Distributed Memory Breadth-First Search Revisited: Enabling Bottom-Up Search

Scott Beamer
EECS Department
University of California
Berkeley, California

Aydın Buluç
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, California

Krste Asanović   David Patterson
EECS Department
University of California
Berkeley, California

Abstract—**Breadth-first search (BFS) is a fundamental graph primitive frequently used as a building block for many complex graph algorithms. In the worst case, the complexity of BFS is linear in the number of edges and vertices, and the conventional top-down approach always takes as much time as the worst case. A recently discovered bottom-up approach manages to cut down the complexity all the way to the number of vertices in the best case, which is typically at least an order of magnitude less than the number of edges. The bottom-up approach is not always advantageous, so it is combined with the top-down approach to make the direction-optimizing algorithm which adaptively switches from top-down to bottom-up as the frontier expands. We present a scalable distributed-memory parallelization of this challenging algorithm and show up to an order of magnitude speedups compared to an earlier purely top-down code. Our approach also uses a 2D decomposition of the graph that has previously been shown to be superior to a 1D decomposition. Using the default parameters of the Graph500 benchmark, our new algorithm achieves a performance rate of over 240 billion edges per second on 115 thousand cores of a Cray XE6, which makes it over 7× faster than a conventional top-down algorithm using the same set of optimizations and data distribution.**

I. Introduction

Breadth-first search (BFS) is a fundamental graph traversal technique that serves as a building block for many graph algorithms. Parallel graph algorithms increasingly rely on BFS as the alternative graph traversal approach, since depth-first search is inherently sequential. The fastest parallel graph algorithms often use BFS even for cases when the optimal sequential algorithm for solving the same problem relies on depth-first search, such as identifying strongly connected components [1] [2].

Given a distinguished source vertex s, BFS systematically explores the graph G to discover every vertex that is reachable from s. In the worst case, BFS has to explore all of the edges in the connected component in order to reach every vertex in the connected component. A simple level-synchronous traversal that explores all of the outgoing edges of the current frontier (the set of vertices discovered in this level) is therefore considered optimal in the worst-case analysis. This level-synchronous algorithm exposes lots of parallelism for low-diameter small-world graphs [3]. Many real-world graphs, such as those representing social interactions and brain anatomy [4], are small-world.

This level-synchronous algorithm (henceforth called *top-down*) is overly pessimistic and can be wasteful in practice, because it always does as many operations as the worst-case. Suppose that a vertex v is d hops away from the source and is reachable by x vertices, $x' \leq x$ of which are $d-1$ hops away from the source. In other words, each one of those x' vertices can potentially be the parent of v. In theory, only one of those x' incoming edges of v needs to be explored, but the top-down algorithm is unable to exploit this and does $x'-1$ extra checks. By contrast, v would quickly find a parent by checking its incoming edges if a significant number of its neighbors are reachable in $d-1$ of hops of the source. The *direction-optimizing* BFS algorithm [5] uses this intuition to significantly outperform the top-down algorithm because it reduces the number of edge examinations by integrating a *bottom-up* algorithm into its search.

Implementing this bottom-up search strategy on distributed memory poses multiple challenges. First, the bottom-up approach needs fast frontier membership tests to find a neighbor in the frontier, but the frontier is far too large to replicate in each processor's memory. Second, each vertex's search for a parent must be sequentialized in order to skip checking unnecessary edges once a parent is found. If a vertex's search for a parent is fully parallelized, there is potential the search will not terminate as soon as a parent is found, resulting in redundant work that could nullify any performance gains. We tackle the first challenge by adapting the two-dimensional graph partitioning approach that reduces the amount of the frontier that needs to be locally replicated for each processor. We tackle the second challenge by using systolic shifts that provide a good compromise between work and parallelism. In this paper we introduce a distributed memory parallel algorithm for bottom-up search.

The primary contributions of this article are:

- A novel distributed-memory parallel algorithm for the bottom-up BFS using a two-dimensional decomposition.
- Demonstration of excellent weak scaling on up to 115,000 cores of a Cray XE6, and $6.5-7.9\times$ performance increase over the top-down algorithm.
- Careful analysis of the communication costs in our new algorithm, which highlights the reduction in the amount of data communicated compared to the top-down algorithm.

The technical heart of our paper is Section IV where we present the distributed memory parallelization of our 2D bottom-up algorithm, its parallel complexity analysis, and implementation detail. To yield a fast direction-optimizing BFS implementation, our bottom-up implementation is combined with an existing high-performance top-down implementation [6]. We provide a parallel complexity analysis of the new algorithm in terms of the bandwidth and synchronization (latency) costs in Section V. Section VI gives details about our direction-optimizing approach that combines top-down and bottom-up steps. Our extensive large scale experiments on Cray XK6 and Cray XE6 machines are in Section VIII.

II. Breadth-First Search

Before delving into the details of implementing our parallel algorithm, we review sequential versions of the top-down and bottom-up BFS algorithms. The level-synchronous top-down BFS can be implemented sequentially using a queue, as shown in Algorithm 1. The algorithm outputs an implicit "breadth-first spanning tree" rooted at s by maintaining parents for each vertex. The parent of a vertex v who is d hops away from the root, can be any of the vertices that are both $d-1$ hops away from the root and have an outgoing edge to v. This algorithm's running time is proportional to $\Theta(n+m)$ where $n = |V|$ is the number of vertices and $m = |E|$ is the number of edges of a graph $G = (V, E)$. This algorithm's best-case and worst-case performance are equal, since it will always examine all of the connected-component the search started from.

The key insight the bottom-up approach leverages is that most edge examinations are unsuccessful because the endpoints have already been visited. In the conventional top-down approach, during each step, every vertex in the frontier examines all of its neighbors and claims the unvisited ones as children and adds them to the next frontier. On a low-diameter graph when the frontier is at its largest, most neighbors of the frontier have already been explored (many of which are within the frontier), but the top-down approach must check every edge in case the neighbor's only legal parent is in the frontier. The bottom-up approach passes this responsibility from the parents to the children (Algorithm 2).

During each step of the bottom-up approach, every unvisited vertex ($parent[u] = -1$) checks its neighbors to see if any of them are in the frontier. If they are, they are a valid parent and the neighbor examinations (line 6 – line 10) can end early. This early termination sequentializes the inner loop in order to get the savings from stopping as soon as a valid parent is found. In general, the bottom-up approach is only advantageous when the frontier constitutes a substantial fraction of the graph. Thus, a high-performance BFS will use the top-down approach for the beginning and end of the search and the bottom-up approach for the middle steps when the frontier is at its largest. Since the BFS for each step is done in whichever direction will require the least work, it is a *direction-optimizing* BFS.

Algorithm 1 Sequential top-down BFS algorithm

Input: $G(V, E)$, source vertex s
Output: $parent[1..n]$, where $parent[v]$ gives the parent of $v \in V$ in the BFS tree or -1 if it is unreachable from s

```
$parent[:] \leftarrow -1$, $parent[s] \leftarrow s$
$frontier \leftarrow \{s\}$, $next \leftarrow \phi$
while $frontier \neq \phi$ do
    for each $u$ in $frontier$ do
        for each neighbor $v$ of $u$ do
            if $parent[v] = -1$ then
                $next \leftarrow next \cup \{v\}$
                $parent[v] \leftarrow u$
    $frontier \leftarrow next$, $next \leftarrow \phi$
```

Algorithm 2 Sequential bottom-up BFS algorithm

Input: $G(V, E)$, source vertex s
Output: $parent[1..n]$, where $parent[v]$ gives the parent of $v \in V$ in the BFS tree or -1 if it is unreachable from s

```
1:  $parent[:] \leftarrow -1$, $parent[s] \leftarrow s$
2:  $frontier \leftarrow \{s\}$, $next \leftarrow \phi$
3:  while $frontier \neq \phi$ do
4:      for each $u$ in $V$ do
5:          if $parent[u] = -1$ then
6:              for each neighbor $v$ of $u$ do
7:                  if $v$ in $frontier$ then
8:                      $next \leftarrow next \cup \{u\}$
9:                      $parent[u] \leftarrow v$
10:                     break
11:     $frontier \leftarrow next$, $next \leftarrow \phi$
```

III. Parallel top-down BFS

Data distribution plays a critical role in parallelizing BFS on distributed-memory machines. The approach of partitioning vertices to individual processors (along with their outgoing edges) is the so-called 1D partitioning. By contrast, 2D partitioning assigns vertices to groups of processors (along with their outgoing edges), which are further assigned to members of the group. 2D checkerboard partitioning assumes the sparse adjacency matrix of the graph is partitioned as follows:

$$A = \left(\begin{array}{c|c|c} A_{1,1} & \ldots & A_{1,p_c} \\ \hline \vdots & \ddots & \vdots \\ \hline A_{p_r,1} & \ldots & A_{p_r,p_c} \end{array} \right) \quad (1)$$

Processors are logically organized in a square $p = p_r \times p_c$ mesh, indexed by their row and column indices. Submatrix A_{ij} is assigned to processor $P(i, j)$. The nonzeros in the ith row of the sparse adjacency matrix A represent the outgoing edges of the ith vertex of G, and the nonzeros in the jth column of A represent the incoming edges of the jth vertex. Our top-down algorithm actually operates on the transpose of this matrix in order to maintain the linear algebra abstraction, but we will omit the transpose and assume that the input is pre-transposed for the rest of this section.

Algorithm 3 Parallel 2D top-down BFS algorithm (adapted from the linear algebraic algorithm [6])

```
Input: $A$: graph represented by a boolean sparse adjacency
    matrix, $s$: source vertex id
Output: $\pi$: dense vector, where $\pi[v]$ is the predecessor vertex
    on the shortest path from $s$ to $v$, or $-1$ if $v$ is unreachable
 1: $\pi(:) \leftarrow -1$, $\pi(s) \leftarrow s$
 2: $f(s) \leftarrow s$                              ▷ $f$ is the current frontier
 3: for all processors $P(i,j)$ in parallel do
 4:     while $f \neq \emptyset$ do
 5:         TransposeVector($f_{ij}$)
 6:         $f_i \leftarrow$ Allgatherv($f_{ij}, P(:,j)$)
 7:         $t_i \leftarrow \emptyset$                   ▷ $t$ is candidate parents
 8:         for each $f_i(u) \neq 0$ do          ▷ $u$ is in the frontier
 9:             $adj(u) \leftarrow$ Indices($A_{ij}(:,u)$)
10:             $t_i \leftarrow t_i \cup$ Pair($adj(u), u$)
11:         $t_{ij} \leftarrow$ Alltoallv($t_i, P(i,:)$)
12:         for $(v,u)$ in $t_{ij}$ do
13:             if $\pi_{ij}(v) \neq -1$ then            ▷ Set parent if new
14:                 $\pi_{ij}(v) \leftarrow u$
15:                 $f_{ij}(v) \leftarrow v$
16:             else                   ▷ Remove if discovered before
17:                 $t_{ij} \leftarrow t_{ij} \setminus (u,v)$
```

The pseudocode for parallel top-down BFS algorithm with 2D partitioning is given in Algorithm 3 for completeness. Both f and t are implemented as sparse vectors. For distributed vectors, the syntax v_{ij} denotes the local n/p sized piece of the vector owned by the $P(i,j)$th processor (not replicated). The syntax v_i denotes the hypothetical n/p_r sized piece of the vector collectively owned by all the processors along the ith processor row $P(i,:)$ (replicated). The algorithm has four major steps:

- **Expand:** Construct the current frontier of vertices on each processor by a collective allgather step along the processor column (line 6).
- **Local discovery:** Inspect adjacencies of vertices in the current frontier and locally merge them (line 8). The operation is actually a sparse matrix-sparse vector multiplication on a special semiring where each scalar multiply returns the second operand and each scalar addition returns the minimum.
- **Fold:** Exchange newly-discovered adjacencies using a collective alltoallv step along the processor row (line 11). This step optionally merges updates from multiple processors to the same vertex using the first pair entry (the discovered vertex id) as the key.
- **Local update:** Update distances/parents for unvisited vertices (line 12). The new frontier is composed of any entries that was not removed from the candidate parents.

In contrast to the 1D case, communication in the 2D algorithm happens only along one processor dimension at a time. If *Expand* happens along one processor dimension, then *Fold* happens along the other processor dimension. Both 1D and 2D algorithms can be enhanced by in-node multithreading, resulting in one MPI process per chip instead of one MPI process per core, which will reduce the number of communicating parties. Large scale experiments of 1D versus 2D show that the 2D approach's communication costs are lower than the respective 1D approach's, with or without in-node multithreading [6]. The study also shows that in-node multithreading gives a further performance boost by decreasing network contention.

IV. Parallel Bottom-up BFS

Implementing a bottom-up BFS on a cluster with distributed memory introduces some challenges that are not present in the shared memory case. The speedup from the algorithm is dependent on fast frontier membership tests and sequentializing the inner loop. On a single compute node, the fast (constant time) membership tests for the frontier can be efficiently implemented with a bitmap that often fits in the last level of cache. Sequentializing the inner loop is trivial since the outer loop can still provide sufficient parallelism to achieve good multicore performance.

A high-performance distributed implementation must have fast frontier membership tests which requires it to be able to determine if a vertex is in the frontier without crossing the network. Holding the entire frontier in each processor's memory is clearly unscalable. Fortunately, the 2D decomposition [6] [7] greatly aids this, since for each processor, only a small subset of vertices can be the sources of a processor's incoming edges. This subset is small enough that it can fit in a processor's memory, and the frontier can be represented with a dense vector for constant time access. The dense format does not necessarily consume more memory than a sparse vector, because it can be compressed by using a bitmap and the frontier is typically a large fraction of the graph during the bottom-up steps.

Although the 2D decomposition helps with providing fast frontier checks, it complicates sequentializing the inner loop. Since all of the edges for a given vertex are spread across multiple processors, the examination of a vertex's neighbors will be done in parallel. If the inner loop is not sequentialized, the bottom-up approach's advantage of terminating the inner loop early once a parent is found will be hard to maintain. Unnecessary edges could be examined during the time it takes for the termination message to propagate across the network.

To sequentialize the inner loop of checking if neighbors are in the frontier, we propose partitioning the work temporally (Figure 1). We break down the search step into p_c sub-steps, and during each sub-step, a given vertex's edges will be examined by only one processor. During each sub-step, a processor processes $(1/p_c)$th of the vertices in that processor row. After each sub-step, it passes on the responsibility for those vertices to the processor to its right and accepts new vertices from the processor to its left. This pairwise communication sends which vertices have been completed (found parents), so the next processor knows to skip over them. This has the effect of the processor responsible for processing a vertex rotating right along the row each sub-step. When a vertex finds a valid

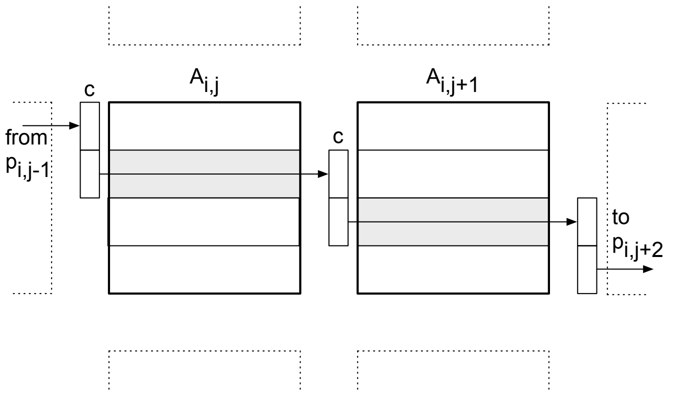

Fig. 1. Sub-step for processors $p_{i,j}$ and $p_{i,j+1}$. They initially use their segment of *completed* (c) to filter which vertices to process from the shaded region and update *completed* for each discovery. At the end of the sub-step, the *completed* segments rotate to the right. The parent updates are also transmitted at the end of the sub-step (not shown).

Operation	Type	Communications Step	64-bit Words Search
Transpose (Expand)	p2p	$O(1)$	n
Gather Frontier (Expand)	ag	$O(1)$	np_r
Sending Edges (Fold)	a2a	$O(1)$	$4m$
Total		$O(1)$	$4m + n(p_r + 1)$

TABLE I
TOP-DOWN COMMUNICATION COSTS

Operation	Type	Communications Step	64-bit Words Search
Transpose	p2p	$O(1)$	$s_b n/64$
Frontier Gather	ag	$O(1)$	$s_b n p_r/64$
Parent Updates	p2p	$O(p_c)$	$2n$
Rotate Completed	p2p	$O(p_c)$	$s_b n p_c/64$
Total		$O(p_c)$	$n(\frac{s_b(p_r+p_c+1)}{64} + 2)$

TABLE II
BOTTOM-UP COMMUNICATION COSTS

parent to become visited, its index along with its discovered parent is queued up and sent to the processor responsible for the corresponding segment of the parent array to update it.

Algorithm 4 Parallel 2D bottom-up BFS algorithm

Input: A: graph represented by a boolean sparse adjacency matrix, s: source vertex id

Output: π: dense vector, where $\pi[v]$ is the parent vertex on the shortest path from s to v, or -1 if v is unreachable

```
1:  $f(:) \leftarrow 0$, $f(s) \leftarrow 1$                ▷ bitmap for frontier
2:  $c(:) \leftarrow 0$, $c(s) \leftarrow 1$                ▷ bitmap for completed
3:  $\pi(:) \leftarrow -1$, $\pi(s) \leftarrow s$
4:  while $f(:) \neq 0$ do
5:      for all processors $P(i,j)$ in parallel do
6:          TRANSPOSEVECTOR($f_{ij}$)
7:          $f_i \leftarrow$ ALLGATHERV($f_{ij}, P(:,j)$)
8:          for s in $0 \ldots p_c - 1$ do              ▷ $p_c$ sub-steps
9:              $t \leftarrow \phi$                      ▷ $t$ holds parent updates
10:             for u in $V_{i,j+s}$ do
11:                 if $c_{ij}(u) = 0$ then              ▷ $u$ is unvisited
12:                     for each neighbor $v$ of $u$ do
13:                         if $f_i(v) = 1$ then
14:                             $t_{ij} \leftarrow t_{ij} \cup \{(u,v)\}$
15:                             $c_{ij}(u) \leftarrow 1$
16:                             break
17:             $f_{ij}(:) \leftarrow 0$
18:             $w_{ij} \leftarrow$ SENDRECV($t_{ij}, P(i,j+s), P(i,j-s)$))
19:             for $(u,v)$ in $w_{ij}$ do
20:                 $\pi_{ij}(u) \leftarrow v$
21:                 $f_{ij}(u) \leftarrow 1$
22:             $c_{ij} \leftarrow$ SENDRECV($c_{ij}, P(i,j+1), P(i,j-1)$))
```

The pseudocode for our parallel bottom-up BFS algorithm with 2D partitioning is given in Algorithm 4 for completeness. f (*frontier*) is implemented as a dense bitmap and π (*parents*) is implemented as a dense vector of integers. c (*completed*) is a dense bitmap and it represents which vertices have found parents and thus no longer need to search. The temporaries t and w are simply queues of updates represented as pairs of vertices of the form (child, parent). All processor column indices are modulo p_c (the number of processor columns). For distributed vectors, the syntax f_{ij} denotes the local n/p sized piece of the frontier owned by the $P(i,j)$th processor. Likewise, the syntax $V_{i,j}$ represents the vertices owned by the $P(i,j)$th processor. The syntax f_j denotes the hypothetical n/p_c sized piece of the frontier collectively owned by all the processors along the jth processor column $P(:,j)$. Each step of the algorithm has four major operations:

- **Gather frontier** (per step) Each processor is given the segment of the frontier corresponding to their incoming edges (lines 6 and 7).
- **Local discovery** (per sub-step) Search for parents with the information available locally (line 10 – line 16).
- **Update parents** (per sub-step) Send updates of children that found parents and process updates for own segment of *parents* (line 17 – line 21).
- **Rotate along row** (per sub-step) Send *completed* to right neighbor and receive *completed* for the next sub-step from left neighbor (line 22).

V. ANALYTIC MODEL OF COMMUNICATION

In addition to the savings in computation, the bottom-up steps will also reduce the communication volume. We summarize the communication costs in Table I and Table II, where we assume that the bottom-up approach is used for only s_b steps of the search (out of d potential steps) on a graph with m edges and n vertices, distributed on a $p_r \times p_c$ processor grid.

We first present a simple model that counts the number of 64-bit words sent and received during the entire search. We use 64-bit words as the unit because we use 64-bit words for vertex identifiers to enable us to scale to graphs with greater than 2^{32} vertices. When representing the compression provided by bitmaps, we divide the number of elements by 64. To further simplify the expressions, we assume $(p_c - 1)/(p_c) \approx 1$ and ignore transfers that send only a word (communicating sizes).

We calculate the data volume for the entire search, and assume that every vertex and every edge is part of the connected component.

The parallel 2D top-down approach transfers data for two operations: gathering the frontier (expand) and sending edges (fold). Every vertex is part of the frontier exactly once, so communicating the frontier sends n words for the transpose and np_r words for the allgather along the column. Every edge is examined once, however sending it requires sending both endpoints (two words). Since the graph is undirected, each edge is examined from both sides, which results in sending $4m$ words. In total, the number of words a search with the top-down approach sends is approximately:

$$w_t = 4m + n(p_r + 1)$$

Since the bottom-up approach is most useful when combined with the top-down approach, we assume the bottom-up approach is used for only s_b steps of the search, but it still processes the entire graph. There are three types of communication that make up the bottom-up approach: gathering the frontier, communicating completed vertices, and sending parent updates. Gathering the frontier is the same combination of a transpose and an allgather along a column like the top-down approach except a dense bitmap is used instead of a sparse vector. Since the bottom-up approach uses a dense data structure and it sends the bitmap every step it is run, it sends $s_b n(1 + p_r)/64$ words to gather the frontier. To rotate the bitmaps for *completed*, it transfers the state of every vertex once per sub-step, and since there are p_c sub-steps, an entire search sends $s_b n p_c/64$ words. Each parent update consists of a pair of words (child, parent), so in total sending the parent updates requires $2n$ words. All combined, the number of words the bottom-up approach sends is approximately:

$$w_b = n(\frac{s_b(p_r + p_c + 1)}{64} + 2)$$

To see the reduction in data volume, we take the ratio of the number of words the top-down approach sends (w_t) to the number of words the bottom-up approach will send (w_b), as shown in Equation 2. We assume our 2D partitioning is square ($p_r = p_c$) since that will send the least amount of data for both approaches. Furthermore, we assume the degree of the target graph is $k = m/n$.

$$\frac{w_t}{w_b} = \frac{p_c + 4k + 1}{s_b(2p_c + 1)/64 + 2} \tag{2}$$

For a typical value of s_b (3 or 4), by inspection the ratio will always be greater than 1; implying the bottom-up approach sends less data. Both approaches suffer when scaling up the number of processors, since it increases the communication volume. This is not unique to either approach presented, and this leads to sub-linear speedups for distributed BFS implementations. This ratio also demonstrates that the higher the degree is, the larger the gain is for the bottom-up approach relative to the top-down approach. Substituting typical values ($k = 16$ and $p_c = 128$), the bottom-up approach needs to take $s_b \approx 47.6$ steps before it sends as much data as the top-down approach. A typical s_b for the low-diameter graphs examined in this work is 3 or 4, so the bottom-up approach typically moves an order of magnitude less data. This is intuitive, since to first order, the amount of data the top-down approach sends is proportional to the number of edges, while for the bottom-up approach, it is proportional to the number of vertices.

The critical path of communication is also important to consider. The bottom-up approach sends less data, but it could be potentially bottlenecked by latency. Each step of the top-down algorithm has a constant number of communication rounds, but each step of the bottom-up approach has $\Theta(p_c)$ rounds which could be significant depending on the network latencies.

The types of communication primitives used is another important factor since primitives with more communicating parties may have higher synchronization penalties. This is summarized in Table I and Table II with the abbreviations: p2p=point-to-point, ag=allgather, and a2a=all-to-all. The communication primitives used by top-down involve more participants, as it uses: point-to-point (transpose to set up expand), allgather along columns (expand), and all-to-all along rows (fold). The bottom-up approach uses point-to-point for all communication except for the allgather along columns for gathering the frontier.

VI. Combining Bottom-up with Top-Down

The bottom-up BFS has the potential to skip many edges to accelerate the search as a whole, but it will not always be more efficient than the top-down approach. Specifically, the bottom-up approach is typically only more efficient when the frontier is large because it increases the probability of finding a valid parent. This leads to the direction-optimizing approach, a hybrid design of the top-down approach powering the search at the beginning and end, and the bottom-up approach processing the majority of the edges during only a few steps in the middle when the frontier is at or near its largest. We leverage the insight gained from prior work [5] [8] to choose when to switch between the two BFS techniques at a step (depth) granularity.

We use the number of edges in the frontier (m_f) to decide when to switch from top-down to bottom-up and the number of vertices in the frontier (n_f) to know when to switch from bottom-up back to top-down. Both the computation and the communication costs per step of the top-down approach is proportional to the number of edges in the frontier, hence the steps when the frontier is the largest consume the majority of the runtime. Conversely, the bottom-up approach is advantageous during these large steps, so using the number of edges in the frontier is appropriate to determine when the frontier is sufficiently large to switch to the bottom-up approach. Using the heuristic as well as the tuning results from prior work [5] [8], we switch from top-down to bottom-up when:

$$m_f > \frac{m}{10}$$

This can be interpreted as once the frontier encompasses at least one tenth of the edges, the bottom-up approach is likely to be advantageous. Even though the probability of finding a parent (and thus stopping early) may continue to be high as the size of the frontier ramps down in later steps, there is sufficient fixed overhead for a step of the bottom-up approach to make it worthwhile to switch back to the top-down approach. Using the results from prior work, where k is the degree, we switch back to top-down when:

$$n_f < \frac{n}{14k}$$

The degree term in the denominator ensures that higher-degree graphs switch back later to top-down since an abundance of edges will continue to help the bottom-up approach.

The switch to bottom-up uses the number of edges in the frontier while the switch back to top-down uses the number of vertices in the frontier because the apex of the number of edges in the frontier is often a step or two before the apex of the number of vertices in the frontier. For scale-free graphs, the high-degree vertices tend to be reached in the early steps since their many edges make them close to much of the graph. In the steps that follow the apex of the number of edges in the frontier, the number of vertices in the frontier becomes its largest as it contains the high-degree vertices' many low-degree neighbors. Since edges are the critical performance predictor, the number of edges in the frontier is used to guide the important switch to bottom-up. Although the number of edges in the frontier could be used to detect when to switch back top-down, it is unnecessary to compute since the number of vertices in the frontier will suffice. Although the control heuristic allows for arbitrary patterns of switches, for each search on all of the graphs studied, the frontier size has the same shape of continuously increasing and then continuously decreasing [5] [8] [9].

To compute the number of edges in the frontier, we sum the degrees of all the vertices in the frontier (volume). An undirected edge with both endpoints in the frontier will be counted twice by this method, but this is appropriate since the top-down approach will check the edge from both sides too. When loading in the graph, we calculate the degree for each vertex and store that in a dense distributed vector. Thus, to calculate the number of edges in the frontier, we take the dot product of the degree vector with the frontier.

The transition between different BFS approaches is not only a change in control, but it also requires some data structure conversion. The bottom-up approach makes use of two bitmaps (*completed* and *frontier*) that need to be generated and distributed. Generating *completed* can be simply accomplished by setting bits to one whenever the corresponding index has been visited ($parent[i] \neq -1$). Converting the frontier is similar, as it involves setting a bit in the bitmap frontier for every index in the sparse frontier used by the top-down approach. For the switch back to top-down, the bitmap frontier is converted back to a sparse list. Another challenge of combining top-down and bottom-up search is that top-down requires fast access to outgoing edges while bottom-up requires fast access to incoming edges. We keep both A^{T} and A in memory to facilitate fast switching between search directions. The alternative of transposing the matrix during execution proves to be more time-consuming than the BFS. A symmetric data structure that allows fast access to both rows (outgoing edges) and columns (incoming edges) without increasing the memory footprint would be beneficial and is considered in future work.

	Hopper	**Jaguar**
Operator	NERSC	ORNL
Supercomputer Model	Cray XE6	Cray XK6
Interconnect	Cray Gemini	Cray Gemini
Processor Model	AMD Opteron 6172	AMD Opteron 6274
Processor Architecture	Magny-Cours	Interlagos
Processor Clockrate	2.1 GHz	2.2 GHz
Sockets/node	2	1
Cores/socket	12	16
L1 Cache/socket	12×64 KB	16×16 KB
L2 Cache/socket	12×512 KB	8×2 MB
L3 Cache/socket	2×6 MB	2×8 MB
Memory/node	32 GB	32 GB

TABLE III
SYSTEM SPECIFICATIONS

VII. Experimental Settings

We run experiments on two major supercomputers: *Hopper* and *Jaguar* (Table III). We benchmark flat (1 thread per process) MPI versions of both the conventional top-down algorithm and the direction-optimizing algorithm for any given concurrency and setting. The additional benefits of using in-node multithreaded has been demonstrated before [6], and its benefits are orthogonal.

We use synthetic graphs based on the R-MAT random graph model [10], as well as the largest publicly available real-world graph that represents the structure of the Twitter social network [11], which has 61.5 million vertices and 1.47 billion edges. The Twitter graph is anonymized to respect privacy. R-MAT is a recursive graph generator that creates networks with skewed degree distributions and a very low graph diameter. R-MAT graphs make for interesting test instances because traversal load-balancing is non-trivial due to the skewed degree distribution, the lack of good graph separators, and common vertex relabeling strategies are also expected to have a minimal effect on cache performance. We use undirected graphs for all of our experiments.

We set the R-MAT parameters a, b, c, and d to $0.59, 0.19, 0.19, 0.05$ respectively and set the degree to 16 unless otherwise stated. These parameters are identical to the ones used for generating synthetic instances in the Graph500 BFS benchmark [12]. Like Graph500, to compactly describe the size of a graph, we use the *scale* variable to indicate the graph has 2^{scale} vertices.

When reporting numbers, we use the performance rate TEPS, which stands for Traversed Edges Per Second. Since the bottom-up approach may skip many edges, we compute the TEPS performance measure consistently by dividing the

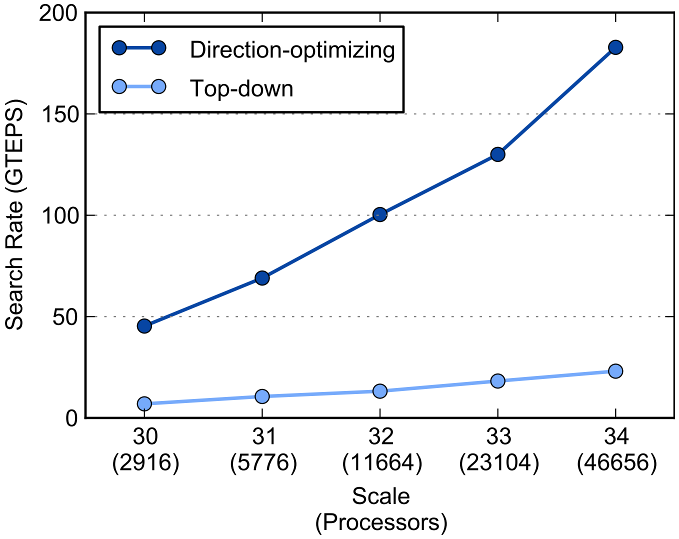

Fig. 2. R-MAT weak scaling on Jaguar

number of input edges by the runtime. During preprocessing, we prune duplicate edges and vertices with no edges from the graph. For all of our timings, we do 16 to 64 BFS runs from randomly selected distinct starting vertices and report the harmonic mean.

Both of our implementations use the Combinatorial BLAS [13] infrastructure so that their input graph data structures, processor grid topology, etc. are the same. Our baseline comparisons are against a previously published top-down implementation [6] that is further improved and tuned for Hopper. We only run experiments on processor counts that are perfect squares because the Combinatorial BLAS internally requires a square processor grid. We use Cray's MPI implementation, which is based on MPICH2, and compile our code with GCC C++ compiler version 4.6.2 with −O2 flag.

VIII. Experimental results

In this section, we first present results using the synthetically generated R-MAT graphs on Jaguar. We then provide a comparison of Hopper to Jaguar and show that our algorithm scales similarly on both architectures. Additionally, we examine the impact of graph size and degree on performance. We present results using the real-world Twitter dataset [11]. Finally, we consider the potential for the performance advantage of our algorithm to dramatically reduce the number of processors required (cost).

Weak scaling results from Jaguar demonstrate the performance improvement the direction-optimizing implementation gets from the addition of the bottom-up approach (Figure 2). At scale=30 the improvement is 6.5×, but as the graph and system size grow, the ratio of improvements extends to 7.9×.

The same implementation also has great weak scaling speedup on Hopper (Figure 3), and it reaches 243 GTEPS at scale=35. At these cluster and problem sizes, there is no slowdown in performance improvement, indicating a larger allotment on these systems could produce even faster search rates on larger problems.

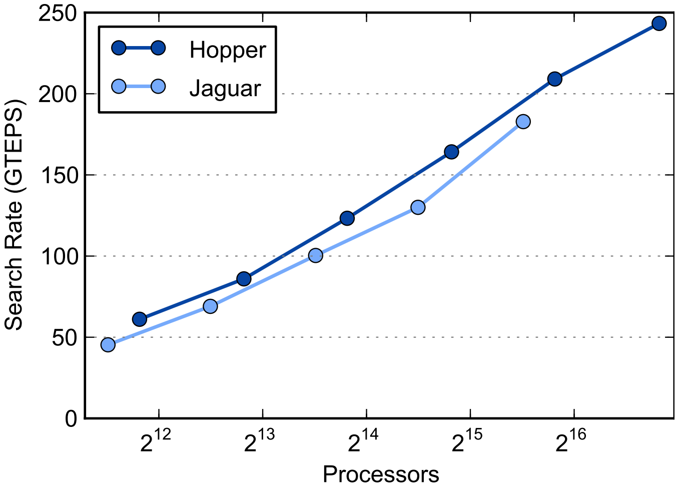

Fig. 3. R-MAT weak scaling comparison between Jaguar and Hopper. Jaguar starts with 2916 processors and Hopper starts with 3600 processors for scale=30 and the number of processors doubles for each increase in scale.

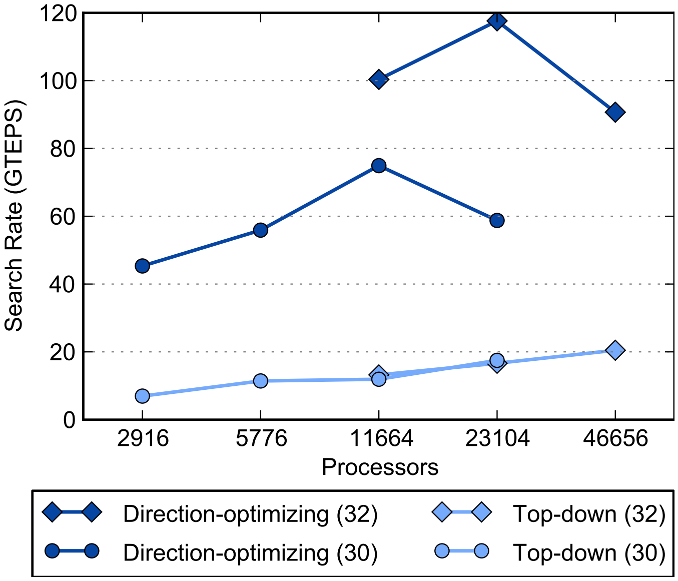

Fig. 4. R-MAT strong scaling on Jaguar for graphs of scale=30 and scale=32

Strong scaling results from Jaguar show promising speedups for the direction-optimizing approach, but it does have some slowdown for a cluster sufficiently large relative to the graph. This behavior is shown on two scales of graph (Figure 4). For both BFS approaches (top-down and bottom-up), increasing the cluster size does have the benefit of reducing the amount of computation per processor, but it comes at the cost of increased communication. The top-down approach does more computation than the bottom-up approach, so this increase in communication is offset by the decrease in computation, producing a net speedup. The bottom-up approach derives some benefit from a larger cluster, but after a certain point the communication overhead hurts its overall performance. Even

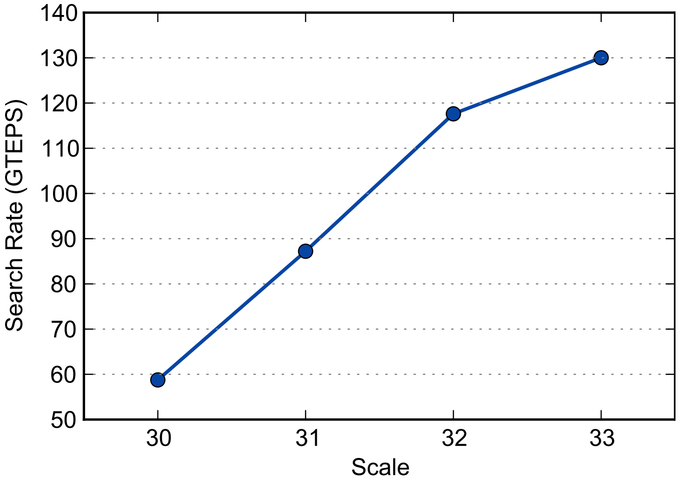

Fig. 5. R-MAT graph size scaled on Jaguar (23104 processors) with direction-optimizing BFS

though it sends less data than the top-down approach, it also has less computation to hide it with. In spite of this slowdown, the direction-optimizing approach still maintains a considerable advantage over the purely top-down implementation.

Scaling the graph size for a fixed cluster size shows the tradeoff between computation and communication for the bottom-up approach more clearly. Figure 6 presents a breakdown of where the runtime is spent on the direction-optimizing searches of Figure 5. The number of communication steps for the bottom-up approach is only proportional to the square root of the cluster size, so increasing the size of the input graph will only increase the amount of data transmitted, but not how many steps it is sent in. When the graph is too small, there is little data to compute and little data to send during each sub-step, so it is susceptible to load imbalance. The load imbalance does not stem from the 2D decomposition itself (the number of nonzeros in any A_{ij} is not more than 5% higher than the average), but it is typically due to variation in the number of parents discovered per processor. A larger graph gets better overall performance since there is more to do each sub-step so more of the time is spent doing useful work. Another cause of the load imbalance is the blocking nature of the send-recv calls of the bottom-up steps. Even when the subgraphs distributed to individual processors are perfectly load balanced, it does not translate into a run-time load balance because depending on the starting vertex, each processor might have more work to do than its row-wise neighbor.

To study the sensitivity of the direction-optimizing approach's performance to graph properties, we vary the degree while keeping the number of edges constant (Figure 7). For the top-down approach, this does not change the amount of computation since it is proportional to the number of edges and they are kept constant. For the bottom-up approach, increasing the degree actually decreases the relative amount of computation since it increases the probability of finding a parent and stopping early. As a side effect, lower degree runs scale better with increasing number of processors because they

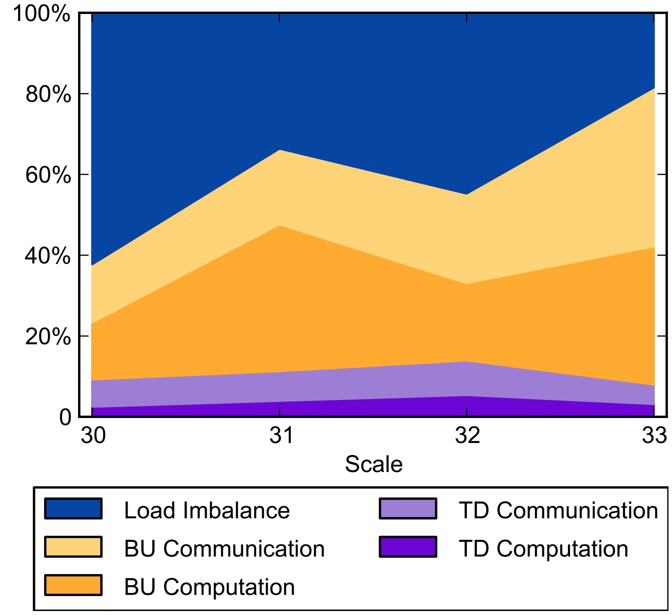

Fig. 6. Breakdown of fraction of direction-optimizing BFS runtime on Jaguar (23104 processors) on R-MAT graph size scaled (same runs as Figure 5). BU stands for bottom-up and TD for top-down.

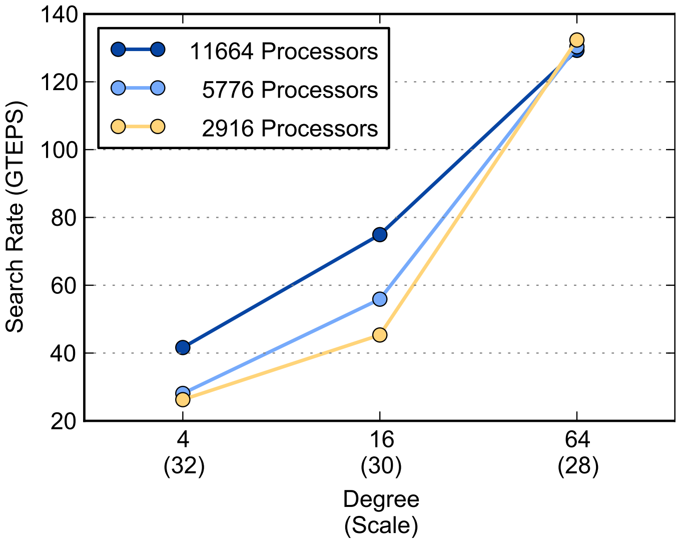

Fig. 7. R-MAT graph with 16B edges with direction-optimizing BFS on Jaguar. Degree varied by changing number of vertices (scale).

do more computation since they are finding more parents. The communication time decreases for increased degree since there are fewer vertices to transmit. This decrease in communication volume is shown by the model presented in Section V. Examining the results of Figure 7 reveals that the TEPS performance rate approximately goes up proportional to the square root of the increase in degree, i.e. the direction-optimizing algorithm runs twice as fast on graphs with degree $4k$ versus graphs with degree k when the number of edges is held constant.

As a sanity check, we run our implementations on the Twitter dataset [11] to demonstrate it works well on other scale-free low-diameter graphs (Figure 8). Because the real-world

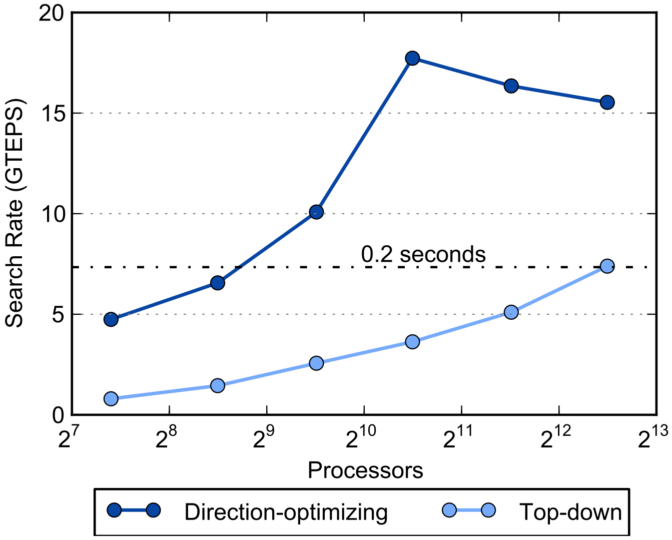

Fig. 8. Twitter dataset [11] on Jaguar

graph is much smaller than the other graphs in this study, and the direction-optimizing approach already takes only 0.08 seconds to do a full BFS on this graph with 1440 cores, its performance does not increase any further by increasing core counts. The direction-optimizing algorithm, however, provides an economic response to the inverse question "how many core are needed to traverse this data set in less than 0.2 seconds?".

Our direction-optimizing approach provides a substantial performance advantage over the top-down approach, but this translates into a far larger cost advantage. Distributed BFS algorithms have sub-linear speedups in general [6][7][14], so for the top-down approach to match the direction-optimizing approach's performance will require a super-linear number of cores. Under weak scaling (Figure 2), if the performance of the top-down approach continues on the same trajectory, it will require a scale=36 graph and approximately 185 thousand cores to match the performance of the scale=30 point of the direction-optimizing approach (a 60× increase in core count). The core count reduction for strong scaling (Figure 4) up to when the direction-optimizing approach peaks also gets dramatic reductions of similar or greater magnitude. Due to this substantial cost advantage, not using the direction-optimizing approach on a low-diameter graph could be a very expensive mistake.

IX. Related Work

Parallel BFS is a widely studied kernel, both in theory and in practice. Algorithmically, the parallelism for a graph with m edges and D diameter is limited to $O(m/D)$, so a top-down level-synchronous approach (reviewed in Section II) is most suitable for low-diameter graphs. For large diameter graphs, the span (critical path) becomes too long and frequent synchronizations limit the parallel performance. Ullman and Yannakakis [15] present an alternative algorithm that is more suitable for large diameter graphs. Their algorithm does parallel path-limited searches from a subset of vertices and combines the path-limited BFS trees by using an all-pairs shortest-paths computation.

Agarwal et al. [16] provide a shared-memory parallelization of the queue-based algorithm on multicore architectures. Leiserson and Schardl [17] provides a bag implementation to store the frontiers, relying on hyperobjects (reducers) instead of atomics. By designing an "adaptive barrier," Xia and Prasanna [18] reduce synchronization costs of BFS on multicore architectures. Hong et al. [9] use the "read-array" approach to get better memory locality and they change BFS approaches on a level granularity. The majority of these efforts focus on minimizing cache traffic and avoiding expensive atomic operations as much as possible.

On Graphical Processing Units (GPUs), Harish and Narayanan [19] give the first implementations of various graph algorithms, including BFS. Hong et al. [20] and Merrill et al. [21] both focus on reducing thread divergence within a warp.

On massively multithreaded systems, Bader and Madduri [22] introduce a fine-grained implementation on the Cray MTA-2 system using the level synchronous approach, achieving good scaling on the 40 processor MTA-2. Mizell and Maschhoff [23] improve and port this algorithm to the Cray XMT, the successor to the MTA-2.

On distributed memory, scalable implementations use a two-dimensional graph decomposition [6] [7] [14], and typically rely on bulk-synchronous computation using MPI. Other distributed memory implementations include the threaded 1D approach using active messages of Edmonds et al. [24], and the partitioned global address space (PGAS) implementation of Cong et al. [25]. Pierce et al. [26] investigate BFS implementations, among other graph algorithms, on semi-external memory.

Checconi et al. [14] provide a distributed-memory parallelization of BFS for BlueGene/P and BlueGene/Q architectures. They use a very low-latency custom communication layer instead of MPI, and specially optimize it for undirected graphs as is the case for the Graph500 benchmark. Another innovation of their work is to reduce communication by maintaining a prefix sum of assigned vertices, hence avoiding sending multiple parent updates for a single output vertex.

Satish et al. [27] reduce communication to improve performance of BFS on a cluster of commodity processors. They use a bit vector to communicate the edge traversals, which is not only more compact than a sparse list when the frontier is large, but it also squashes duplicates. They use software pipelining to overlap communication and computation and are even able to communicate information from multiple depths simultaneously. Finally, they use the servers' energy counters to estimate the power consumption of their implementation.

X. Conclusion

We give a new distributed memory parallel direction-optimizing BFS algorithm that combines the scalable two-dimensional approach with top-down and novel bottom-up search steps. We evaluate the new algorithm extensively with real and synthetically generated graphs of various sizes and degrees on tens of thousands of cores. The algorithm performs approximately 7× faster than an already scalable top-down only algorithm that uses the same data distribution and management of parallelism. Furthermore, this performance advantage allows the direction-optimizing approach to achieve the same performance as the top-down approach with more than an order of magnitude fewer processors. We believe that the algorithm is amenable to further optimizations and tuning, which we plan to pursue as future work.

Acknowledgments

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research also used resources of the Oak Ridge Leadership Computing Facility located in the Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725. The second author was supported in part by the DARPA UHPC program under contract HR0011-10-9-0008, and in part by the Director, Office of Science, U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Research was also supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung.

References

[1] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, Jun. 1972.

[2] W. McLendon III, B. Hendrickson, S. J. Plimpton, and L. Rauchwerger, "Finding strongly connected components in distributed graphs," *Journal Parallel Distributed Computing*, vol. 65, no. 8, pp. 901–910, Aug. 2005.

[3] D. J. Watts, "Networks, dynamics and the small-world phenomenon," *American Journal of Sociology*, vol. 105, no. 2, pp. 493–527, 1999.

[4] D. S. Bassett and E. Bullmore, "Small-world brain networks," *The Neuroscientist*, vol. 12, no. 6, pp. 512–23, 2006.

[5] S. Beamer, K. Asanović, and D. A. Patterson, "Direction-optimizing breadth-first search," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[6] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[7] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and Ü. V. Çatalyürek, "A scalable distributed parallel breadth-first search algorithm on BlueGene/L," in *Conference on High Performance Computing (SC)*, 2005.

[8] S. Beamer, K. Asanović, and D. A. Patterson, "Searching for a parent instead of fighting over children: A fast breadth-first search implementation for graph500," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-117, 2011.

[9] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU," *Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[10] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *International Conference on Data Mining (SDM)*. Orlando, FL: SIAM, Apr. 2004.

[11] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" *International World Wide Web Conference (WWW)*, 2010.

[12] "Graph500 benchmark." `www.graph500.org`.

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

[14] F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A. R. Choudhury, and Y. Sabharwal, "Breaking the speed and scalability barriers for graph exploration on distributed-memory machines," *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[15] J. D. Ullman and M. Yannakakis, "High-probability parallel transitive-closure algorithms," *SIAM J. Comput.*, vol. 20, no. 1, pp. 100–125, 1991.

[16] V. Agarwal, F. Petrini, D. Pasetto, and D. Bader, "Scalable graph exploration on multicore processors," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2010.

[17] C. Leiserson and T. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, Jun. 2010, pp. 303–314.

[18] Y. Xia and V. Prasanna, "Topologically adaptive parallel breadth-first search on multicore processors," in *International Conference on Parallel and Distributed Computing Systems (PDCS)*, Nov. 2009.

[19] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *International Conference on High-Performance Computing (HiPC)*, dec 2007, pp. 197–208.

[20] S. Hong, S. Kim, T. Oguntebi, and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011.

[21] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," *Principles and Practice of Parallel Programming (PPoPP)*, 2012.

[22] D. Bader and K. Madduri, "Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2," in *Proc. 35th International Conference on Parallel Processing (ICPP)*, Aug. 2006, pp. 523–530.

[23] D. Mizell and K. Maschhoff, "Early experiences with large-scale XMT systems," in *Workshop on Multithreaded Architectures and Applications (MTAAP)*, 2009.

[24] N. Edmonds, J. Willcock, T. Hoefler, and A. Lumsdaine, "Design of a large-scale hybrid-parallel graph library," in *International Conference on High Performance Computing, Student Research Symposium*, Goa, India, 2010.

[25] G. Cong, G. Almasi, and V. Saraswat, "Fast PGAS implementation of distributed graph algorithms," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2010.

[26] R. Pearce, M. Gokhale, and N. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010, pp. 1–11.

[27] N. Satish, C. Kim, J. Chhugani, and P. Dubey, "Large-scale energy-efficient graph traversal: A path to efficient data-intensive supercomputing," *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.