A Work-Efficient Parallel Breadth-First Search Algorithm (or How to Cope with the Nondeterminism of Reducers)

Charles E. Leiserson
Tao B. Schardl

MIT Computer Science and Artificial Intelligence Laboratory
32 Vassar Street
Cambridge, MA 02139

ABSTRACT

We have developed a multithreaded implementation of breadth-first search (BFS) of a sparse graph using the Cilk++ extensions to C++. Our PBFS program on a single processor runs as quickly as a standard C++ breadth-first search implementation. PBFS achieves high work-efficiency by using a novel implementation of a multiset data structure, called a "bag," in place of the FIFO queue usually employed in serial breadth-first search algorithms. For a variety of benchmark input graphs whose diameters are significantly smaller than the number of vertices — a condition met by many real-world graphs — PBFS demonstrates good speedup with the number of processing cores.

Since PBFS employs a nonconstant-time "reducer" — a "hyperobject" feature of Cilk++ — the work inherent in a PBFS execution depends nondeterministically on how the underlying work-stealing scheduler load-balances the computation. We provide a general method for analyzing nondeterministic programs that use reducers. PBFS also is nondeterministic in that it contains benign races which affect its performance but not its correctness. Fixing these races with mutual-exclusion locks slows down PBFS empirically, but it makes the algorithm amenable to analysis. In particular, we show that for a graph $G = (V, E)$ with diameter D and bounded out-degree, this data-race-free version of PBFS algorithm runs in time $O((V + E)/P + D \lg^3(V/D))$ on P processors, which means that it attains near-perfect linear speedup if $P \ll (V + E)/D \lg^3(V/D)$.

Categories and Subject Descriptors

F.2.2 [**Theory of Computation**]: Nonnumerical Algorithms and Problems—*Computations on discrete structures*; D.1.3 [**Software**]: Programming Techniques—*Concurrent programming*; G.2.2 [**Mathematics of Computing**]: Graph Theory—*Graph Algorithms*.

General Terms

Algorithms, Performance, Theory

Keywords

Breadth-first search, Cilk, graph algorithms, hyperobjects, multithreading, nondeterminism, parallel algorithms, reducers, work-stealing.

1. INTRODUCTION

Algorithms to search a graph in a breadth-first manner have been studied for over 50 years. The first breadth-first search (BFS) algorithm was discovered by Moore [26] while studying the problem of finding paths through mazes. Lee [22] independently discovered the same algorithm in the context of routing wires on circuit boards. A variety of parallel BFS algorithms have since been explored [3, 9, 21, 25, 31, 32]. Some of these parallel algorithms are *work efficient*, meaning that the total number of operations performed is the same to within a constant factor as that of a comparable serial algorithm. That constant factor, which we call the *work efficiency*, can be important in practice, but few if any papers actually measure work efficiency. In this paper, we present a parallel BFS algorithm, called PBFS, whose performance scales linearly with the number of processors and for which the work efficiency is nearly 1, as measured by comparing its performance on benchmark graphs to the classical FIFO-queue algorithm [10, Section 22.2].

Given a graph $G = (V, E)$ with vertex set $V = V(G)$ and edge set $E = E(G)$, the BFS problem is to compute for each vertex $v \in V$ the distance $v.dist$ that v lies from a distinguished *source* vertex $v_0 \in V$. We measure distance as the minimum number of edges on a path from v_0 to v in G. For simplicity in the statement of results, we shall assume that G is connected and undirected, although the algorithms we shall explore apply equally as well to unconnected graphs, digraphs, and multigraphs.

Figure 1 gives a variant of the classical serial algorithm [10, Section 22.2] for computing BFS, which uses a FIFO queue as an auxiliary data structure. The FIFO can be implemented as a simple array with two pointers to the head and tail of the items in the queue. Enqueueing an item consists of incrementing the tail pointer and storing the item into the array at the pointer location. Dequeueing consists of removing the item referenced by the head pointer and incrementing the head pointer. Since these two operations take only $\Theta(1)$ time, the running time of SERIAL-BFS is $\Theta(V + E)$. Moreover, the constants hidden by the asymptotic notation are small due to the extreme simplicity of the FIFO operations.

Although efficient, the FIFO queue Q is a major hindrance to parallelization of BFS. Parallelizing BFS while leaving the FIFO queue intact yields minimal parallelism for *sparse* graphs — those for which $|E| \approx |V|$. The reason is that if each ENQUEUE operation must be serialized, the *span*[1] of the computation — the longest

---

[1]Sometimes called *critical-path length* or *computational depth*.

```
SERIAL-BFS(G, v_0)
 1  for each vertex u ∈ V(G) − {v_0}
 2      u.dist = ∞
 3  v_0.dist = 0
 4  Q = {v_0}
 5  while Q ≠ ∅
 6      u = DEQUEUE(Q)
 7      for each v ∈ V such that (u,v) ∈ E(G)
 8          if v.dist == ∞
 9              v.dist = u.dist + 1
10              ENQUEUE(Q,v)
```

Figure 1: A standard serial breadth-first search algorithm operating on a graph G with source vertex $v_0 \in V(G)$. The algorithm employs a FIFO queue Q as an auxiliary data structure to compute for each $v \in V(G)$ its distance $v.dist$ from v_0.

serial chain of executed instructions in the computation — must have length $\Omega(V)$. Thus, a ***work-efficient*** algorithm — one that uses no more work than a comparable serial algorithm — can have ***parallelism*** — the ratio of work to span — at most $O((V+E)/V) = O(1)$ if $|E| = O(V)$.[2]
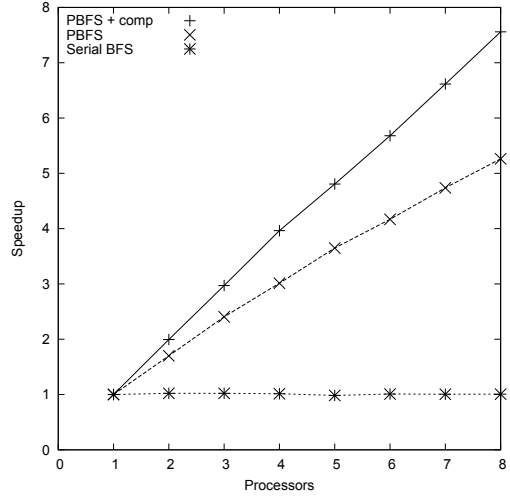
Replacing the FIFO queue with another data structure in order to parallelize BFS may compromise work efficiency, however, because FIFO's are so simple and fast. We have devised a multiset data structure called a ***bag***, however, which supports insertion essentially as fast as a FIFO, even when constant factors are considered. In addition, bags can be split and unioned efficiently.

We have implemented a parallel BFS algorithm in Cilk++ [20, 23]. Our ***PBFS*** algorithm, which employs bags instead of a FIFO, uses the "reducer hyperobject" [14] feature of Cilk++. Our implementation of PBFS runs comparably on a single processor to a good serial implementation of BFS. For a variety of benchmark graphs whose diameters are significantly smaller than the number of vertices — a common occurrence in practice — PBFS demonstrates high levels of parallelism and generally good speedup with the number of processing cores.

Figure 2 shows the typical speedup obtained for PBFS on a large benchmark graph, in this case, for a sparse matrix called Cage15 arising from DNA electrophoresis [30]. This graph has $|V| = 5,154,859$ vertices, $|E| = 99,199,551$ edges, and a diameter of $D = 50$. The code was run on an Intel Core i7 machine with eight 2.53 GHz processing cores, 12 GB of RAM, and two 8 MB L3-caches, each shared among 4 cores. As can be seen from the figure, although PBFS scales well initially, it attains a speedup of only about 5 on 8 cores, even though the parallelism in this graph is nearly 700. The figure graphs the impact of artificially increasing the ***computational intensity*** — the ratio of the number of CPU operations to the number of memory operations, suggesting that this low speedup is due to limitations of the memory system, rather than to the inherent parallelism in the algorithm.

PBFS is a nondeterministic program for two reasons. First, because the program employs a bag reducer which operates in nonconstant time, the asymptotic amount of work can vary from run to run depending upon how Cilk++'s work-stealing scheduler load-balances the computation. Second, for efficient implementation, PBFS contains a benign race condition, which can cause additional work to be generated nondeterministically. Our theoretical analysis of PBFS bounds the additional work due to the bag reducer when the race condition is resolved using mutual-exclusion locks. Theoretically, on a graph G with vertex set $V = V(G)$, edge set

---
[2]For convenience, we omit the notation for set cardinality within asymptotic notation.

Figure 2: The performance of PBFS for the Cage15 graph showing speedup curves for serial BFS, PBFS, and a variant of PBFS where the computational intensity has been artificially enhanced and the speedup normalized.

$E = E(G)$, diameter D, and bounded out-degree, this "locking" version of PBFS performs BFS in $O((V + E)/P + D\lg^3(V/D))$ time on P processors and exhibits effective parallelism $\Omega((V + E)/D\lg^3(V/D))$, which is considerable when $D \ll V$, even if the graph is sparse. Our method of analysis is general and can be applied to other programs that employ reducers. We leave it as an open question how to analyze the extra work when the race condition is left unresolved.

The remainder of this paper is divided into two parts. Part I consists of Sections 2 through 5 and describes PBFS and its empirical performance. Part II consists of Sections 6 through 9 and describes how to cope with the nondeterminism of reducers in the theoretical analysis of PBFS. Section 10 concludes by discussing thread-local storage as an alternative to reducers.

Part I — Parallel Breadth-First Search

The first half of this paper consists of Sections 2 through 5 and describes PBFS and its empirical performance. Section 2 provides background on dynamic multithreading. Section 3 describes the basic PBFS algorithm, and Section 4 describes the implementation of the bag data structure. Section 5 presents our empirical studies.

2. BACKGROUND ON DYNAMIC MULTITHREADING

This section overviews the key attributes of dynamic multithreading. The PBFS software is implemented in Cilk++ [14, 20, 23], which is a linguistic extension to C++ [28], but most of the vagaries of C++ are unnecessary for understanding the issues. Thus, we describe Cilk-like pseudocode, as is exemplified in [10, Ch. 27], which the reader should find more straightforward than real code to understand and which can be translated easily to Cilk++.

Multithreaded pseudocode

The linguistic model for multithreaded pseudocode in [10, Ch. 27] follows MIT Cilk [15, 29] and Cilk++ [20, 23]. It augments ordinary serial pseudocode with three keywords — **spawn**, **sync**, and **parallel** — of which **spawn** and **sync** are the more basic.

Parallel work is created when the keyword **spawn** precedes the invocation of a function. The semantics of spawning differ from a

```
1   x = 10          1   x = 10          1   x = 10
2   x++             2   x++             2   x++
3   x += 3          3   x += 3          3   x += 3
4   x += -2         4   x += -2             x' = 0
5   x += 6          5   x += 6          4   x' += -2
6   x--                 x' = 0          5   x' += 6
7   x += 4          6   x'--            6   x'--
8   x += 3          7   x' += 4             x'' = 0
9   x++             8   x' += 3         7   x'' += 4
10  x += -9         9   x'++            8   x'' += 3
                    10  x' += -9        9   x''++
                        x += x'         10  x'' += -9
                                            x += x'
                                            x += x''

       (a)                (b)                (c)
```

Figure 3: The intuition behind reducers. (a) A series of additive updates performed on a variable x. (b) The same series of additive updates split between two "views" x and x'. The two update sequences can execute in parallel and are combined at the end. (c) Another valid splitting of these updates among the views x, x', and x''.

C or C++ function call only in that the parent ***continuation*** — the code that immediately follows the spawn — may execute in parallel with the child, instead of waiting for the child to complete, as is normally done for a function call. A function cannot safely use the values returned by its children until it executes a **sync** statement, which suspends the function until all of its spawned children return. Every function syncs implicitly before it returns, precluding orphaning. Together, **spawn** and **sync** allow programs containing fork-join parallelism to be expressed succinctly. The scheduler in the runtime system takes the responsibility of scheduling the spawned functions on the individual processor cores of the multicore computer and synchronizing their returns according to the fork-join logic provided by the **spawn** and **sync** keywords.

Loops can be parallelized by preceding an ordinary **for** with the keyword **parallel**, which indicates that all iterations of the loop may operate in parallel. Parallel loops do not require additional runtime support, but can be implemented by parallel divide-and-conquer recursion using **spawn** and **sync**.

Cilk++ provides a novel linguistic construct, called a ***reducer hyperobject*** [14], which allows concurrent updates to a shared variable or data structure to occur simultaneously without contention. A reducer is defined in terms of a binary associative **REDUCE** operator, such as sum, list concatenation, logical AND, etc. Updates to the hyperobject are accumulated in local ***views***, which the Cilk++ runtime system combines automatically with "up-calls" to REDUCE when subcomputations join. As we shall see in Section 3, PBFS uses a reducer called a "bag," which implements an unordered set and supports fast unioning as its REDUCE operator.

Figure 3 illustrates the basic idea of a reducer. The example involves a series of additive updates to a variable x. When the code in Figure 3(a) is executed serially, the resulting value is $x = 16$. Figure 3(b) shows the same series of updates split between two "views" x and x' of the variable. These two views may be evaluated independently in parallel with an additional step to ***reduce*** the results at the end, as shown in Figure 3(b). As long as the values for the views x and x' are not inspected in the middle of the computation, the associativity of addition guarantees that the final result is deterministically $x = 16$. This series of updates could be split anywhere else along the way and yield the same final result, as demonstrated in Figure 3(c), where the computation is split across three views x, x', and x''. To encapsulate nondeterminism in this way, each of the views must be reduced with an associative RE-

---

```
PBFS(G, v_0)
1    parallel for each vertex v ∈ V(G) − {v_0}
2        v.dist = ∞
3    v_0.dist = 0
4    d = 0
5    V_0 = BAG-CREATE()
6    BAG-INSERT(V_0, v_0)
7    while ¬BAG-IS-EMPTY(V_d)
8        V_{d+1} = new reducer BAG-CREATE()
9        PROCESS-LAYER(revert V_d, V_{d+1}, d)
10       d = d + 1

PROCESS-LAYER(in-bag, out-bag, d)
11   if BAG-SIZE(in-bag) < GRAINSIZE
12       for each u ∈ in-bag
13           parallel for each v ∈ Adj[u]
14               if v.dist == ∞
15                   v.dist = d + 1          // benign race
16                   BAG-INSERT(out-bag, v)
17       return
18   new-bag = BAG-SPLIT(in-bag)
19   spawn PROCESS-LAYER(new-bag, out-bag, d)
20   PROCESS-LAYER(in-bag, out-bag, d)
21   sync
```

Figure 4: The PBFS algorithm operating on a graph G with source vertex $v_0 \in V(G)$. PBFS uses the recursive parallel subroutine PROCESS-LAYER to process each layer. It contains a benign race in line 15.

DUCE operator (addition for this example) and intermediate views must be initialized to the identity for REDUCE (0 for this example).

Cilk++'s reducer mechanism supports this kind of decomposition of update sequences automatically without requiring the programmer to manually create various views. When a function spawns, the spawned child inherits the parent's view of the hyperobject. If the child returns before the continuation executes, the child can return the view and the chain of updates can continue. If the continuation begins executing before the child returns, however, the continuation receives a new view initialized to the identity for the associative REDUCE operator. Sometime at or before the **sync** that joins the spawned child with its parent, the two views are combined with REDUCE. If REDUCE is indeed associative, the result is the same as if all the updates had occurred serially. Indeed, if the program is run on one processor, the entire computation updates only a single view without ever invoking the REDUCE operator, in which case the behavior is virtually identical to a serial execution that uses an ordinary object instead of a hyperobject. We shall formalize reducers in Section 7.

3. THE PBFS ALGORITHM

PBFS uses ***layer synchronization*** [3, 32] to parallelize breadth-first search of an input graph G. Let $v_0 \in V(G)$ be the source vertex, and define ***layer*** d to be the set $V_d \subseteq V(G)$ of vertices at distance d from v_0. Thus, we have $V_0 = \{v_0\}$. Each iteration processes layer d by checking all the neighbors of vertices in V_d for those that should be added to V_{d+1}.

PBFS implements layers using an unordered-set data structure, called a ***bag***, which provides the following operations:

- bag = BAG-CREATE(): Create a new empty bag.
- BAG-INSERT(bag, x): Insert element x into bag.
- BAG-UNION(bag_1, bag_2): Move all the elements from bag_2 to bag_1, and destroy bag_2.
- bag_2 = BAG-SPLIT(bag_1): Remove half (to within some

```
15.1    set = FALSE
15.2    if TRY-LOCK(v)
15.3        if v.dist == ∞
15.4            v.dist = d + 1
15.5            set = TRUE
15.6        RELEASE-LOCK(v)
15.7    if set
15.8        BAG-INSERT(out-bag, v)
```

Figure 5: Modification to the PBFS algorithm to resolve the benign race.

constant amount GRAINSIZE of granularity) of the elements from bag_1, and put them into a new bag bag_2.

As Section 4 shows, BAG-CREATE operates in $O(1)$ time, and BAG-INSERT operates in $O(1)$ amortized time. Both BAG-UNION and BAG-SPLIT operate in $O(\lg n)$ time on bags with n elements.

Let us walk through the pseudocode for PBFS, which is shown in Figure 4. For the moment, ignore the **revert** and **reducer** keywords in lines 8 and 9.

After initialization, PBFS begins the **while** loop in line 7 which iteratively calls the auxiliary function PROCESS-LAYER to process layer $d = 0, 1, \ldots, D$, where D is the diameter of the input graph G. To process $V_d = in\text{-}bag$, PROCESS-LAYER uses parallel divide-and-conquer, producing $V_{d+1} = out\text{-}bag$. For the recursive case, line 18 splits $in\text{-}bag$, removing half its elements and placing them in $new\text{-}bag$. The two halves are processed recursively in parallel in lines 19–20.
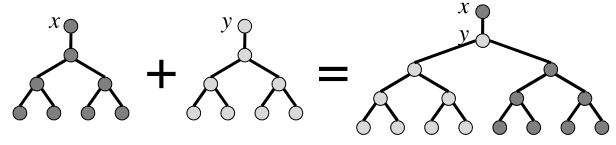
This recursive decomposition continues until $in\text{-}bag$ has fewer than GRAINSIZE elements, as tested for in line 11. Each vertex u in $in\text{-}bag$ is extracted in line 12, and line 13 examines each of its edges (u, v) in parallel. If v has not yet been visited — $v.dist$ is infinite (line 14) — then line 15 sets $v.dist = d + 1$ and line 16 inserts v into the level-$(d+1)$ bag. As an implementation detail, the destructive nature of the BAG-SPLIT routine makes it particularly convenient to maintain only two bags at a time, ignoring additional views set up by the runtime system.

This description skirts over two subtleties that require discussion, both involving races.

First, the update of $v.dist$ in line 15 creates a race, since two vertices u and u' may both be examining vertex v at the same time. They both check whether $v.dist$ is infinite in line 14, discover that it is, and both proceed to update $v.dist$. Fortunately, this race is benign, meaning that it does not affect the correctness of the algorithm. Both u and u' set $v.dist$ to the same value, and hence no inconsistency arises from both updating the location at the same time. They both go on to insert v into bag $V_{d+1} = out\text{-}bag$ in line 16, which could induce another race. Putting that issue aside for the moment, notice that inserting multiple copies of v into V_{d+1} does not affect correctness, only performance for the extra work it will take when processing layer $d + 1$, because v will be encountered multiple times. As we shall see in Section 5, the amount of extra work is small, because the race is rarely actualized.

Second, a race in line 16 occurs due to parallel insertions of vertices into $V_{d+1} = out\text{-}bag$. We employ the reducer functionality to avoid the race by making V_{d+1} a bag reducer, where BAG-UNION is the associative operation required by the reducer mechanism. The identity for BAG-UNION — an empty bag — is created by BAG-CREATE. In the common case, line 16 simply inserts v into the local view, which, as we shall see in Section 4, is as efficient as pushing v onto a FIFO, as is done by serial BFS.

Unfortunately, we are not able to analyze PBFS due to unstructured nondeterminism created by the benign race, but we can ana-

Figure 6: Two pennants, each of size 2^k, can be unioned in constant time to form a pennant of size 2^{k+1}.

lyze a version where the race is resolved using a mutual-exclusion lock. The locking version involves replacing lines 15 and 16 with the code in Figure 5. In the code, the call TRY-LOCK(v) in line 15.2 attempts to acquire a lock on the vertex v. If it is successful, we proceed to execute lines 15.3–15.6. Otherwise, we can abandon the attempt, because we know that some other processor has succeeded, which then sets $v.dist = d + 1$ regardless. Thus, there is no contention on v's lock, because no processor ever waits for another, and processing an edge (u, v) always takes constant time. The apparently redundant lines 14 and 15.3 avoid the overhead of lock acquisition when $v.dist$ has already been set.

4. THE BAG DATA STRUCTURE

This section describes the bag data structure for implementing a dynamic unordered set. We first describe an auxiliary data structure called a "pennant." We then show how bags can be implemented using pennants, and we provide algorithms for BAG-CREATE, BAG-INSERT, BAG-UNION, and BAG-SPLIT. Finally, we discuss some optimizations of this structure that PBFS employs.

Pennants

A *pennant* is a tree of 2^k nodes, where k is a nonnegative integer. Each node x in this tree contains two pointers $x.left$ and $x.right$ to its children. The root of the tree has only a left child, which is a complete binary tree of the remaining elements.

Two pennants x and y of size 2^k can be combined to form a pennant of size 2^{k+1} in $O(1)$ time using the following PENNANT-UNION function, which is illustrated in Figure 6.

```
PENNANT-UNION(x, y)
1    y.right = x.left
2    x.left = y
3    return x
```

The function PENNANT-SPLIT performs the inverse operation of PENNANT-UNION in $O(1)$ time. We assume that the input pennant contains at least 2 elements.

```
PENNANT-SPLIT(x)
1    y = x.left
2    x.left = y.right
3    y.right = NULL
4    return y
```

Each of the pennants x and y now contains half the elements.

Bags

A *bag* is a collection of pennants, no two of which have the same size. PBFS represents a bag S using a fixed-size array $S[0..r]$, called the *backbone*, where 2^{r+1} exceeds the maximum number of elements ever stored in a bag. Each entry $S[k]$ in the backbone contains either a null pointer or a pointer to a pennant of size 2^k. Figure 7 illustrates a bag containing 23 elements. The function BAG-CREATE allocates space for a fixed-size backbone of null pointers, which takes $\Theta(r)$ time. This bound can be improved to $O(1)$ by keeping track of the largest nonempty index in the backbone.
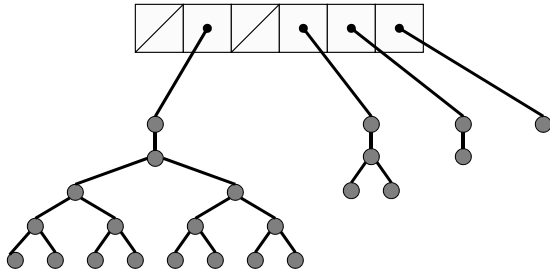
Figure 7: A bag with $23 = 010111_2$ elements.

The BAG-INSERT function employs an algorithm similar to that of incrementing a binary counter. To implement BAG-INSERT, we first package the given element as a pennant x of size 1. We then insert x into bag S using the following method.

BAG-INSERT(S,x)

```
1   k = 0
2   while S[k] ≠ NULL
3       x = PENNANT-UNION(S[k],x)
4       S[k++] = NULL
5   S[k] = x
```

The analysis of BAG-INSERT mirrors the analysis for incrementing a binary counter [10, Ch. 17]. Since every PENNANT-UNION operation takes constant time, BAG-INSERT takes $O(1)$ amortized time and $O(\lg n)$ worst-case time to insert into a bag of n elements. The BAG-UNION function uses an algorithm similar to ripple-carry addition of two binary counters. To implement BAG-UNION, we first examine the process of unioning three pennants into two pennants, which operates like a full adder. Given three pennants x, y, and z, where each either has size 2^k or is empty, we can merge them to produce a pair of pennants (s,c), where s has size 2^k or is empty, and c has size 2^{k+1} or is empty. The following table details the function $FA(x,y,z)$ in which (s,c) is computed from (x,y,z), where 0 means that the designated pennant is empty, and 1 means that it has size 2^k:

x	y	z	s	c
0	0	0	NULL	NULL
1	0	0	x	NULL
0	1	0	y	NULL
0	0	1	z	NULL
1	1	0	NULL	PENNANT-UNION(x,y)
1	0	1	NULL	PENNANT-UNION(x,z)
0	1	1	NULL	PENNANT-UNION(y,z)
1	1	1	x	PENNANT-UNION(y,z)

With this full-adder function in hand, BAG-UNION can be implemented as follows:

BAG-UNION(S_1,S_2)

```
1   y = NULL    // The "carry" bit.
2   for k = 0 to r
3       (S1[k],y) = FA(S1[k],S2[k],y)
```

Because every PENNANT-UNION operation takes constant time, computing the value of $FA(x,y,z)$ also takes constant time. To compute all entries in the backbone of the resulting bag takes $\Theta(r)$ time. This algorithm can be improved to $\Theta(\lg n)$, where n is the number of elements in the smaller of the two bags, by maintaining the largest nonempty index of the backbone of each bag and unioning the bag with the smaller such index into the one with the larger. The BAG-SPLIT function operates like an arithmetic right shift:

BAG-SPLIT(S_1)

```
1    S2 = BAG-CREATE()
2    y = S1[0]
3    S1[0] = NULL
4    for k = 1 to r
5        if S1[k] ≠ NULL
6            S2[k−1] = PENNANT-SPLIT(S1[k])
7            S1[k−1] = S1[k]
8            S1[k] = NULL
9    if y ≠ NULL
10       BAG-INSERT(S1,y)
11   return S2
```

Because PENNANT-SPLIT takes constant time, each loop iteration in BAG-SPLIT takes constant time. Consequently, the asymptotic runtime of BAG-SPLIT is $O(r)$. This algorithm can be improved to $\Theta(\lg n)$, where n is the number of elements in the input bag, by maintaining the largest nonempty index of the backbone of each bag and iterating only up to this index.

Optimization

To improve the constant in the performance of BAG-INSERT, we made some simple but important modifications to pennants and bags, which do not affect the asymptotic behavior of the algorithm. First, in addition to its two pointers, every pennant node in the bag stores a constant-size array of GRAINSIZE elements, all of which are guaranteed to be valid, rather than just a single element. Our PBFS software uses the value GRAINSIZE $= 128$. Second, in addition to the backbone, the bag itself maintains an additional pennant node of size GRAINSIZE called the ***hopper***, which it fills gradually. The impact of these modifications on the bag operations is as follows.

First, BAG-CREATE must allocate additional space for the hopper. This overhead is small and is done only once per bag.

Second, BAG-INSERT first attempts to insert the element into the hopper. If the hopper is full, then it inserts the hopper into the backbone of the data structure and allocates a new hopper into which it inserts the element. This optimization does not change the asymptotic runtime analysis of BAG-INSERT, but the code runs much faster. In the common case, BAG-INSERT simply inserts the element into the hopper with code nearly identical to inserting an element into a FIFO. Only once in every GRAINSIZE insertions does a BAG-INSERT trigger the insertion of the now full hopper into the backbone of the data structure.

Third, when unioning two bags S_1 and S_2, BAG-UNION first determines which bag has the less full hopper. Assuming that it is S_1, the modified implementation copies the elements of S_1's hopper into S_2's hopper until it is full or S_1's hopper runs out of elements. If it runs out of elements in S_1 to copy, BAG-UNION proceeds to merge the two bags as usual and uses S_2's hopper as the hopper for the resulting bag. If it fills S_2's hopper, however, line 1 of BAG-UNION sets y to S_2's hopper, and S_1's hopper, now containing fewer elements, forms the hopper for the resulting bag. Afterward, BAG-UNION proceeds as usual.

Finally, rather than storing $S_1[0]$ into y in line 2 of BAG-SPLIT for later insertion, BAG-SPLIT sets the hopper of S_2 to be the pennant node in $S_1[0]$ before proceeding as usual.

5. EXPERIMENTAL RESULTS

We implemented optimized versions of both the PBFS algorithm in Cilk++ and a FIFO-based serial BFS algorithm in C++. This section compares their performance on a suite of benchmark graphs. Figure 8 summarizes the results.

Implementation and Testing

Our implementation of PBFS differs from the abstract algorithm in some notable ways. First, our implementation of PBFS does not use locks to resolve the benign races described in Section 3. Second, our implementation of PBFS does not use the Bag-Split routine described in Section 4. Instead, our implementation uses a "lop" operation to traverse the bag. It repeatedly divides the bag into two approximately equal halves by lopping off the most significant pennant from the bag. After each lop, the removed pennant is traversed using a standard parallel tree walk. Third, our implementation assumes that all vertices have bounded out-degree, and indeed most of the vertices in our benchmark graphs have relatively small degree. Finally, our implementation of PBFS sets $\text{GRAINSIZE} = 128$, which seems to perform well in practice. The FIFO-based serial BFS uses an array and two pointers to implement the FIFO queue in the simplest way possible. This array was sized to the number of vertices in the input graph.

These implementations were tested on eight benchmark graphs, as shown in Figure 8. Kkt_power, Cage14, Cage15, Freescale1, Wikipedia (as of February 6, 2007), and Nlpkkt160 are all from the University of Florida sparse-matrix collection [11]. Grid3D200 is a 7-point finite difference mesh generated using the Matlab Mesh Partitioning and Graph Separator Toolbox [16]. The RMat23 matrix [24], which models scale-free graphs, was generated by using repeated Kronecker products [2]. Parameters $A = 0.7, B = C = D = 0.1$ for RMat23 were chosen in order to generate skewed matrices. We stored these graphs in a compressed-sparse-rows (CSR) format in main memory for our empirical tests.

Results

We ran our tests on an Intel Core i7 quad-core machine with a total of eight 2.53-GHz processing cores (hyperthreading disabled), 12 GB of DRAM, two 8-MB L3-caches each shared between 4 cores, and private L2- and L1-caches with 256 KB and 32 KB, respectively. Figure 8 presents the performance of PBFS on eight different benchmark graphs. (The parallelism was computed using the Cilkview [19] tool and does not take into account effects from reducers.) As can be seen in Figure 8, PBFS performs well on these benchmark graphs. For five of the eight benchmark graphs, PBFS is as fast or faster than serial BFS. Moreover, on the remaining three benchmarks, PBFS is at most 15% slower than serial BFS.

Figure 8 shows that PBFS runs faster than a FIFO-based serial BFS on several benchmark graphs. This performance advantage may be due to how PBFS uses memory. Whereas the serial BFS performs a single linear scan through an array as it processes its queue, PBFS is constantly allocating and deallocating fixed-size chunks of memory for the bag. Because these chunks do not change in size from allocation to allocation, the memory manager incurs little work to perform these allocations. Perhaps more importantly, PBFS can reuse previously allocated chunks frequently, making it more cache-friendly. This improvement due to memory reuse is also apparent in some serial BFS implementations that use two queues instead of one.

Although PBFS generally performs well on these benchmarks, we explored why it was only attaining a speedup of 5 or 6 on 8 processor cores. Inadequate parallelism is not the answer, as most of the benchmarks have parallelism over 100. Our studies indicate that the multicore processor's memory system may be hurting performance in two ways.

First, the memory bandwidth of the system seems to limit performance for several of these graphs. For Wikipedia and Cage14, when we run 8 independent instances of PBFS serially on the 8 processing cores of our machine simultaneously, the total runtime is at

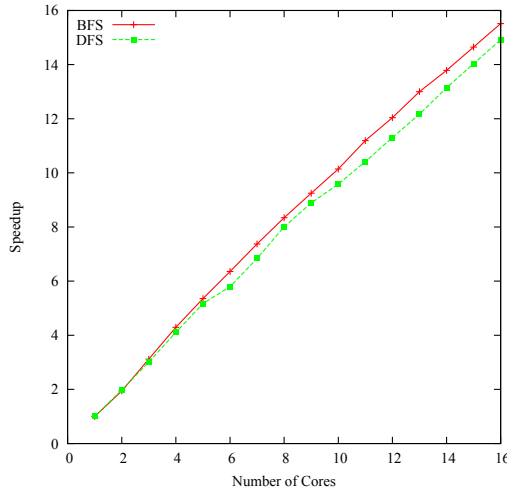| Name *Description*	*Spy Plot*	$\|V\|$ $\|E\|$ D	*Work* *Span* *Parallelism*	Serial-BFS T_1 PBFS T_1 PBFS T_1/T_8
Kkt_power Optimal power flow, nonlinear opt.		2.05M 12.76M 31	241M 2.3M 103.85	0.504 0.359 5.983
Freescale1 Circuit simulation		3.43M 17.1M 128	349M 2.3M 152.72	0.285 0.327 5.190
Cage14 DNA electrophoresis		1.51M 27.1M 43	390M 1.6M 245.70	0.262 0.283 5.340
Wikipedia Links between Wikipedia pages		2.4M 41.9M 460	606M 3.4M 178.73	0.914 0.721 6.381
Grid3D200 3D 7-point finite-diff mesh		8M 55.8M 598	1,009M 12.7M 79.27	1.544 1.094 4.862
RMat23 Scale-free graph model		2.3M 77.9M 8	1,050M 11.3M 93.22	1.100 0.936 6.500
Cage15 DNA electrophoresis		5.15M 99.2M 50	1,410M 2.1M 674.65	1.065 1.142 5.263
Nlpkkt160 Nonlinear optimization		8.35M 225.4M 163	3,060M 9.2M 331.45	1.269 1.448 5.983

Figure 8: Performance results for breadth-first search. The vertex and edge counts listed correspond to the number of vertices and edges evaluated by Serial-BFS. The work and span are measured in instructions. All runtimes are measured in seconds.

least 20% worse than the expected $8T_1$. This experiment suggests that the system's available memory bandwidth limits the performance of the parallel execution of PBFS.

Second, for several of these graphs, it appears that contention from true and false sharing on the distance array constrains the speedups. Placing each location in the distance array on a different cache line tends to increase the speedups somewhat, although it slows down overall performance due to the loss of spatial locality. We attempted to modify PBFS to mitigate contention by randomly permuting or rotating each adjacency list. Although these approaches improve speedups, they slow down overall performance due to loss of locality. Thus, despite its somewhat lower relative speedup numbers, the unadulterated PBFS seems to yield the best overall performance.

PBFS obtains good performance despite the benign race which induces redundant work. On none of these benchmarks does PBFS examine more than 1% of the vertices and edges redundantly. Using a mutex lock on each vertex to resolve the benign race costs a substantial overhead in performance, typically slowing down PBFS by more than a factor of 2.

Yuxiong He [18], formerly of Cilk Arts and Intel Corporation, used PBFS to parallelize the Murphi model-checking tool [12]. Murphi is a popular tool for verifying finite-state machines and is widely used in cache-coherence algorithms and protocol design, link-level protocol design, executable memory-model analysis, and analysis of cryptographic and security-related protocols. As can be seen in Figure 9, a parallel Murphi using PBFS scales well, even

Figure 9: Multicore Murphi application speedup on a 16-core AMD processor [18]. Even though the DFS implementation uses a parallel depth-first search for which Cilk++ is particularly well suited, the BFS implementation, which uses the PBFS library, outperforms it.

outperforming a version based on parallel depth-first search and attaining the relatively large speedup of 15.5 times on 16 cores.

Part II — Nondeterminism of Reducers

The second half of this paper consists of Sections 6 through 9 and describes how to cope with the nondeterminism of reducers in the theoretical analysis of PBFS. Section 6 provides background on the theory of dynamic multithreading. Section 7 gives a formal model for reducer behavior, Section 8 develops a theory for analyzing programs that use reducers, and Section 9 employs this theory to analyze the performance of PBFS.
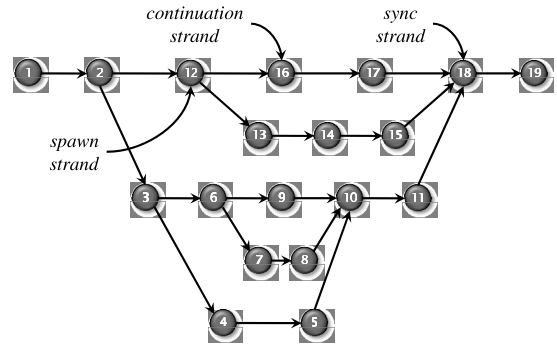
6. BACKGROUND ON THE DAG MODEL

This section overviews the theoretical model of Cilk-like parallel computation. We explain how a multithreaded program execution can be modeled theoretically as a dag using the framework of Blumofe and Leiserson [7], and we overview assumptions about the runtime environment. We define deterministic and nondeterministic computations. Section 7 will describe how reducer hyperobjects fit into this theoretical framework.

The dag model

We shall adopt the dag model for multithreading similar to the one introduced by Blumofe and Leiserson [7]. This model was designed to model the execution of spawns and syncs. We shall extend it in Section 7 to deal with reducers.

The dag model views the executed computation resulting from the running of a multithreaded program[3] as a *dag (directed acyclic graph)* A, where the vertex set consists of *strands* — sequences of serially executed instructions containing no parallel control — and the edge set represents parallel-control dependencies between strands. We shall use A to denote both the dag and the set of strands in the dag. Figure 10 illustrates such a dag, which can be viewed as a parallel program "trace," in that it involves executed instructions, as opposed to source instructions. A strand can be as small as a single instruction, or it can represent a longer computation. We shall assume that strands respect function boundaries, meaning

[3]When we refer to the running of a program, we shall generally assume that we mean "on a given input."

Figure 10: A dag representation of a multithreaded execution. Each vertex represents a strand, and edges represent parallel-control dependencies between strands.

that calling or spawning a function terminates a strand, as does returning from a function. Thus, each strand belongs to exactly one function instantiation. A strand that has out-degree 2 is a *spawn strand*, and a strand that resumes the caller after a spawn is called a *continuation strand*. A strand that has in-degree at least 2 is a *sync strand*.

Generally, we shall dice a chain of serially executed instructions into strands in a manner that is convenient for the computation we are modeling. The *length* of a strand is the time it takes for a processor to execute all its instructions. For simplicity, we shall assume that programs execute on an *ideal parallel computer*, where each instruction takes unit time to execute, there is ample memory bandwidth, there are no cache effects, etc.

Determinacy

We say that a dynamic multithreaded program is *deterministic* (on a given input) if every memory location is updated with the same sequence of values in every execution. Otherwise, the program is *nondeterministic*. A deterministic program always behaves the same, no matter how the program is scheduled. Two different memory locations may be updated in different orders, but each location always sees the same sequence of updates. Whereas a nondeterministic program may produce different dags, i.e., behave differently, a deterministic program always produces the same dag.

Work and span

The dag model admits two natural measures of performance which can be used to provide important bounds [6, 8, 13, 17] on performance and speedup. The *work* of a dag A, denoted by $\text{Work}(A)$, is the sum of the lengths of all the strands in the dag. Assuming for simplicity that it takes unit time to execute a strand, the work for the example dag in Figure 10 is 19. The *span*[4] of A, denoted by $\text{Span}(A)$, is the length of the longest path in the dag. Assuming unit-time strands, the span of the dag in Figure 10 is 10, which is realized by the path $\langle 1,2,3,6,7,8,10,11,18,19 \rangle$. Work/span analysis is outlined in tutorial fashion in [10, Ch. 27] and [23].

Suppose that a program produces a dag A in time T_P when run on P processors of an ideal parallel computer. We have the following two lower bounds on the execution time T_P:

$$T_p \geq \text{Work}(A)/P , \qquad (1)$$
$$T_P \geq \text{Span}(A) . \qquad (2)$$

Inequality (2), which is called the *Work Law*, holds in this simple performance model, because each processor executes at most 1 instruction per unit time, and hence P processors can execute at most

[4]The literature also uses the terms *depth* [4] and *critical-path length* [5].

P instructions per unit time. Inequality (2), called the ***Span Law***, holds because no execution that respects the partial order of the dag can execute faster than the longest serial chain of instructions.

We define the ***speedup*** of a program as T_1/T_P — how much faster the P-processor execution is than the serial execution. Since for deterministic programs, all executions produce the same dag A, we have that $T_1 = \text{Work}(A)$, and $T_\infty = \text{Span}(A)$ (assuming no overhead for scheduling). Rewriting the Work Law, we obtain $T_1/T_P \le P$, which is to say that the speedup on P processors can be at most P. If the application obtains speedup P, which is the best we can do in our model, we say that the application exhibits ***linear speedup***. If the application obtains speedup greater than P (which cannot happen in our model due to the Work Law, but can happen in models that incorporate caching and other processor effects), we say that the application exhibits ***superlinear speedup***.

The ***parallelism*** of the dag is defined as $\text{Work}(A)/\text{Span}(A)$. For a deterministic computation, the parallelism is therefore T_1/T_∞. The parallelism represents the maximum possible speedup on any number of processors, which follows from the Span Law, because $T_1/T_P \le T_1/\text{Span}(A) = \text{Work}(A)/\text{Span}(A)$. For example, the parallelism of the dag in Figure 10 is $19/10 = 1.9$, which means that any advantage gained by executing it with more than 2 processors is marginal, since the additional processors will surely be starved for work.

Scheduling

A randomized "work-stealing" scheduler [1, 7], such as is provided by MIT Cilk and Cilk++, operates as follows. When the runtime system starts up, it allocates as many operating-system threads, called ***workers***, as there are processors (although the programmer can override this default decision). Each worker's stack operates like a ***deque***, or double-ended queue. When a subroutine is spawned, the subroutine's activation frame containing its local variables is pushed onto the bottom of the deque. When it returns, the frame is popped off the bottom. Thus, in the common case, the parallel code operates just like serial code and imposes little overhead. When a worker runs out of work, however, it becomes a ***thief*** and "steals" the top frame from another ***victim*** worker's deque. In general, the worker operates on the bottom of the deque, and thieves steal from the top. This strategy has the great advantage that all communication and synchronization is incurred only when a worker runs out of work. If an application exhibits sufficient parallelism, stealing is infrequent, and thus the cost of bookkeeping, communication, and synchronization to effect a steal is negligible.

Work-stealing achieves good expected running time based on the work and span. In particular, if A is the executed dag on P processors, the expected execution time T_P can be bounded as

$$T_P \le \text{Work}(A)/P + O(\text{Span}(A)) \,, \qquad (3)$$

where we omit the notation for expectation for simplicity. This bound, which is proved in [7], assumes an ideal computer, but it includes scheduling overhead. For a deterministic computation, if the parallelism exceeds the number P of processors sufficiently, Inequality (3) guarantees near-linear speedup. Specifically, if $P \ll \text{Work}(A)/\text{Span}(A)$, then $\text{Span}(A) \ll \text{Work}(A)/P$, and hence Inequality (3) yields $T_P \approx \text{Work}(A)/P$, and the speedup is $T_1/T_P \approx P$.

For a nondeterministic computation such as PBFS, however, the work of a P-processor execution may not readily be related to the serial running time. Thus, obtaining bounds on speedup can be more challenging. As Section 9 shows, however, PBFS achieves

$$T_P(A) \le \text{Work}(A_\upsilon)/P + O(\tau^2 \cdot \text{Span}(A_\upsilon)) \,, \qquad (4)$$

where A_υ is the "user dag" of A — the dag from the programmer's perspective — and τ is an upper bound on the time it takes to perform a REDUCE, which may be a function of the input size. (We shall formalize these concepts in Sections 7 and 8.) For nondeterministic computations satisfying Inequality (4), we can define the ***effective parallelism*** as $\text{Work}(A_\upsilon)/(\tau^2 \cdot \text{Span}(A_\upsilon))$. Just as with parallelism for deterministic computations, if the effective parallelism exceeds the number P of processors by a sufficient margin, the P-processor execution is guaranteed to attain near-linear speedup over the serial execution.

Another relevant measure is the number of steals that occur during a computation. As is shown in [7], the expected number of steals incurred for a dag A produced by a P-processor execution is $O(P \cdot \text{Span}(A))$. This bound is important, since the number of REDUCE operations needed to combine reducer views is bounded by the number of steals.

7. MODELING REDUCERS

This section reviews the definition of reducer hyperobjects from [14] and extends the dag model to incorporate them. We define the notion of a "user dag" for a computation, which represents the strands that are visible to the programmer. We also define the notion of a "performance dag," which includes the strands that the runtime system implicitly invokes.

A reducer is defined in terms of an algebraic ***monoid***: a triple $(T, \otimes, e)$, where T is a set and $\otimes$ is an associative binary operation over T with identity e. From an object-oriented programming perspective, the set T is a base type which provides a member function REDUCE implementing the binary operator $\otimes$ and a member function CREATE-IDENTITY that constructs an identity element of type T. The base type T also provides one or more UPDATE functions, which modify an object of type T. In the case of bags, the REDUCE function is BAG-UNION, the CREATE-IDENTITY function is BAG-CREATE, and the UPDATE function is BAG-INSERT. As a practical matter, the REDUCE function need not actually be associative, although in that case, the programmer typically has some idea of "logical" associativity. Such is the case, for example, with bags. If we have three bags B_1, B_2, and B_3, we do not care whether the bag data structures for $(B_1 \cup B_2) \cup B_3$ and $B_1 \cup (B_2 \cup B_3)$ are identical, only that they contain the same elements.

To specify the nondeterministic behavior encapsulated by reducers precisely, consider a computation A of a multithreaded program, and let $V(A)$ be the set of executed strands. We assume that the implicitly invoked functions for a reducer — REDUCE and CREATE-IDENTITY — execute only serial code. We model each execution of one of these functions as a single strand containing the instructions of the function. If an UPDATE causes the runtime system to invoke CREATE-IDENTITY implicitly, the serial code arising from UPDATE is broken into two strands sandwiching the point where CREATE-IDENTITY is invoked.

We partition $V(A)$ into three classes of strands:

- V_ι: ***Init strands*** arising from the execution of CREATE-IDENTITY when invoked implicitly by the runtime system, which occur when the user program attempts to update a reducer, but a local view has not yet been created.

- V_ρ: ***Reducer strands*** arising from the execution of REDUCE, which occur implicitly when the runtime system combines views.

- V_υ: ***User strands*** arising from the execution of code explicitly invoked by the programmer, including calls to UPDATE.

We call $V_\iota \cup V_\rho$ the set of ***runtime strands***.

Since, from the programmer's perspective, the runtime strands

are invoked "invisibly" by the runtime system, his or her understanding of the program generally relies only on the user strands. We capture the control dependencies among the user strands by defining the ***user dag*** $A_\upsilon = (V_\upsilon, E_\upsilon)$ for a computation A in the same manner as we defined an ordinary multithreaded dag. For example, a spawn strand e_1 has out-degree 2 in A_υ with an edge (v_1, v_2) going to the first strand v_2 of the spawned child and the other edge (v_2, v_3) going to the continuation v_3; if v_1 is the final strand of a spawned subroutine and v_2 is the sync strand with which v_1 syncs, then we have $(v_1, v_2) \in E_\upsilon$; etc.

To track the views of a reducer h in the user dag, let $h(v)$ denote the view of h seen by a strand $v \in V_\upsilon$. The runtime system maintains the following invariants:

1. If $u \in V_\upsilon$ has out-degree 1 and $(u, v) \in E_\upsilon$, then $h(v) = h(u)$.

2. Suppose that $u \in V_\upsilon$ is a spawn strand with outgoing edges $(u, v), (u, w) \in E_\upsilon$, where $v \in V_\upsilon$ is the first strand of the spawned subroutine and $w \in V_\upsilon$ is the continuation in the parent. Then, we have $h(v) = h(u)$ and

$$h(w) = \begin{cases} h(u) & \text{if } u \text{ was not stolen;} \\ \text{new view} & \text{otherwise.} \end{cases}$$

3. If $v \in V_\upsilon$ is a sync strand, then $h(v) = h(u)$, where u is the first strand of v's function.

When a new view $h(w)$ is created, as is inferred by Invariant 2, we say that the old view $h(u)$ ***dominates*** $h(w)$, which we denote by $h(u) \succ h(w)$. For a set H of views, we say that two views $h_1, h_2 \in H$, where $h_1 \succ h_2$, are ***adjacent*** if there does not exist $h_3 \in H$ such that $h_1 \succ h_3 \succ h_2$.

A useful property of sync strands is that the views of strands entering a sync strand $v \in V_\upsilon$ are totally ordered by the "dominates" relation. That is, if k strands each have an edge in E_υ to the same sync strand $v \in V_\upsilon$, then the strands can be numbered $u_1, u_2, \ldots, u_k \in V_\upsilon$ such that $h(u_1) \succeq h(u_2) \succeq \cdots \succeq u_k$. Moreover, $h(u_1) = h(v) = h(u)$, where u is the first strand of v's function. These properties can be proved inductively, noting that the views of the first and last strands of a function must be identical, because a function implicitly syncs before it returns. The runtime system always reduces adjacent pairs of views in this ordering, destroying the dominated view in the pair.

If a computation A does not involve any runtime strands, the "delay-sequence" argument in [7] can be applied to A_υ to bound the P-processor execution time: $T_P(A) \leq \text{Work}(A_\upsilon)/P + O(\text{Span}(A_\upsilon))$. Our goal is to apply this same analytical technique to computations containing runtime strands. To do so, we augment the A_υ with the runtime strands to produce a ***performance dag*** $A_\pi = (V_\pi, E_\pi)$ for the computation A, where

- $V_\pi = V(A) = V_\upsilon \cup V_\iota \cup V_\rho$,
- $E_\pi = E_\upsilon \cup E_\iota \cup E_\rho$,

where the edge sets E_ι and E_ρ are constructed as follows.

The edges in E_ι are created in pairs. For each init strand $v \in V_\iota$, we include (u, v) and (v, w) in E_ι, where $u, w \in V_\upsilon$ are the two strands comprising the instructions of the UPDATE whose execution caused the invocation of the CREATE-IDENTITY corresponding to v.

The edges in E_ρ are created in groups corresponding to the set of REDUCE functions that must execute before a given sync. Suppose that $v \in V_\upsilon$ is a sync strand, that k strands $u_1, u_2, \ldots, u_k \in A_\upsilon$ join at v, and that $k' < k$ reduce strands $r_1, r_2, \ldots, r_{k'} \in A_\rho$ execute before the sync. Consider the set $U = \{u_1, u_2, \ldots, u_k\}$, and let $h(U) = \{h(u_1), h(u_2), \ldots, h(u_k)\}$ be the set of $k' + 1$ views that must be reduced. We construct a ***reduce tree*** as follows:

1 **while** $|h(U)| \geq 2$
2   Let $r \in \{r_1, r_2, \ldots, r_{k'}\}$ be the reduce strand that reduces a "minimal" pair $h_j, h_{j+1} \in h(U)$ of adjacent strands, meaning that if a distinct $r' \in \{r_1, r_2, \ldots, r_{k'}\}$ reduces adjacent strands $h_i, h_{i+1} \in h(U)$, we have $h_i \succ h_j$
3   Let $U_r = \{u \in U : h(u) = h_j \text{ or } h(u) = h_{j+1}\}$
4   Include in E_ρ the edges in the set $\{(u, r) : u \in U_r\}$
5      $U = U - U_r \cup \{r\}$
6 Include in E_ρ the edges in the set $\{(r, v) : r \in U\}$

Since the reduce trees and init strands only add more dependencies between strands in the user A_υ that are already in series, the performance dag A_π is indeed a dag.

Since the runtime system performs REDUCE operations opportunistically, the reduce strands in the performance dag may execute before their predecessors have completed. The purpose of performance dags, as Section 8 shows, is to account for the cost of the runtime strands, not to describe how computations are scheduled.

8. ANALYSIS OF PROGRAMS WITH NONCONSTANT-TIME REDUCERS

This section provides a framework for analyzing programs that contain reducers whose REDUCE functions execute in more than constant time.

We begin with a lemma that bounds the running time of a computation in terms of the work and span of its performance dag.

LEMMA 1. *Consider the execution of a computation A on a parallel computer with P processors using a work-stealing scheduler. The expected running time of A is $T_P(A) \leq \text{Work}(A_\pi)/P + O(\text{Span}(A_\pi))$.*

PROOF. The proof follows those of [7] and [14], with some salient differences. As in [7], we use a delay-sequence argument, but we base it on the performance dag.

The normal delay-sequence argument involves only a user dag. This dag is augmented with "deque" edges, each running from a continuation on the deque to the next in sequence from top to bottom. These deque edges increase the span of the dag by at most a constant factor. The argument then considers a path in the dag, and it defines an instruction as being ***critical*** if all its predecessors in the augmented dag have been executed. The key property of the work-stealing algorithm is that every critical instruction sits atop of some deque (or is being executed by a worker). Thus, whenever a worker steals, it has a $1/P$ chance of executing a critical instruction. With constant probability, P steals suffice to reduce the span of the dag of the computation that remains to be executed by 1. Consequently, the expected number of steals is $O(P \cdot \text{Span}(A_\pi))$. A similar but slightly more complex bound holds with high probability.

This argument can be modified to work with performance dags containing reducers that operate in nonconstant-time. As instructions in the computation are executed, we can mark them off in the performance dag. Since we have placed reduce strands after strands in the performance dag before which they may have actually executed, some reduce strands may execute before all of their predecessors in the performance dag complete. That is okay. The main property is that if an instruction is critical, it has a $1/P$ chance of being executed upon a steal, and that P steals have a constant expectation of reducing the span of the dag that remains to execute by 1. The crucial observation is that if an instruction in a reduce strand is critical, then its sync node has been reached, and thus a worker must be executing the critical instruction, since reduces are performed eagerly when nothing impedes their execution. It follows that the expected running time of A is $T_P(A) \leq \text{Work}(A_\pi)/P + O(\text{Span}(A_\pi))$. $\square$

We want to ensure that the runtime system joins strands quickly when reducers are involved. Providing a guarantee requires that we examine the specifics of how the runtime system handles reducers.

First, we review how the runtime system handles spawns and steals, as described by Frigo *et al.* [14]. Every time a Cilk function is stolen, the runtime system creates a new *frame*.[5] Although frames are created and destroyed dynamically during a program execution, the ones that exist always form a rooted *spawn tree*. Each frame F provides storage for temporary values and local variables, as well as metadata for the function, including the following:

- a pointer $F.lp$ to F's left sibling, or if F is the first child, to F's parent;
- a pointer $F.c$ to F's first child;
- a pointer $F.r$ to F's right sibling.

These pointers form a left-child right-sibling representation of the part of the spawn tree that is distributed among processors, which is known as the *steal tree*.

To handle reducers, each worker in the runtime system uses a hash table called a *hypermap* to map reducers into its local views. To allow for lock-free access to the hypermap of a frame F while siblings and children of the frame are terminating, F stores three hypermaps, denoted $F.hu$, $F.hr$, and $F.hc$. The $F.hu$ hypermap is used to look up reducers for the user's program, while the $F.hr$ and $F.hc$ hypermaps store the accumulated values of F's terminated right siblings and terminated children, respectively.

When a frame is initially created, its hypermaps are empty. If a worker using a frame F executes an UPDATE operation on a reducer h, the worker tries to get h's current view from the $F.hu$ hypermap. If h's view is empty, the worker performs a CREATE-IDENTITY operation to create an identity view of h in $F.hu$.

When a worker returns from a spawn, first it must perform up to two REDUCE operations to reduce its hypermaps into its neighboring frames, and then it must *eliminate* its current frame. To perform these REDUCE operations and elimination without races, the worker grabs locks on its neighboring frames. The algorithm by Frigo *et al.* [14] uses an intricate protocol to avoid long waits on locks, but the analysis of its performance assumes that each RE-DUCE takes only constant time.

To support nonconstant-time REDUCE functions, we modify the locking protocol. To eliminate a frame F, the worker first reduces $F.hu \otimes= F.hr$. Second, the worker reduces $F.lp.hc \otimes= F.hu$ or $F.lp.hr \otimes= F.hu$, depending on whether F is a first child.

Workers eliminating $F.lp$ and $F.r$ might race with the elimination of F. To resolve these races, Frigo *et al.* describe how to acquire an abstract lock between F and these neighbors, where an abstract lock is a pair of locks that correspond to an edge in the steal tree. We shall use these abstract locks to eliminate a frame F according to the locking protocol shown in Figure 11.

The next lemma analyzes the work required to perform all eliminations using this locking protocol.

LEMMA 2. *Consider the execution of a computation A on a parallel computer with P processors using a work-stealing scheduler. The total work involved in joining strands is $O(\tau P \cdot \text{Span}(A_\pi))$, where τ is the worst-case cost of any* REDUCE *or* CREATE-IDENTITY *for the given input.*

PROOF. Since lines 3–15 in the new locking protocol all require $O(1)$ work, each abstract lock is held for a constant amount of time.

The analysis of the time spent waiting to acquire an abstract lock in the new locking protocol follows the analysis of the locking protocol in [14]. The key issue in the proof is to show that

```
1   while TRUE
2       Acquire the abstract locks for edges (F, F.lp) and (F, F.r) in an
        order chosen uniformly at random
3       if F is a first child
4           L = F.lp.hc
5       else L = F.lp.hr
6       R = F.hr
7       if L == ∅ and R == ∅
8           if F is a first child
9               F.lp.hc = F.hu
10          else F.lp.hr = F.hu
11          Eliminate F
12          break
13      R' = R; L' = L
14      R = ∅; L = ∅
15      Release the abstract locks
16      for each reducer h ∈ R
17          if h ∈ F.hu
18              F.hu(h) ⊗= R(h)
19          else F.hu(h) = R(h)
20      for each reducer h ∈ L
21          if h ∈ F.hu
22              F.hu(h) = L(h) ⊗ F.hu(h)
23          else F.hu(h) = L(h)
24
```

Figure 11: A modified locking protocol for managing reducers, which holds locks for $O(1)$ time.

the time for the ith abstract lock acquisition by some worker w is independent of the time for w's jth lock acquisition for all $j > i$. To prove this independence result, we shall argue that for two workers w and v, we have $\Pr\{v \text{ delays } w_j | v \text{ delays } w_i\} = \Pr\{v \text{ delays } w_j | v \text{ does not delay } w_i\} = \Pr\{v \text{ delays } w_j\}$, where w_i and w_j are w's ith and jth lock acquisitions, respectively.

We shall consider each of these cases separately. First, suppose that v delays w_i. After w_i, v has succeeded in acquiring and releasing its abstract locks, and all lock acquisitions in the directed path from w's lock acquisition to v's have also succeeded. For v to delay w_j, a new directed path of dependencies from w to v must occur. Each edge in that path is oriented correctly with a $1/2$ probability, regardless of any previous interaction between v and w. Similarly, suppose that v does not delay w_i. For v to delay w_j, a chain of dependencies must form from one of w's abstract locks to one of v's abstract locks after w_i completes. Forming such a dependency chain requires every edge in the chain to be correctly oriented, which occurs with a $1/2$ probability per edge regardless of the fact that v did not delay w_i. Therefore, we have $\Pr\{v \text{ delays } w_j | v \text{ delays } w_i\} = \Pr\{v \text{ delays } w_j | v \text{ does not delay } w_i\} = \Pr\{v \text{ delays } w_j\}$.

For all workers $v \neq w$, the probability that v delays w_j is independent of whether v delays w_i. Consequently, every lock acquisition by some worker is independent of all previous acquisitions, and by the analysis in [14], the total time a worker spends in abstract lock acquisitions is $O(m)$ in expectation, where m is the number of abstract lock acquisitions that worker performs. Moreover, the total time spent in abstract lock acquisitions is proportional to the number of elimination attempts.

Next, we bound the total number of elimination attempts performed in this protocol. Since each successful steal creates a frame in the steal tree that must be eliminated, the number of elimination attempts is at least as large as the number M of successful steals. Each elimination of a frame may force two other frames to repeat this protocol. Therefore, each elimination increases the number of elimination attempts by at most 2. Thus, the total number of elimination attempts is no more than $3M$.

Finally, we bound the total work spent joining strands using this protocol. The total time spent acquiring abstract locks and performing the necessary operations while the lock is held is $O(M)$. Each failed elimination attempt triggers at most two REDUCE operations, each of which takes τ work in the worst case. Therefore, the total expected work spent joining strands is $O(\tau M)$. Using the analysis of steals from [7], the total work spent joining strands is $O(\tau P \cdot \text{Span}(A_\pi))$. $\square$

The following two lemmas bound the work and span of the performance dag in terms of the span of the user dag. For simplicity, assume that A makes use of a single reducer. (These proofs can be extended to handle many reducers within a computation.)

LEMMA 3. *Consider a computation A with user dag A_υ and performance dag A_π, and let τ be the worst-case cost of any* CREATE-IDENTITY *or* REDUCE *operation for the given input. Then, we have* $\text{Span}(A_\pi) = O(\tau \cdot \text{Span}(A_\upsilon))$.

PROOF. Each successful steal in the execution of A may force one CREATE-IDENTITY. Each CREATE-IDENTITY creates a nonempty view that must later be reduced using a REDUCE operation. Therefore, at most one REDUCE operation may occur per successful steal, and at most one reduce strand may occur in the performance dag for each steal. Each spawn in A_υ provides an opportunity for a steal to occur. Consequently, each spawn operation in A may increase the size of the dag by 2τ in the worst case.

Consider a critical path in A_π, and let p_υ be the corresponding path in A_υ. Suppose that k steals occur along p_υ. The length of that corresponding path in A_π is at most $2k\tau + |p_\upsilon| \leq 2\tau \cdot \text{Span}(A_\upsilon) + |p_\upsilon| \leq 3\tau \cdot \text{Span}(A_\upsilon)$. Therefore, we have $\text{Span}(A_\pi) = O(\tau \cdot \text{Span}(A_\upsilon))$. $\square$

LEMMA 4. *Consider a computation A with user dag A_υ, and let τ be the worst-case cost of any* CREATE-IDENTITY *or* REDUCE *operation for the given input. Then, we have* $\text{Work}(A_\pi) = \text{Work}(A_\upsilon) + O(\tau^2 P \cdot \text{Span}(A_\upsilon))$.

PROOF. The work in A_π is the work in A_υ plus the work represented in the runtime strands. The total work in reduce strands equals the total work to join stolen strands, which is $O(\tau P \cdot \text{Span}(A))$ by Lemma 2. Similarly, each steal may create one init strand, and by the analysis of steals from [7], the total work in init strands is $O(\tau P \cdot \text{Span}(A))$. Thus, we have $\text{Work}(A_\pi) = \text{Work}(A_\upsilon) + O(\tau P \cdot \text{Span}(A_\pi))$. Applying Lemma 3 yields the lemma. $\square$

We now prove Inequality (4), which bounds the runtime of a computation whose nondeterminism arises from reducers.

THEOREM 5. *Consider the execution of a computation A on a parallel computer with P processors using a work-stealing scheduler. Let A_υ be the user dag of A. The total running time of A is* $T_P(A) \leq \text{Work}(A_\upsilon)/P + O(\tau^2 \cdot \text{Span}(A_\upsilon))$.

PROOF. The proof follows from Lemmas 1, 3, and 4. $\square$

9. ANALYZING PBFS

This section applies the results of Section 8 to bound the expected running time of the locking version of PBFS.

First, we bound the work and span of the user dag for PBFS.

LEMMA 6. *Suppose that the locking version of PBFS is run on a connected graph $G = (V, E)$ with diameter D. The total work in PBFS's user dag is $O(V + E)$, and the total span of PBFS's user dag is $O(D \lg(V/D) + D \lg \Delta)$, where Δ is the maximum out-degree of any vertex in V.*

PROOF. In each layer, PBFS evaluates every vertex v in that layer exactly once, and PBFS checks every vertex u in v's adjacency list exactly once. In the locking version of PBFS, each u is assigned its distance exactly once and added to the bag for the next layer exactly once. Since this holds for all layers of G, the total work for this portion of PBFS is $O(V + E)$.

PBFS performs additional work to create a bag for each layer and to repeatedly split the layer into GRAINSIZE pieces. If D is the number of layers in G, then the total work PBFS spends in calls to BAG-CREATE is $O(D \lg V)$. The analysis for the work PBFS performs to subdivide a layer follows the analysis for building a binary heap [10, Ch. 6]. Therefore, the total time PBFS spends in calls to BAG-SPLIT is $O(V)$.

The total time PBFS spends executing BAG-INSERT depends on the parallel execution of PBFS. Since a steal resets the contents of a bag for subsequent update operations, the maximum running time of BAG-INSERT depends on the steals that occur. Each steal can only decrease the work of a subsequent BAG-INSERT, and therefore the amortized running time of $O(1)$ for each BAG-INSERT still applies. Because BAG-INSERT is called once per vertex, PBFS spends $O(V)$ work total executing BAG-INSERT, and the total work of PBFS is $O(V + E)$.

The sequence of splits performed in each layer cause the vertices of the layer to be processed at the leaves of a balanced binary tree of height $O(\lg V_d)$, where V_d is the set of vertices in the dth layer. Since the series of syncs that PBFS performs mirror this split tree, the divide-and-conquer computation to visit the vertices in a layer and then combine the results has span $O(\lg V_d)$. Each leaf of this tree processes at most a constant number of vertices and looks at the outgoing edges of those vertices in a similar divide-and-conquer fashion. This divide-and-conquer evaluation results in a computation at each leaf with span $O(\lg \Delta)$. Each edge evaluated performs some constant-time work and may trigger a call to BAG-INSERT, whose worst-case running time would be $O(\lg V_{d+1})$. Consequently, the span of PBFS for processing the dth layer is $O(\lg V_d + \lg V_{d+1} + \lg \Delta)$. Summing this quantity over all layers in G, the maximum span for PBFS is $O(D \lg(V/D) + D \lg \Delta)$. $\square$

We now bound the expected running time of PBFS.

THEOREM 7. *Consider the parallel execution of PBFS on a connected graph $G = (V, E)$ with diameter D running on a parallel computer with P processors using a work-stealing scheduler. The expected running time of the locking version of PBFS is $T_P(\text{PBFS}) \leq O(V + E)/P + O(D \lg^2(V/D)(\lg(V/D) + \lg \Delta))$, where Δ is the maximum out-degree of any vertex in V. If we have $\Delta = O(1)$, then the expected running time of PBFS is $T_P(\text{PBFS}) \leq O(V + E)/P + O(D \lg^3(V/D))$.*

PROOF. To maximize the cost of all CREATE-IDENTITY and REDUCE operations in PBFS, the worst-case cost of each of these operations must be $O(\lg(V/D))$. Applying Theorem 5 with $\tau = O(\lg(V/D))$, $\text{Work}(\text{PBFS}) = O(V + E)$, and $\text{Span}(\text{PBFS}) = O(D \lg(V/D) + D \lg \Delta)$, we get $T_P(\text{PBFS}) \leq O(V + E)/P + O(D \lg^2(V/D)(\lg(V/D) + \lg \Delta))$. If we have $\Delta = O(1)$, this formula simplifies to $T_P(\text{PBFS}) \leq O(V + E)/P + O(D \lg^3(V/D))$. $\square$

10. CONCLUSION

Thread-local storage [27], or *TLS*, presents an alternative to bag reducers for implementing the layer sets in a parallel breadth-first search. The bag reducer allows PBFS to write the vertices of a layer in a single data structure in parallel and later efficiently traverse them in parallel. As an alternative to bags, each of the P workers

could store the vertices it encounters into a vector within its own TLS, thereby avoiding races. The set of elements in the P vectors could then be walked in parallel using divide-and-conquer. Such a structure appears simple to implement and practically efficient, since it avoids merging sets.

Despite the simplicity of the TLS solution, reducer-based solutions exhibit some advantages over TLS solutions. First, reducers provide a processor-oblivious alternative to TLS, enhancing portability and simplifying reasoning of how performance scales. Second, reducers allow a function to be instantiated multiple times in parallel without interference. To support simultaneous running of functions that use TLS, the programmer must manually ensure that the TLS regions used by the functions are disjoint. Third, reducers require only a monoid — associativity and an identity — to ensure correctness, whereas TLS also requires commutativity. The correctness of some applications, including BFS, is not compromised by allowing commutative updates to its shared data structure. Without commutativity, an application cannot easily use TLS, whereas reducers seem to be good whether commutativity is allowed or not. Finally, whereas TLS makes the nondeterminism visible to the programmer, reducers encapsulate nondeterminism. In particular, reducers hide the particular nondeterministic manner in which associativity is resolved, thereby allowing the programmer to assume specific semantic guarantees at well-defined points in the computation. This encapsulation of nondeterminism simplifies the task of reasoning about the program's correctness compared to a TLS solution.

Nondeterminism can wreak havoc on the ability to reason about programs, to test their correctness, and to ascertain their performance, but it also can provide opportunities for additional parallelism. Well-structured linguistic support for encapsulating nondeterminism may allow parallel programmers to enjoy the benefits of nondeterminism without suffering unduly from the inevitable complications that nondeterminism engenders. Reducers provide an effective way to encapsulate nondeterminism. We view it as an open question whether a semantics exists for TLS that would encapsulate nondeterminism while providing a potentially more efficient implementation in situations where commutativity is allowed.

11. ACKNOWLEDGMENTS

12. REFERENCES

[1] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA*, pp. 119–129, 1998.

[2] D. Bader, J. Feo, J. Gilbert, J. Kepner, D. Keoster, E. Loh, K. Madduri, B. Mann, and T. Meuse. HPCS scalable synthetic compact applications #2, 2007. Available at `http://www.graphanalysis.org/benchmark/HPCS-SSCA2_Graph-Theory_v2.2.doc`.

[3] D. A. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and *st*-connectivity on the Cray MTA-2. In *ICPP*, pp. 523–530, 2006.

[4] G. E. Blelloch. Programming parallel algorithms. *CACM*, 39(3), 1996.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *JPDC*, 37(1):55–69, 1996.

[6] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. on Comput.*, 27(1):202–229, 1998.

[7] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.

[8] R. P. Brent. The parallel evaluation of general arithmetic expressions. *JACM*, 21(2):201–206, 1974.

[9] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving large, irregular graph problems using adaptive work-stealing. In *ICPP*, pp. 536–545, 2008.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[11] T. A. Davis. University of Florida sparse matrix collection, 2010. Available at `http://www.cise.ufl.edu/research/sparse/matrices/`.

[12] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *ICCD*, pp. 522–525, 1992.

[13] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Trans. Comput.*, 38(3):408–423, 1989.

[14] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *SPAA*, pp. 79–90, 2009.

[15] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pp. 212–223, 1998.

[16] J. R. Gilbert, G. L. Miller, and S.-H. Teng. Geometric mesh partitioning: Implementation and experiments. *SIAM J. on Sci. Comput.*, 19(6):2091–2110, 1998.

[17] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell Sys. Tech. J.*, 45:1563–1581, 1966.

[18] Y. He. Multicore-enabling the Murphi verification tool. Available from `http://software.intel.com/en-us/articles/multicore-enabling-the-murphi-verification-tool/`, 2009.

[19] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview scalability analyzer. In *SPAA*, 2010.

[20] Intel Corporation. *Intel Cilk++ SDK Programmer's Guide*, 2009. Document Number: 322581-001US.

[21] R. E. Korf and P. Schultze. Large-scale parallel breadth-first search. In *AAAI*, pp. 1380–1385, 2005.

[22] C. Y. Lee. An algorithm for path connection and its applications. *IRE Trans. on Elec. Comput.*, EC-10(3):346–365, 1961.

[23] C. E. Leiserson. The Cilk++ concurrency platform. *J. Supercomput.*, 51(3):244–257, 2010.

[24] J. Leskovec, D. Chakrabarti, J. M. Kleinberg, and C. Faloutsos. Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication. In *PKDD*, pp. 133–145, 2005.

[25] J. B. Lubos, L. Brim, and J. Chaloupka. Parallel breadth-first search LTL model-checking. In *ASE*, pp. 106–115, 2003.

[26] E. F. Moore. The shortest path through a maze. In *Int. Symp. on Th. of Switching*, pp. 285–292, 1959.

[27] D. Stein and D. Shah. Implementing lightweight threads. In *USENIX*, pp. 1–9, 1992.

[28] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 2000.

[29] Supertech Research Group, MIT/LCS. *Cilk 5.4.6 Reference Manual*, 1998. Available from `http://supertech.csail.mit.edu/cilk/`.

[30] A. van Heukelum, G. T. Barkema, and R. H. Bisseling. Dna electrophoresis studied with the cage model. *J. Comput. Phys.*, 180(1):313–326, 2002.

[31] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *SC '05*, p. 25, 2005.

[32] Y. Zhang and E. A. Hansen. Parallel breadth-first heuristic search on a shared-memory architecture. In *AAAI Workshop on Heuristic Search, Memory-Based Heuristics and Their Applications*, 2006.