most performance-critical aspect of our design: the policy for processing *Wl* elements. In § 3.2 we discuss how to incrementally update the *Wl*. In § 3.3, § 3.4 we describe key scheduling optimizations for improving the basic algorithm. Finally, in § 3.5 we present two algorithm variants that we can derive by exploiting all the scheduling insights.

3.1 Choosing a Worklist Processing Policy

During the algorithm execution, *Wl* will usually contain many edges where operators may be applied. An important design decision is picking a work-efficient order to process edges, since this will affect the convergence rate of the algorithm [24, 30, 31].

During the DAG construction phase, the shortest-path exploration (*SP* operator) is the backbone of our algorithm. The number of all other operator applications is ultimately a function of the times we mispredict the length of the shortest path to a node. Hence, a good ordering policy in our case is one that efficiently identifies shortest-paths, but which is also flexible enough to allow threads to optimistically try to expose parallelism. In the unweighted case, for example, a level-parallel approach allows only nodes within successive levels to be processed concurrently and can stifle parallelism if there are not enough nodes in a level. However, an approach that allows exploration of arbitrary paths on the graph may end up performing too much wasted work since it may mispredict the distance of nodes multiple times.

Our approach is loosely based on the Δ-stepping approach of Meyer *et al.* [30] and effectively tackles such problems. We partition the edges (u, v) in *Wl* into equivalence classes based on an approximation $l^*(v)$ of $l(v)$ defined as $l^*(v) := \frac{l(u)+w(uv)}{\Delta}$. The term $l(u) + w(uv)$ is the length of the path from the *Root* to v through u. Δ is a user specified parameter that defines a range of distance values that fall in the same equivalence class. Each equivalence class is associated with a bucket B_i. Each time an edge (u, v) is inserted in *Wl*, we compute $l^*(v)$ and place it in the appropriate B_i. Intuitively, we want to explore shorter paths before longer ones in order to minimize the number of mispredictions on the level of a node. Hence, our policy places these edges in lower-numbered buckets. Threads query the buckets for work in increasing order. The deviation from the classic Δ-stepping is that *there is no barrier between the processing of successive buckets*. Each thread queries buckets in increasing order repeatedly until no more work is left, but different threads can be simultaneously operating on different buckets and, consequently, can extract more parallelism in case there is limited amount of work in a particular bucket. The role of Δ is to control the interplay between exposing parallelism, decreasing the probability of mispredicting levels, and controlling the overhead of iterating over and querying buckets in a concurrent setting. This strategy is enabled only because our operators are general enough to restore the correct values of attributes, in case of misspeculation. Note that the equivalence class of an edge can change after the point it is inserted in B_i. This however can only affect the performance and not the correctness of our algorithm. To mitigate the overhead even further, each thread gets a chunk of edges out of B_i each time it queries *Wl*.

The backward DAG traversal phase operates in a data-flow driven manner. When the δ value of a node is fully updated, we propagate changes backward by scheduling updates to its DAG predecessors. This scheme avoids a level-parallel traversal of the shortest-path DAG and can expose more concurrency in the case where the DAG contains independent components. Edges to be processed are inserted into a worklist. A FIFO or LIFO based worklist policy gave us the best performance.

The best value of Δ and of the other scheduling parameters varies across inputs. § 4 provides details about the chosen values.

3.2 Incrementally Maintaining the Worklist

We first discuss the policy for the DAG construction phase. Finding the minimal set of edges that need to be processed, and therefore need to be added to *Wl*, due to an operator application on edge (u, v) is challenging in a concurrent setting. This is because the immediate neighborhoods of u and v are, in general, unbounded. Therefore, we over-approximate the minimal set of edges by using the following policy: Whenever an operator changes the attributes of a node v, we add to *Wl* all edges adjacent to v that *may* need to be updated due to the change to v. We determine these edges for each operator as follows.

***SP*:** An *SP*(uv) moves v to a new level and identifies a new shortest path P to v. All outgoing edges of v are inserted to *Wl* so that the exploration along P continues. Additionally, incoming edges of v must be examined to correctly update neighboring nodes $w \neq u$ that lie on paths P' (longer than P). Such nodes may have recorded v as their successor. Hence, we add all adjacent edges to v to *Wl*.

***FU*, *US*:** Both operators update $\sigma(v)$, so all outgoing edges of v are added to *Wl* to further propagate this update.

***CN*:** This operator simply corrects the successor list of the source node u of the edge it is applied on and does not enable any other operators. Hence no updates to *Wl* are required.

To initialize *Wl*, we insert all outgoing edges of the *Root* to initiate the shortest-path explorations.

For the backward DAG traversal we use a data-flow driven policy that considers the minimal set of edges. Whenever $succs(v) = \emptyset$ an operator is enabled on each edge (u, v) between v and $u \in preds(v)$. In the baseline scheme, we add each such (u, v) to *Wl*. When operator merging is applied (see § 3.3), *Wl* can simply contain nodes; in this case we insert v to *Wl*. We initialize *Wl* with all (u, v) (or v), such that v is at the fringe of the DAG generated in the first phase. Such nodes are easy to identify after the first phase, since they have no successors. We perform a scan over all nodes and populate *Wl* appropriately, when this property is satisfied.

3.3 Optimization I: Operator Merging

In certain cases it is beneficial for performance reasons to merge multiple operators together and create a new composite operator. Operator merging is essentially a transformation that binds scheduling decisions at compile-time and provides benefits similar to classical loop optimizations. Co-scheduling the application of multiple operators can improve locality. Also, in a similar spirit to loop fusion, combining multiple operators together improves locality and reduces loop overheads. We identify two cases where merging is applicable and describe additional performance benefits it provides.

In the first phase, every $SP(uv)$ will lead to a subsequent $FU(uv)$. Merging the two firstly improves locality, since the data of v is already local to the thread due to $SP(uv)$. Second, it reduces the locking overhead, by eliminating locking for $FU(uv)$. Finally, it reduces the pressure on *Wl*, since we need insert the outgoing edges of v to *Wl* only once in the combined operator.

In the second phase, we can merge all applications of the operator between v and its predecessors u_i. The composite operator is shown in Fig. 5(b). Merging firstly permits all updates starting from v to be performed in a single operator application. Thus, we can avoid the removal of u from $preds(v)$, because now we do not need to keep track of whether processing v is over. In order to update $BC(v)$ correctly the composite operator needs to be applied once per node v and its predecessors. This can be achieved by simply inserting a single instance of v into Wl, upon receiving an update from the last successor. A special check to avoid updating *Root* is also necessary. Note that we can perform fine-grained locking on each u_i and do not need to access atomically an unbounded number of nodes.

3.4 Optimization II: Improving the Worklist Update Policy

In § 3.2 we discussed a policy that over-approximates the set of edges that need to be processed after an operator application, and therefore need to be added to *Wl*. We identify various ways of improving that policy. Our optimizations reduce the number of redundant checks on parts of the graph for enabled operators and the number of `add` calls to *Wl*, thus leading to fewer accesses to shared resources. In practice they are important in speeding up the first phase of the algorithm.

First, when an $SP(uv)$ is executed, if $preds(v) = \emptyset$, then no incoming neighbor u of v needs to be processed, since $v \in succs(u)$ iff $u \in preds(v)$. Hence, we can avoid inserting incoming edges of v to *Wl*.

Second, when a $US(uv)$ is executed, if $succs(v) = \emptyset$, then we can avoid adding outgoing edges (v, w) of v to *Wl*. The reason is that in order for $US(vw)$ to be executed, w must already be a successor of v (an $FU(uv)$ must have been executed). Since v has no recorded successors, all (v, w) are already in *Wl* waiting to be processed, and we can avoid re-inserting them. This way, we expose opportunities to batch up all path-count updates and avoid redundant *Wl* population.

Third, when using the combined $SP(uv), FU(uv)$ operator, we can avoid adding outgoing edges (v, w) after an $FU(uv)$ if $succs(v) = \emptyset$, since they are already there due to the application of the combined operator. Note that all these optimizations are inexpensive since they require simple checks on operator-local state, and do not require acquiring extra locks.

3.5 Putting it All Together: Derivation of Two Asynchronous Variants

We now describe how to combine all the ideas presented in the previous sections in order to produce two asynchronous algorithms.

In the first algorithm, *async1*, during the forward phase (line 4 in Fig. 1), each thread extracts an edge out of the worklist and tries to find an operator to apply to it by checking the operator guards in some arbitrary order, while also merging applications of *SP* with *FU*. The order we chose is $[CN, SP \circ FU, FU, US]$. Note that the guards are mutually exclusive, therefore any order of checking the guards is guaranteed not to postpone an operator application indefinitely. For the backward phase (line 7 in Fig. 1), each thread extracts a node out of the worklist and tries to perform the composite operator in Fig. 5(b) to all predecessor nodes. In Fig. 7 we show in pseudocode for the forward pass of *async1*.

```
Worklist  Wl = $\{(srcNode, w) : (srcNode, w) \in G.edges\}$

foreach $(u, v) \in$ Wl {
  lock(u,v)
  if ($g_{CN}(u, v)$) {
    apply $CN$ ; unlock(u,v)
  } else if ($g_{SP \circ FU}(u, v)$) {
    apply $SP \circ FU$; unlock(u,v)
    $Wl = Wl \cup \{(v, w) : w \in outNbrs(v)\}$
    if (uHasPreds)
      $Wl = Wl \cup \{(w, u) : w \in inNbrs(u)\}$
  } else if ($g_{FU}(u, v)$) {
     apply $FU$; unlock(u,v)
     if (vHasSuccs)
       $Wl = Wl \cup \{(v, w) : w \in outNbrs(v)\}$
  } else if ($g_{US}(u, v)$) {
     apply $US$; unlock(u,v)
     if (vHasSuccs)
       $Wl = Wl \cup \{(v, w) : w \in outNbrs(v)\}$
  } else { unlock(u,v) }
}
```

Figure 7. Pseudocode for forward pass of *async1*.

In the second algorithm, *async2*, we consider a node and all its immediate neighbors, and statically co-schedule operator applications on the edges connecting them. Additionally, we perform potential `CN` applications in place whenever we discover a new shorter path to a node. The motivation behind these design choices is to exploit spatial and temporal locality by having a a single thread work on an entire neighborhood. A potential issue with this design though is that it may be harder to load balance work, in case the node degrees are not evenly distributed. This can be more problematic in an environment with high degree of parallelism. The pseudocode for *async2* is presented in Fig. 8. Note that in this algorithm we insert nodes instead of edges into the worklist. The second phase is the same as in *async1*.

To guarantee atomic execution of operators we acquire fine-grained spinlocks on the end-points of each edge that an operator works on. The graph nodes are totally ordered based on their runtime (allocation) addresses. We acquire locks respecting this order to avoid deadlock between threads working on the same edge. To amortize locking overhead we acquire the locks in the beginning of the loop iteration and release them after the first successful operator application, or at the end if no operator is applicable. In case of a successful operator application we release a lock on a node immediately after the last access to a node attribute. Inserting edges/nodes in *Wl* after an operator application is done without holding any locks. This is because this action does not access any data attributes, and also because the graph structure is not mutated. This way we significantly reduce the length of atomic sections and allow more fine-grained thread interleavings.

4. Experimental Evaluation

We implemented two versions of the algorithm, based on the operators in Fig. 3 and Fig. 6 and compared their performance against a number of publicly available versions of BC algorithms for weighted and unweighted graphs. We ran our experiments on two architectures. First, an Intel Xeon machine (**Nehalem**) running Scientific Linux 6.3 with four 6-core 2.00 GHz Intel Xeon E7540

```
Worklist  Wl = $\{srcNode\}$

foreach $u \in$ Wl {
  forall $v \in outNbrs(u)$ {
    lock(u,v)
    if ($g_{SP \circ FU}(u,v)$) {
      apply $SP \circ FU$; unlock(u,v)
      $Wl = Wl \cup \{v\}$
      if (vHasPreds) {
        // Inline CN applications
        forall $w \in inNbrs(v)$ {
          lock(v,w)
          if ($g_{CN}(w,v)$) { apply CN }
          unlock(v,w)
        }
      }
    } else if ($g_{FU}(u,v)$) {
      apply FU; unlock(u,v)
      if (vHasSuccs) $Wl = Wl \cup \{v\}$
    } else if ($g_{US}(u,v)$) {
      apply US; unlock(u,v)
      if (vHasSuccs) $Wl = Wl \cup \{v\}$
    } else { unlock(u,v) }
  }
}
```

Figure 8. Pseudocode for forward pass of *async2*.

processors that share 128 GB of memory. Second, a Sun T5440 machine (**Niagara**) running SunOS 5.10. It contains two 8-core 1.4 GHz Sun UltraSPARC T2 Plus (Niagara 2) processors, and provides 128 concurrent hardware threads, sharing 32 GB of memory. On the Nehalem, the compiler used was GCC 4.7.1. On the Niagara, the compiler used was GCC 4.5.1. We report the average time (*t*) of 5 runs and the standard deviation (*sd*). We consider the following classes of input graphs:

- Real-world road network graphs of the USA from the DIMACS shortest paths challenge [2]. We use the full USA network (*USA-net*) with 24M nodes and 58M edges and the central USA network (*USA-ctr*) with 14M nodes and 34M edges.
- A network of scientific co-authorships [32] (*coauth*) with 391K nodes and 873K edges, where edge weights are converted to integers (by multiplying all weights by 1000).
- Scale-free graphs generated using the Recursive MATrix (R-MAT) scale-free graph generation algorithm [13]. The size of the graphs is controlled by a *SCALE* parameter; a graph contains $N = 2^{SCALE}$ nodes, $M = 8 \times N$ edges, with each edge having strictly positive integer weight with maximum value $C = 2^{SCALE}$. The RMAT graphs we used were generated using the tools provided by the SSCA v2.2 benchmark [3]. The parameters used for the graph construction were the default ones, as specified by the generator ($a = 0.55, b = 0.1, c = 0.1, d = 0.25$). For our experiments we removed multi-edges from the graphs. We denote a graph of $SCALE = X$ as $rmatX$.
- Random graphs containing $N = 2^k$ nodes and $M = 4 \times N$ edges. There are $N - 1$ edges connecting nodes in a circle to guarantee the existence of a connected component and all the other edges are created randomly, following a uniform distribution. A graph with $k = X$ is denoted as $randX$.

For large-scale graphs, it is computationally infeasible to compute BC by doing shortest path computations from every node of the graph. Therefore, like previous studies [4, 16, 19, 29], we perform shortest path computations for only a subset of nodes.

Nehalem	Forward		Backward
coauth	$\Delta = 512$	$CF128$	$CL256$
USA-net	$\Delta = 32768$	$CF64$	$CL256$
rmat25	$\Delta = 32768$	$CF256$	$CL256$
Niagara	Forward		Backward
coauth	$\Delta = 2048$	$CF128$	$CL256$
USA-net	$\Delta = 32768$	$CF128$	$CL256$
USA-ctr	$\Delta = 32768$	$CF128$	$CL256$

Table 1. Scheduling parameters for weighted graph experiments.CFx (CLx): Chunked FIFO (LIFO) with chunk size x.

4.1 Experiments on Weighted Graphs

We consider two algorithms for weighted graphs: (i) our algorithm *async1*, which is based on the operators of Fig. 6, with all scheduling optimizations enabled, and (ii) a serial reference implementation of Brandes' algorithm (**boost-s**) for weighted graphs, available in the Boost Graph Library [37] V. 1.47.0. The only parallel algorithm for weighted graphs with a publicly available implementation that we are aware of is [19]. However, that solution targets a distributed-memory environment while our implementation targets shared-memory multicores, so a direct comparison of performance is not meaningful.

4.1.1 Implementation Details

We parallelized *async1* in C++ using the Galois system [1]. Each of the two major phases of the algorithm in Fig. 1 is implemented as a parallel `foreach` loop over a worklist of edges and nodes, respectively. Below we discuss in more detail several aspects of our implementation.

Graph data-structure Our graph implementation is based on the compressed sparse row (CSR) format, with node and edge data stored in two different arrays. Our graphs were not initialized in a NUMA aware manner. In our experience this does not have an observable impact on our experimental machines, since they do not have deep NUMA hierarchies.

Worklist policies We use worklist implementations provided by the Galois system [1, 31]. In Tab. 1, we present the scheduling parameters that gave us the best performance. These values were obtained by performing a small manual search on the parameter space. A more exhaustive search can potentially lead to improved performance; we plan to examine this in future work. Recall that in the forward pass, we use a delta-stepping-like policy that prioritizes the work-items into buckets using the parameter Δ. Within each bucket, our scheme follows a `FIFO` or `LIFO` policy with additional chunking of work. For the backward pass, we use either a chunked `FIFO` or a chunked `LIFO` policy. For example, for the *coauth* graph forward pass on the Nehalem we use $\Delta = 512$ and each bucket is implemented using a `FIFO` policy with chunk size of 128 edges. For the backward pass we use a `LIFO` with chunk size 256.

4.1.2 Analysis of Results

Tab. 2 presents results for various real-world and one synthetic graph. Our algorithm scales very well on all graphs across both architectures. On Nehalem, *async1* achieves scalability of $9.5\times$ on both the USA road-network, and the *rmat25*. The high thread count on the Niagara, allows *async1* to mine the available parallelism through the fine-grained operator execution. For the two road-networks it achieves its best scalability, $32\times$ and $37\times$, at 64 threads. By exploiting cross-level parallelism *async1* is able to achieve scalability even on the really small co-author network ($4\times$ on Nehalem, $18\times$ on Niagara). We report the *boost-s* runtime to provide a reference against a publicly available serial implementation.

Nehalem	coauth (500 steps)		USA-net (10 steps)		rmat25 (10 steps)	
Threads	*t*	*sd*	*t*	*sd*	*t*	*sd*
boost-serial	68	0	510	4	3020	53
1	32	3	285	1	1493	102
4	12	0	78	0	383	27
8	10	0	43	1	223	6
12	10	0	34	0	170	8
16	8	0	31	0	163	6
20	8	0	30	0	159	10
24	8	0	30	0	157	15

Niagara	coauth (100 steps)		USA-net (10 steps)		USA-ctr (10 steps)	
Threads	*t*	*sd*	*t*	*sd*	*t*	*sd*
boost-serial	83	0	1872	107	1086	1
1	110	0	1583	6	981	5
16	9	0	115	0	69	0
32	6	0	65	0	37	0
48	6	0	53	0	29	0
64	7	0	49	0	26	0
96	10	0	53	1	26	0
128	15	0	61	2	31	2

Table 2. Average execution time (sec.) and stdv. of *async1* for weighted graphs

4.2 Experiments on Unweighted Graphs

We evaluated a number of BC algorithms for unweighted graphs. Below, we describe each of them and give details about the parallelization strategy and scheduling policy. We note that the graph format for all unweighted algorithms is based on the CSR format.

outer This is a parallelization only of the outer loop. The graph state is replicated P times, where P is the number of threads. Each thread performs an iteration of the outer loop which is mostly independent from other iterations. The updates on the betweenness value of each node form a reduction which is straightforward to handle. The serial algorithm executed in the inner loop by each thread uses the successor lists to represent the DAG, as discussed in [29]. The algorithm was implemented in the Galois system.

async1*, *async2 These are our algorithms based on the operators of Fig. 3 with all scheduling optimizations enabled. *async1* was used on the Niagara experiments, while *async2* was used on the Nehalem experiments. The algorithms were implemented in the Galois system following the same design choices as the weighted version in § 4.1.1. We now discuss the scheduling parameters that we used. On both architectures we select $\Delta = 1$. The intuition behind this is that in the case of unweighted graphs the diameter is low, which potentially increases the amount of work per level. Setting $\Delta = 1$ focuses the search for work in individual levels, while still allowing for cross-level speculation. We note that setting $\Delta = 1$ *does not make them level-synchronous algorithms*. As discussed in § 3.1 these algorithms allow threads to simultaneously work on an arbitrary number of buckets. Setting $\Delta = 1$ simply reduces the speculation window for the threads but does not restrict them to a single bucket. If bucket B_i becomes temporarily empty, threads move to buckets $T_j, j > i$, and can return later to T_i if new work exists there. This strategy is applicable only because our operators are general enough to restore the correct values of attributes in case of mispeculation. On the Nehalem, where *async2* is used, each bucket is processed using a `LIFO` policy. For `rmat25` we use a chunk size of 8 nodes and for `rand26` we use chunks of 32 nodes. On the Niagara, where *async1* is used, each bucket is processed using a `FIFO` policy with chunks of edges of size 512. The second phase uses a `LIFO` policy with chunk sizes of 16 (Nehalem) and 2048 (Niagara).

preds This algorithm is an inner-level, level-synchronous parallelization presented in [5]. The implementation is part of the SSCA v2.2 benchmark [3]. The implementation uses OpenMP directives to parallelize the loops and define their schedules, and OpenMP barriers for the synchronization of levels. The OpenMP scheduling policy for the forward phase is `dynamic` and for the backward phase is `static`.

succs/succs-serial This algorithm is an inner-level, level synchronous parallelization presented in [29]. The implementation is our adaption of the implementation provided in GraphCT v.0.5 [18]. GraphCT is a parallel toolkit for analyzing massive graphs on the massively multithreaded Cray XMT; the implementation is optimized for the Cray and uses compiler directives specific to that system. The algorithm, as presented in [29] admits a lock-free implementation. Our adaption is a parallelization using OpenMP. We replace the original atomic intrinsics with the equivalent ones provided by GCC to implement it. We experimented with various scheduling policies and present results for the `guided` policy, which performed the best. This algorithm also serves as the serial baseline that we compare all algorithms against.

4.2.1 Analysis of Results

In Tab. 3 we present the running time of all four algorithms for 100 iterations of the outer loop. Due to large running times, we present results only for high thread counts on the `rmat25` graph, on the Nehalem. In Tab. 4, Tab. 5 we focus on the inner-loop parallelization strategies and present results for 10 iterations of the outer loop.

First, we discuss performance on the Nehalem (Tab. 3, Tab. 4). We can make a number of interesting observations. We note that for `rmat25` *outer* is the best performing. We expect this to be the case when the graph fits in the available memory. However, this comes at the cost of high memory usage; for 24 threads *outer* requires about 63% of the 128 GB of main memory and it will exceed the machine's memory capacity for larger graphs. For this reason, we do not present results for *outer* on `rand26` or on the Niagara, which has less memory and a larger number of threads. Second, *preds* starts by being more than 2× slower than *succs* (due to the locality benefits of *succs*), something that is in accordance with previous studies [29]. At higher thread counts, though, the performance gap between the two is reduced. Interestingly, on our system *preds* is sensitive to thread pinning. Avoiding pinning makes *preds* more than 2× slower than *succs* even on high thread counts (e.g. on 24 threads *preds* takes 174 sec. for 10 steps on `rmat25` on Nehalem). Focusing on *async2*, on the Nehalem, we observe that it is slower than the other algorithms on the `rmat25` graph, while it is the fastest on the `rand26`. We believe this is due to the structure of the input graphs. The average longest BFS level for the outer loop iterations we performed provides an approximation of the graph diameter. It is 15.9 (stdv. 0.7) on `rmat25` and 20.1 (stdv. 0.57) on `rand26`. By increasing the "diameter" of the explored graph, hence decreasing the amount of work per level, an algorithm that tries to mine work from multiple levels becomes comparatively better.

On the Niagara, we experiment with a different variant of our algorithm, *async1*. In *async1* the unit of work is the processing of a single edge instead of a node and its immediate neighbors. Intuitively, this can lead to more fine-grained distribution of work, which can be more suitable for an environment with a high degree of parallelism. *async1* is able to scale to high thread counts on both inputs, as we see in Tab. 5. We do not report performance numbers on *preds*, due to execution problems on this platform. Regarding *succs*, its best running time is 325 seconds (stdv. 35) for `rmat25` and 1044 seconds (stdv. 138) for `rand26`, both on 16 threads. *succs* did not scale beyond 16 threads. We believe this behavior is partly due to the poor exploitation of parallel resources by the level-parallel approach (more threads means less work per thread on each level) and partly to scaling issues in the OpenMP runtime on the Niagara.

4.3 Assessing the Effectiveness of Scheduling

To illustrate the effect of scheduling on the performance we consider one more experiment using *async2* on the *rmat25* unweighted graph on the Nehalem architecture. We execute two outer loop iterations and record the number of executed operators. We consider the following variants of *async2*: The first, `Ord`, is the version we already presented with all scheduling optimizations enabled. The second variant, `Ord-d`, is a de-optimized version of `Ord` where, the optimizations presented in § 3.4 are disabled. Both `Ord` variants use the same worklist policy. Finally, `CF` is a variant of *async2* with a `FIFO` worklist policy (with chunk size of 16 nodes) and all other scheduling optimizations enabled. Using a `FIFO` schedule for the unweighted operators gives us the optimal order of processing the edges in the sequential setting. The `CF` worklist policy maintains this FIFO order for each thread in the parallel setting, while relaxing the constraint of maintaining a total FIFO ordering among threads, to achieve better scalability. In Tab. 6, we report the average number of operator applications and runtime of the forward pass per outer loop iteration.

Firstly, note that for `Ord` and `CF` the operator counts for one thread are identical. This is natural since, with $\Delta = 1$, on one thread `Ord` emulates the best schedule (`FIFO`). Additionally, we see that the number of *US* applications and *CN* evaluations is zero. This is normal, since in the best schedule no corrections should be done to the path-count and we should never mispredict the level of a node. This is not the case for `Ord-d`, though, which disables some of the optimizations and performs unnecessary work. Also, we note that not all nodes and edges will be reachable from all possible roots, in a directed graph, therefore the number of operator applications will not, in general, match the node and edge count of the graph.

Focusing on runtime performance, we observe a correlation between the number of operator applications and the running time. `CF` deviates quickly from the optimal schedule in terms of operator applications. For 4 threads and above we observe an increase on the number of *SP* operators, which translates to increases in the number of other operators. Although we increase the number of threads, the extra work eliminates the gain from the increased degree of parallelism, something that hinders both scalability and absolute performance. `Ord-d` does not deviate from the best schedule in terms of *SP* applications, due to the use of the right scheduling policy, but lacking the worklist maintenance optimizations ends up performing much more work (observe the increased values of *e(CN)*, *None*). `Ord`, on the other hand, combines both a good policy for processing the worklist elements along with a careful policy for populating the worklist with new work, and manages to outperform both other variants. As the thread count increases it experiences a much smaller increase in the number processed elements (mainly manifesting as increased *e(CN)* and *None* values), but scalability is mainly hindered by a slower per-operator processing time.

Nehalem	**async2**		**succs**		**preds**		**outer**	
Threads	*t*	*sd*	*t*	*sd*	*t*	*sd*	*t*	*sd*
16	985	8	785	33	1051	4	275	2
20	1010	6	771	6	997	36	242	1
24	1063	6	802	73	960	8	209	4

Table 3. Execution time (sec) and stdv. for executing 100 outer-loop iterations on rmat25.

Nehalem	**rmat25 (10 steps)**						**rand26 (10 steps)**					
	preds		**succs**		***async2***		**preds**		**succs**		***async2***	
Threads	*t*	*sd*	*t*	*sd*	*t*	*sd*	*t*	*sd*	*t*	*sd*	*t*	*sd*
serial			261	16					601	44		
1	1014	7	385	1	806	11	1636	4	850	4	1449	24
4	309	2	155	1	205	1	515	4	308	7	364	0
8	161	1	92	1	115	1	266	1	199	8	191	1
12	120	1	80	3	101	1	206	4	202	27	144	0
16	101	0	78	4	99	2	195	5	191	19	128	0
20	91	0	76	2	101	1	187	1	191	18	125	0
24	86	1	74	1	106	1	183	0	190	9	126	0

Table 4. Average execution time (sec) and stdv on Nehalem.

Niagara	**rmat25 (10 steps)**		**rand26 (10 steps)**	
	async1		***async1***	
Threads	*t*	*sd*	*t*	*sd*
succs-serial	1794	39	3265	89
1	5019	81	6444	99
16	350	13	409	6
32	209	12	210	6
48	148	5	151	6
64	123	8	123	10
96	121	14	106	14
128	127	24	98	15

Table 5. Average execution time (sec) and stdv on Niagara.

5. Conclusion and Future Work

This paper presents a new formulation of BC in terms of the operator formulation of algorithms. This formulation captures the essence of the problem and allows us to not only express existing solutions as schedules of these operators, but also to derive new asynchronous algorithms that are able to extract large amounts of parallelism, are work-efficient, and work for both weighted and unweighted graphs. Our experiments show that our algorithms benefit from a high-degree of parallelism, achieve good scalability on real world inputs, and perform competitively against other solutions. As future work, we want to target a distributed memory environment. More importantly, our high-level operator formulation of BC opens up an avenue for automatically synthesizing different solutions. This will allow to perform a systematic design-space exploration in order to to find the solution that performs the best under the constraints of a particular input and architecture.

Acknowledgments

We would like to thank Professor Vijaya Ramachandran, the members of the ISS Group and the anonymous referees for their helpful comments. We are grateful to Andrew Lenharth and Xin Sui for many useful discussions.

References

[1] Galois system. http://iss.ices.utexas.edu/?p=projects/galois.

[2] 9th DIMACS Implementation Challenge. http://www.dis.uniroma1.it/~challenge9/download.shtml, 2009.

[3] D. Bader., J. Gilbert, J. Kepner, and K. Madduri. Hpcs scalable synthetic compact applications graph analysis (SSCA2) benchmark v2.2, 2007. http://www.graphanalysis.org/benchmark/.

[4] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In *WAW*, Berlin, Heidelberg, 2007.

[5] D. A. Bader and K. Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *ICPP*, 2006.

[6] G. Blelloch, J. Fineman, P. Gibbons, and H. Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *SPAA*, 2011.

P	SP	FU	US	e(CN)	CN	None	Time/Iter.(sec)
				Ord			
1	25.3	43.5	0*	0*	0*	190.9	56.5
4	25.3	43.5	0.0	0.0	0.0	190.9	14.13
8	25.4	44.1	4.1	5.6	0.6	198.8	8.1
12	25.4	44.0	3.3	6.4	0.5	198.9	6.6
16	25.5	44.0	3.2	9.3	0.6	202.0	6.3
20	25.5	43.7	0.8	16.7	0.3	209.2	6.6
24	25.7	43.8	1.2	27.2	0.6	222.3	7.2
				Ord-d			
1	25.3	43.5	0*	265.2	0*	1934.0	332.2
4	25.3	43.5	0.0	265.5	0.0	1934.8	83.6
8	25.3	43.5	0.0	265.5	0.0	1934.9	50.5
12	25.4	43.5	0.0	265.7	0.0	1935.1	44.2
16	25.4	43.5	0.0	265.7	0.0	1935.2	42.5
20	25.4	43.5	0.0	265.9	0.0	1936.0	42.9
24	25.4	43.5	0.0	266.0	0.0	1935.8	44.5
				CF			
1	25.3	43.5	0*	0*	0*	190.9	55.2
4	26.4	44.2	9.8	23.1	1.8	497.6	24.6
8	28.3	47.1	18.6	68.9	6.1	979.5	25.6
12	28.6	47.0	17.9	77.2	6.3	995.6	23.2
16	29.4	47.6	19.4	104.3	7.2	1036.7	24.5
20	27.6	45.8	18.5	55.7	4.4	841.8	21.1
24	30.1	48.7	21.3	125.1	8.5	1188.7	29.7

Table 6. Average number of operator applications (millions) and runtime of the forward pass per outer-loop iteration, in an execution of 2 iterations on unweighted *rmat25*(Nehalem). *P*: thread count, *e(CN)*: is the number of *CN* evaluations, *CN*: is the number of actual *CN* applications. 0^* denotes an absolute zero value. *None* denotes the number of checked edges for which no operator is enabled.

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *PPOPP*, 1995.

[8] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5), 1999.

[9] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25, 2001.

[10] U. Brandes and C. Pich. Centrality Estimation in Large Networks. *International Journal of Birfucation and Chaos*, 17(7), 2007.

[11] N. Bulkley and M. V. Alstyne. Does e-mail make white collar workers more productive? Technical report, University of Michigan, 2004.

[12] A. Buluc and J. Gilbert. The combinatorial BLAS: design, implementation, and applications. *Int. Journal of High Perf. Computing Applications*, 2011.

[13] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *In SIAM Data Mining*, 2004.

[14] R. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran. Oblivious algorithms for multicores and network of processors. In *IPDPS*, 2010.

[15] T. Coffman, S. Greenblatt, and S. Marcus. Graph-based technologies for intelligence analysis. *Commun. ACM*, 47, 2004.

[16] G. Cong and K. Makarychev. Optimizing large-scale graph analysis on a multi-threaded, multi-core platform. In *IPDPS*, 2011.

[17] A. Del Sol, H. Fujihashi, and P. O'Meara. Topology of small-world networks of protein–protein complex structures. *Bioinformatics*, 2005.

[18] D. Ediger, K. Jiang, J. Riedy, D. A. Bader, and C. Corley. Massive social network analysis: Mining twitter for social good. In *ICPP*, 2010.

[19] N. Edmonds, T. Hoefler, and A. Lumsdaine. A space-efficient parallel algorithm for computing betweenness centrality in distributed memory. In *HiPC*, 2010.

[20] L. C. Freeman. A set of measures of centrality based on betweenness. 1977.

[21] R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *ALENEX*, 2008.

[22] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12), 2002.

[23] R. Guimerà, S. Mossa, A. Turtschi, and L. A. N. Amaral. The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles. *NAS*, 102(22), 2005.

[24] M. Hassaan, M. Burtscher, and K. Pingali. Ordered vs unordered: a comparison of parallelism and work-efficiency in irregular algorithms. In *PPOPP*, 2011.

[25] H. Jeong, S. P. Mason, A. L. Barabási, and Z. N. Oltvai. Lethality and centrality in protein networks. *Nature*, 411, May 2001.

[26] S. Jin, Z. Huang, Y. Chen, D. G. Chavarría-Miranda, J. Feo, and P. C. Wong. A novel application of parallel betweenness centrality to power grid contingency analysis. In *IPDPS*, 2010.

[27] V. Krebs. Mapping networks of terrorist cells. *Connections*, 2002.

[28] F. Liljeros, C. Edling, L. Amaral, H. Stanley, and Y. Åberg. The web of human sexual contacts. *Nature*, 411, 2001.

[29] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. G. Chavarra-miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *IPDS*, 2009.

[30] U. Meyer and P. Sanders. Delta-stepping: A parallel single source shortest path algorithm. In *ESA*, 1998.

[31] D. Nguyen and K. Pingali. Synthesizing concurrent schedulers for irregular algorithms. In *ASPLOS*, 2011.

[32] G. Palla, I. J. Farkas, P. Pollner, I. Derenyi, and T. Vicsek. Fundamental statistical features and self-similar properties of tagged networks. *New Journal of Physics*, 10, 2008. http://cfinder.org/wiki/?n=Main.Data#toc2.

[33] R. Pearce, M. Gokhale, and N. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *SC*, 2010.

[34] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The TAO of parallelism in algorithms. In *PLDI*, 2011.

[35] D. Prountzos and K. Pingali. Betweenness centrality: Algorithms and implementations. Technical Report TR-13-31, UT Austin, Jan 2013.

[36] Z. Shi and B. Zhang. Fast network centrality analysis using gpus. *BMC Bioinformatics*, 2011.

[37] J. G. Siek. and a. A. L. L.Q. Lee. *The Boost Graph Library: User Guide and Reference Manual (C++ In-Depth Series)*. Addison-Wesley Professional, 2001.

[38] G. Tan, V. C. Sreedhar, and G. R. Gao. Analysis and performance results of computing betweenness centrality on ibm cyclops64. *J. Supercomput.*, 56, 2011.

[39] G. Tan, D. Tu, and N. Sun. A parallel algorithm for computing betweenness centrality. In *ICPP*, 2009.

[40] J. Yang and Y. Chen. Fast computing betweenness centrality with virtual nodes on large sparse networks. *PLoS ONE*, 6(7), 2011.

[41] Q. Yang and S. Lonardi. A parallel algorithm for clustering protein-protein interaction networks. *Comp. Systems Bioinformatics*, 2005.