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 near-perfect linear speedup in the number of processing cores, as long as the machine has sufficient memory bandwidth.

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 $O((V + E)/P + O(D \lg^3(V/D))$ on P processors.

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 [28] in the context of finding paths through mazes. Lee [23] 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, 22, 26, 33, 35]. 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 the best 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 on benchmark graphs.

```
SERIAL-BFS$(V, E, v_0)$
 1  for each vertex $u \in V - \{v_0\}$
 2      $u.dist = \infty$
 3  $v_0.dist = 0$
 4  $Q = \{v_0\}$
 5  while $Q \neq \emptyset$
 6      $u = $ DEQUEUE$(Q)$
 7      for each $v \in V$ such that $(u, v) \in E$
 8          if $v.dist == \infty$
 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.

Given a graph G 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 $v.dist$ of v 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 (see, for example, [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 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

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

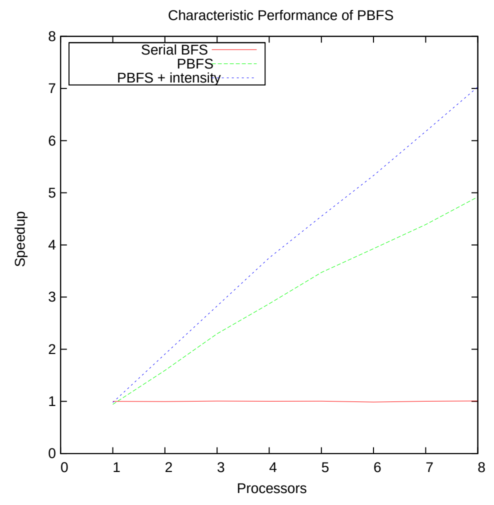


Figure 2: The characteristic performance of PBFS for a large graph showing speedup curves for serial BFS and for PBFS. In addition, the figure shows the curve for a variant of PBFS where the computational intensity has been artificially enhanced and the speedup normalized.

parallelism — the ratio of work to span — at most $O((V+E)/V) = O(1)$ if $|E| = O(V)$.

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, 24], called ***PBFS***, which employs bags instead of a FIFO. The bags are incorporated into the program using a Cilk++ "hyperobject" feature called a "reducer" [16]. Our Cilk++ implementation of PBFS runs as fast on a single processor as 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 linear speedup with the number of processing cores, at least until it encounters architectural limits on memory bandwidth.

Figure 2 shows the typical speedup obtained for PBFS on a large benchmark graph, in this case, for a sparse matrix arising from the study of electrical power flow [34]. This graph has $|V| = 2,063,494$ vertices, $|E| = 12,771,361$ edges, and a diameter of $D = 31$. The code was run on an Intel Core i7 (Nahalem) machine with eight 2.53GHz processing cores, 12GB of RAM, and an 8MB cache per core. As can be seen from the figure, although PBFS scales well initially, it attains a speedup of only about 5 on 8 cores. By artificially increasing the ***computational intensity*** — the ratio of the number of CPU operations to the number of memory operations — the figure shows that the leveling off is due to the architectural limitations of memory bandwidth, 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. In addition, for efficient implementation, PBFS contains a benign race condition, which also 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 $E = E(G)$, 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 organized as follows. Section 2 provides background on the dag model of multithreading. Section 3 describes the basic PBFS algorithm, and Section 4 describes the implementation of the bag data structure. Section 5 gives a formal model for reducer behavior, Section 6 develops a theory for analyzing programs that use reducers, and Section 7 employs this theory to analyze the performance of PBFS. Section 8 presents our empirical studies. Finally, Section 9 offers some generalizations and concluding remarks.

2. BACKGROUND ON DYNAMIC MULTITHREADING

This section overviews the key attributes of dynamic multithreading. The PBFS software is implemented in Cilk++ [16, 20, 24], which is a linguistic extension to C++ [31], but most of the vagaries of C++ are unnecessary for understanding the issues. Thus, this section describes Cilk-like pseudocode, as is exemplified in [10, Ch. 27], which should be more straightforward than real code for the reader to understand and which can be translated easily to Cilk++. We explain how a multithreaded program execution can be modeled theoretically using the framework of Feng and Leiserson [15] (as opposed to the similar framework in [10, Ch. 27]), and overview assumptions about the runtime environment. We define deterministic and nondeterministic computations and describe how reducer hyperobjects fit into the theoretical framework.

Multithreaded pseudocode

The linguistic model for multithreaded pseudocode in [10, Ch. 27] follows MIT Cilk [17,32] and Cilk++ [20,24]. 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 C or C++ function call only in that the parent may continue to 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. 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**.

A key property of this linguistic model is that a multithreaded program admits a serial execution. Simply eliding the keywords **spawn**, **sync**, and **parallel** produces a serial program, called the ***serialization***, which implements the semantics of the multithreaded program. The serialization has the property that spawned children

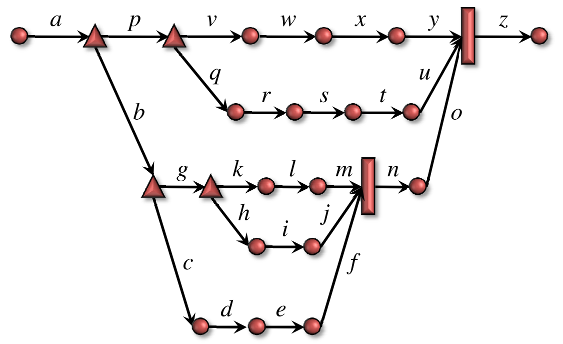


Figure 3: A directed acyclic graph representation of a multithreaded execution. Each edge represents an instruction. Triangular nodes represent spawns, and rectangular nodes represent syncs. The partial order given by the dag represents the ordering constraints among instructions during program execution.

are simply called, and they complete their execution before the parent resumes, as with an ordinary function call.

Cilk++ provides a novel linguistic construct, called ***reducer hyperobjects*** [16], which allow 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 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.

Cilk++'s reducer mechanism supports this kind of decomposition of update sequences automatically without requiring the programmer to manually create the view x'. 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 chain of updates can continue. If the continuation begins executing before the child returns, however, it 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. The behavior is virtually identically to a serial execution that uses an ordinary object instead of a hyperobject.

We shall discuss precise semantics for reducers in Section 5.

The dag model

We shall adopt the dag model for multithreading similar to the one introduced by Feng and Leiserson [15]. This model was designed to model the execution of spawns and syncs. We shall extend it in Section 5 to deal with reducers.

The dag model views the executed computation resulting from the running of a multithreaded program (on a given input[2]) as a ***dag (directed acyclic graph)*** A, where the edge set consists of ***strands*** — sequences of serially executed instructions containing no parallel control — and the vertex set contains connective nodes. We shall use A to denote both the dag and the set of strands in the dag. Figure 3 illustrates such a dag, which is in some sense 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. Generally, we 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 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.

[2]When we talk about program executions, we shall generally assume that we mean "on a given input."

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.

A program execution contains a ***determinacy race***[3] [15] if two parallel strands access the same memory location and at least one of the strands updates it. A program with no determinacy races always produces the same dag, no matter how the execution is scheduled. A ***race-free*** dag contains no determinacy races.

Work and span

The dag model admits two natural measures of performance which can be used to provide important bounds [6, 8, 13, 19] 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 3 is 26.
- The ***span***[4] of A, denoted by $\text{Span}(A)$, is length of the longest path in the dag. Assuming unit-time strands, the span of the dag in Figure 3 is 9, which is realized by both the path *abgklmnoz* and the path *abghijnoz*.

The work/span model is outlined in tutorial fashion in [10, Ch. 27], although that presentation uses nodes for strands and edges to indicate dependencies between strands.

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

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

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

[3]Determinacy races have been given many different names in the literature. For example, they are sometimes called ***access anomalies*** [12], ***data races*** [27], ***race conditions*** [21], or ***harmful shared-memory accesses*** [30]. Netzer and Miller [29] clarify different types of races and define a ***general race*** or ***determinacy race*** to be a race that causes a supposedly deterministic program to behave nondeterministically. (They also define a ***data race*** or ***atomicity race*** to be a race in a nondeterministic program involving nonatomic accesses to critical regions.) We prefer the more descriptive term "determinacy race." Emrath and Padua [14] call a deterministic program ***internally deterministic*** if the program execution on the given input exhibits no determinacy race and ***externally deterministic*** if the program has determinacy races but its output is deterministic because of the commutative and associative operations performed on the shared locations.

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

execute at most P instructions per unit time. Thus, with P processors, to do all the work, it must take at least $\text{Work}(A)/P$ 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 $T_/T_P \le P$, which is to say that the speedup on P processors can be at most P. If the application obtains speedup proportional to P, we say that the application exhibits ***linear speedup***. If it obtains speedup exactly P (which is the best we can do in our model), we say that the application exhibits ***perfect 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_∞ and 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 3 is $26/9 \approx 2.89$, which means that there is little point in executing it with more than 3 processors, since the additional processors will surely be starved for work.

Scheduling

A ***greedy scheduler*** [6, 8, 13, 19] schedules a computation without ever leaving a processor idle if there is work that can be done. If a program scheduled by a greedy scheduler produces a dag A, then we have

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

This bound assumes an ideal computer and ignores overheads for scheduling. For a deterministic computation, if the parallelism exceeds the number P of processors by a sufficient margin, Inequality (3) guarantees near-perfect 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 $T_1/T_P \approx P$.

A randomized "work-stealing" scheduler [1, 7], such as is provided by Cilk++ and MIT Cilk, is a more practical scheduling algorithm than greedy for multithreaded programs. Cilk++'s work-stealing scheduler 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 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, Cilk++ operates just like C++ 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.

Consequently, 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

$$T_P \le \text{Work}(A)/P + O(\text{Span}(A)) , \qquad (4)$$

where we omit the notation for expectation for simplicity. This bound, which is proved in [7], includes scheduling overhead.

Notice that the only difference between Inequality (3) and Inequality (4) is the O around the span. Thus, we can generally use greedy scheduling to reason about computations scheduled by the more-practical work-stealing scheduler. For example, for a deterministic computation, if the parallelism exceeds the number P of processors by a sufficient margin, Inequality (4) guarantees near-perfect linear speedup, just like Inequality (3). The only difference is that the "sufficient margin" must also overcome the constant hidden by the O.

For a nondeterministic computation, 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 we shall show in Section 6, however, PBFS provides a bound of the form

$$T_P \le \text{Work}(A_1)/P + O(\lambda \cdot \text{Span}(A_1)) , \qquad (5)$$

where A_1 is the dag of a serial execution and λ is a function of the input size. For nondeterministic computations satisfying Inequality (5), we can define the ***effective parallelism*** as $\text{Work}(A_1)/\lambda \cdot \text{Span}(A_1)$. 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 guarantee to attain near-perfect 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 that are required to combine reducer views is proportional to the number of steals.

3. THE PBFS ALGORITHM

PBFS uses ***layer synchronization*** [3, 35] 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 $L_d \subset V(G)$ of vertices at distance d from v_0. Thus, we have $L_0 = \{v_0\}$. Each iteration processes layer L_d by checking all the neighbors of vertices in L_d for those that should be added to L_{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 constant amount GRAINSIZE of granularity) of the elements from bag_1, and put them into a new bag_2.

As we shall see in Section 4, 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.

```
PBFS$(G, v_0)$
1   parallel for each vertex $v \in V(G) - \{v_0\}$
2       $v.dist = \infty$
3   $v_0.dist = 0$
4   $d = 0$
5   $L_0 =$ BAG-CREATE()
6   BAG-INSERT$(L_0, v_0)$
7   while $\neg$BAG-IS-EMPTY$(L_d)$
8       $L_{d+1} =$ new reducer BAG-CREATE()
9       PROCESS-LAYER(revert $L_d, L_{d+1}, d$)
10      $d = d + 1$
```

```
PROCESS-LAYER(in-bag, out-bag, d)
11  if BAG-SIZE(in-bag) < GRAINSIZE
12      for each $u \in$ in-bag
13          parallel for each $v \in Adj[u]$
14              if $v.dist == \infty$
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.

After initialization, PBFS begins the **while** loop in line 7 which iteratively calls the auxiliary function PROCESS-LAYER to process layer L_d for $d = 0, 1, \ldots, D$, where D is the diameter of the input graph G. To process a layer *in-bag*, PROCESS-LAYER uses parallel divide-and-conquer, producing an the next layer *out-bag*. For the recursive case, line 18 splits *in-bag*, removing half its elements and placing them in *new-bag*. The two halves are processed recursively in parallel in lines 19–20.

This recursive decomposition continues until *in-bag* has fewer than GRAINSIZE elements, as tested for in line 11. Each vertex u in *in-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 — $v.dist == \infty$ — then line 15 sets $v.dist = d + 1$ and line 16 inserts v into the level-$d+1$ bag.

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 the level-$d+1$ bag $L_{d+1} = out\text{-}bag$ in line 16, but inserting multiple copies of v into L_{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 8, the amount of extra work is small, because the race is rarely actualized.

The other race occurs due to parallel insertions of vertices into $L_{d+1} = out\text{-}bag$ in line 16. We employ the reducer functionality to avoid this race by making L_{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'll see in Section 4, is as efficient as pushing it onto a FIFO as is done in serial BFS.

```
15.1  set = FALSE
15.2  if TRY-LOCK(v)
15.3      if $v.dist == \infty$
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.

Unfortunately, we are not able to analyze PBFS due to unstructured nondeterminism created by the benign race, but we can analyze 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, in which case it will set $v.dist$ $d+1$ 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 in the case 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 are 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 in this tree is an element that contains two pointers to children. The root of the tree has only one child, which is a complete binary tree of the remaining elements.

Two pennants A and B of size 2^k can be combined to form a pennant of size 2^{k+1} using the following PENNANT-UNION procedure, which is illustrated in Figure 6.

```
PENNANT-UNION(A, B)
1  Modify the root of A so that its second child is the child of B.
2  Modify the root of B so that its only child is the root of A.
3  return the root of B as the root of the new pennant.
```

This operation executes $O(1)$ time.

The function PENNANT-SPLIT performs the inverse operation of PENNANT-UNION. We assume that the input pennant contains at least 2 elements.

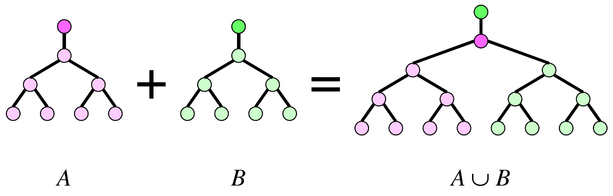


Figure 6: Two pennants, each of size 2^k, can be unioned in constant time to form a pennant of size 2^{k+1}.

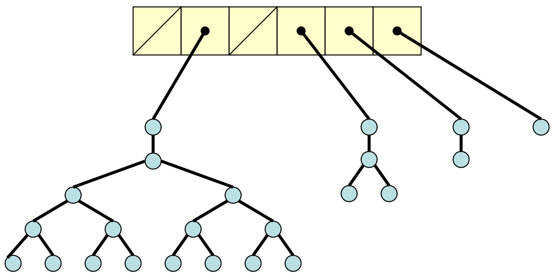


Figure 7: A bag with $23 = 010111_2$ elements.

```
PENNANT-SPLIT(A)
1  Set the root of a new pennant B to be the root of A.
2  Set the root of A to be the only child of B.
3  Set the only child of B to be the second child of A.
4  Remove the second child of A.
5  return the root of B.
```

Each of the pennants A and B now contain half the elements. Like PENNANT-UNION, PENNANT-SPLIT operates in $O(1)$ time.

Bags

A ***bag*** is a collection of different-sized pennants. PBFS represents a bag S using a fixed-size array $S[0\,..\,r]$, where 2^{r+1} exceeds the maximum number of elements ever stored in a bag. Each entry $S[k]$ contains either a null pointer or a pointer to a pennant of size 2^k. Figure 7 illustrates a bag containing 23 elements. Implementing BAG-CREATE simply allocates space for a fixed-size array of null pointers, which takes $\Theta(r)$ time. This bound can in fact be improved to $O(1)$ by maintaining a pointer to the largest nonempty index of the array.

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 of size 1. We then insert x using the following method for inserting a pennant x_k into *bag*.

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

The analysis of BAG-INSERT parallels 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 $\text{FA}(x,y,z)$ 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      (S_1[k], y) = FA(S_1[k], S_2[k], y)
```

Because every PENNANT-UNION operation takes constant time, computing the value of $\text{FA}(x,y,z)$ also takes constant time. To compute all entries in the resulting bag takes $\Theta(r)$ time. This algorithm can be improved slightly by keeping track of the sizes of bags and iterating only up to the high-order "bit" of the binary representation of the bag size.

The BAG-SPLIT function operates like an arithmetic right shift:

```
BAG-SPLIT(S_1)
 1  S_2 = BAG-CREATE()
 2  y = S_1[0]
 3  for k = 1 to r
 4      if S_1 ≠ NULL
 5          S_2[k-1] = PENNANT-SPLIT(S_1[k])
 6          S_1[k-1] = S_1[k]
 7          S_1[k] = NULL
 8  if y ≠ NULL
 9      BAG-INSERT(S_1, y)
10      return S_2
```

Because PENNANT-SPLIT takes constant time, each loop iteration in BAG-SPLIT takes constant time. Consequently, the asymptotic runtime of BAG-SPLIT is $O(r)$.

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 = 64. Second, in addition to the array of pointers to pennants, 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 bag portion 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 insert operations does a BAG-INSERT trigger the insertion of the now-full hopper into the bag portion of the data structure.

Third, when unioning two bags S_1 and S_2, BAG-UNION first checks 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. REDUCERS

This section defines reducer hyperobjects formally and introduces an extension to the multithreaded dag model that incorporates them. We characterize the behavior of a reducer during an execution in terms of a "reducer string," which specifies the inherent nondeterminism.

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 an identity e. From a programming perspective, the set T is a base type that 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.

To specify the nondeterministic behavior encapsulated by reducers precisely, let us define the execution of a multithreaded program in terms of its set A of executed strands. We assume that the functions for a reducer hyperobject — REDUCE, CREATE-IDENTITY, and UPDATE — execute only serial code. We model each invocation of one of these functions as a single strand which contains the instructions of the function but no other instructions that might be in series before or after. The other strands can be diced up arbitrarily in any convenient manner.

During execution, the runtime system may from time to time invoke CREATE-IDENTITY and REDUCE functions to maintain reducer views. We call the CREATE-IDENTITY and REDUCE strands so invoked ***init strands*** and ***reduce strands***, both of which are ***callback strands***. The other strands are ***user strands***. From the programmer's perspective, the callback strands are invoked invisibly by the runtime system (if REDUCE is associative and there are no determinacy races), and thus his or her understanding of the program is based only on the user strands.

We define the ***user dag*** $\widehat{A}$ in the same manner that an ordinary multithreaded dag is defined: spawns generate nodes with out-degree 2, syncs create nodes with in-degree greater than 1, etc. We define the ***walk*** of $\widehat{A}$ as the list of user strands encountered during a depth-first search of the dag which visits spawned children before continuations and backtracks at sync nodes until all the strands entering the sync have been visited.

If the execution is race free and the REDUCE function associative, the final value of the reducer is the value that would be obtained if the updates were performed in the serial order given by the walk of $\widehat{A}$. The program may have races and REDUCE may not be truly associative, however, and thus characterizing the full spectrum of behaviors of the computation is somewhat more complex.

Recall that when a function spawns, the spawned child inherits the parent's view of the reducer, but the continuation's view is chosen nondeterministically. On the one hand, it may receive a fresh identity view, causing a later REDUCE to combine the views sometime before the **sync** statement that matches the **spawn**. Alternatively, it may appropriate an existing view that reflects prior updates.

We can specify which of these alternatives occurs at each point in the computation using a ***hyperstring***, which consists of the strands of $\widehat{A}$ in walk order into which the callback strands have been embedded. The hyperstring distinguishes between two types of user strand: ***update strands***, which query or modify the state of the reducer, and ***nonupdate strands***, which do not. For the hyperstring to be ***legal***, the embedding must satisfy the following grammar with starting nonterminal X:

$$\begin{aligned}
Z &\rightarrow \langle \textit{update strand} \rangle \mid \langle \textit{nonupdate strand} \rangle \\
Y &\rightarrow \langle \textit{nonupdate strand} \rangle \mid Y \ \langle \textit{nonupdate strand} \rangle \\
X_i &\rightarrow \langle \textit{init strand} \rangle \mid Y \ \langle \textit{init strand} \rangle \\
X &\rightarrow Z \mid X\ Z \mid (X, X_i\ X)\ \langle \textit{reduce strand} \rangle \ .
\end{aligned}$$

In addition, the production $X \rightarrow (X, X_i\ X)\ X_R$ describes in reverse polish notation the functional operation of a REDUCE strand.

The hyperstring describes precedence relations between the user strands. Consider the parse tree for some hyperstring, and consider the rule applied at each internal node in this parse tree. If this rule matches $Y\ \langle nonupdatestrand \rangle$ or $Y\ \langle initstrand \rangle$, then all strands under Y precede the nonupdate strand or the init strand, respectively. Similarly, if the rule matches $X\ Z$, then all strands under X precede the strand under Z. If the rule matches $X \rightarrow (X, X_I\ X)\ X_R$, let X_1 represent the first generated X and X_2 represent the second generated X. The strands under X_1 must precede those under X_R, and the strands under X_I must precede the strands under X_2, which must precede the strands under X_R. The strands under X_1 may occur in parallel with those under X_I or X_2.

We assert that the Cilk++ work-stealing scheduler uses reducers in a way that is consistent with their hyperstring semantics. The proof of this is complicated, however, and is omitted from this draft.

A ***legal schedule*** Σ of a computation A and hyperstring H is a list of all the instructions in all the strands of A such that

- the chain of instructions within any strand $i \in A$ form a subsequence of Σ;
- if $i \in A_U$ precedes $j \in A_U$ in A_U, then all the instructions of i precede all the instructions of j in Σ;
- if $i \in A$ precedes $j \in A$ according to H, then all the instructions of i precede all the instructions of j in Σ.

The semantics of a computation A are given by executing the instructions in order from a legal schedule Σ of A with a legal hyperstring H.

6. ANALYSIS OF PROGRAMS WITH NONCONSTANT-TIME REDUCERS

This section provides a framework for analyzing programs such as PBFS that use reducers that execute in more than constant time. We define a "performance dag" to model the scheduling costs of a computation A using a work-stealing scheduler. We show that the span of A can be bounded as $\text{Span}(A) = O(\tau\text{Span}(\widehat{A}))$, where τ is the worst-case cost of any REDUCE or CREATE-IDENTITY, and $\widehat{A}$ is the user dag for the computation. We show that the work of A can be bounded as $\text{Work}(A) \le \text{Work}(\widehat{A}) + O(\tau^2 P\text{Span}(\widehat{A}))$.

The ***performance dag*** for a computation A with user dag $\widehat{A}$ is the dag A which augments the user dag $\widehat{A}$ as follows:

- the edges consist of the strands of A;
- all the reduce strands that must execute before a given **sync** are inserted in series immediately after the corresponding sync node;
- each init strand is inserted immediately before the corresponding update strand that caused the CREATE-IDENTITY callback.

The following lemma shows that the running time of the computation can be bounded in terms of the work and span of the performance dag.

LEMMA 1. *Consider the execution of a computation A on a parallel computer with P processors using a work-stealing scheduler. Then, the expected running time is* $\text{Work}(A)/P + O(\text{Span}(A))$. *Moreover, the total work involved in joining strands is* $O(\tau P \cdot \text{Span}(A))$.

PROOF. The proof follows those of [7] and [16], with some salient differences. As in [7], we use a delay-sequence argument, but we base it on the performance dag. We also employ a locking protocol similar to that in [16], but we modify it so that it will only hold locks for a constant amount of time, rather than for the duration of a REDUCE operation.

In the normal delay-sequence argument, there is 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 only 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 in a strand 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 at the top 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. After P steals, there is a constant probability that the span of the dag corresponding to the computation that remains to be executed is reduced by 1. That suffices to ensure that the expected number of steals is $O(P\text{Work}(A))$, and 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 the sync nodes before which they must actually execute, it can be the case that some instructions in a reducer strand execute before all of its predecessors complete. That is okay. The main property is that if an instruction is critical, it has a $1/P$ of being executed, and that P steals have a constant expectation of reducing the span of the dag that remains to execute. The crucial observation is that if an instruction in a reduce strand is critical, then it means that 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.

The second issue to deal with is the locking protocol used to synchronize workers when they attempt to call REDUCE. The algorithm in [16] uses an intricate protocol to avoid long waits on locks, but it assumes that reducing takes only constant time.

To support arbitrary cost REDUCE functions, we modify the locking protocol in [16] as follows.[5] Let F be the so-called full frame we are eliminating, and let $F.p$ be the parent full frame of F in the steal tree. Define the "right hypermap" for F to be F's RIGHT hypermap. Define the "parent hypermap" for F to be $F.p$'s CHILDREN hypermap if $F.p$ is the parent of F in the spawn tree, or $F.p$'s RIGHT hypermap if $F.p$ is the left sibling of F in the spawn tree.

1. Acquire both abstract locks for F.
2. Examine F's right hypermap R and parent hypermap P.
3. If all reducers in both R and P contain identity values, place U into $F.p$ in the place of P, and eliminate F.
4. Otherwise:
    1. Extract the reducer views from the right and parent hypermaps.
    2. Replace these reducers with identity reducers.
    3. Release the abstract locks for F.
    4. For all reducers x, $U(\texttt{x}) = R(\texttt{x}) \otimes U(\texttt{x}) \otimes P(\texttt{x})$.
    5. Repeat this locking protocol.

We want to analyze the work required to perform all eliminations using this locking protocol. Note that steps 2 through 4-3 all take at most a constant amount of work, and therefore each abstract lock is held for a constant amount of time. The abstract lock acquired in step 1 is acquired in the fashion described in [16].

The analysis of the time spent waiting to acquire an abstract lock in this locking protocol follows the analysis of the locking protocol in [16]. The key issue in the proof is to show that the time for the ith abstract lock acquisition by some worker w is independent of the time for w's jth lock acquisition for $i < j$. We prove this by considering some other worker v which is also performing abstract lock acquisitions. In particular, we consider the effect of v on the delay of the ith and jth lock acquisitions.

After some simultaneous lock acquisition by w and v, if either w or v succeed in eliminating their frame, then all future interaction between their abstract lock acquisitions is independent of this lock acquisition. Therefore, we need only consider the case where both w and v fail to eliminate their frame.

First, we show that v delaying w's jth lock acquisition is independent of v delaying w's ith lock acquisition. Suppose v delays w's ith lock acquisition. After w's ith lock acquisition, 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 in acquiring and releasing their abstract locks. For v to delay w's jth lock acquisition, 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 per edge, regardless of the previous interference event. Therefore, the event that v delays w a second time is independent of the first event.

Next, we show that v delaying w's jth lock acquisition is independent of v failing to delay w's ith lock acquisition. Suppose v does not delay w's ith lock acquisition. For v to delay w's jth lock acquisition, a chain of dependencies must form from a one of w's abstract lock to one of v's abstract locks after w performs its ith lock acquisition. Forming such a dependency chain requires every edge

[5]We regret that our nomenclature assumes the reader is familiar with [16]. The final version of the paper will contain a description that stands on its own.

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's ith lock acquisition. Therefore, for all workers $v \neq w$, the probability that v delays w's jth lock acquisition is independent of whether or not v delayed w's ith lock acquisition. Consequently, each subsequent lock acquisition by some worker is independent of all previous lock acquisitions. Because every subsequent lock acquisition for some worker is independent, the analysis of the time spent waiting for an abstract lock follows that of [16], which produces the time bounds stated in the lemma.

To bound the time spent joining strands, we first bound the number of acquisitions performed in this protocol. Since each steal creates a frame in the steal tree that must be eliminated, the number of abstract lock acquisitions to obtain is at least as large as the number of steals. In this locking protocol, the elimination of a frame may force the parent and the child of that frame to repeat the locking protocol. Therefore, each elimination increases the number of necessary abstract lock acquires by at most 2. Thus, if M is the number of successful steals, the number of elimination attempts that must be performed is no more than $3M$.

Finally, we bound the total work spent joining strands using this protocol. Each elimination attempt requires $O(1)$ time to acquire a lock and perform the necessary operations while the lock is held. Each failed elimination attempt triggers at most two REDUCE operations, each of which takes τ 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 $O(\tau P \cdot \text{Span}(A))$. □

The following two lemmas bound the work and span of the performance dag in terms of the span of the user dag.

LEMMA 2. *Consider a computation A with user dag $\widehat{A}$. The span of A is $O(\tau\text{Span}(\widehat{A}))$.*

PROOF. Each steal that occurs in the execution of A may force one CREATE-IDENTITY, and by the proof of Lemma 1, each steal may also force at most 4 REDUCE operations to execute in the performance dag. Each spawn in $\widehat{A}$ provides an opportunity for a steal to occur. Consequently, each spawn operation in A may increase the size of the dag by $O(\tau)$ in the worst case.

Consider a critical path in the performance dag of A, and let $\widehat{p}$ be the corresponding path in $\widehat{A}$. Suppose k steals occur along $\widehat{p}$. The length of that corresponding path in the performance dag is at most $5k\tau + |\widehat{p}| \leq 5\text{Span}(\widehat{A})\tau + |\widehat{p}| \leq 6\tau\text{Span}(\widehat{A})\tau$. Therefore, $\text{Span}(A) = O(\tau\text{Span}(\widehat{A}))$. □

LEMMA 3. *Consider a computation A with user dag $\widehat{A}$. The work of A is at most* $\text{Work}(\widehat{A}) + O(\tau^2 P\text{Span}(\widehat{A}))$.

PROOF. First, note that all computation modeled in the user dag appears in the performance dag. Therefore, to bound the work of A in terms of $\widehat{A}$, we must add in the work performing REDUCE and CREATE-IDENTITY performed due to steals. By Lemma 1, the total work in joining stolen strands is $O(\tau P \cdot \text{Span}(A))$. Similarly, each steal may force a call to CREATE-IDENTITY, and by the analysis of steals from [7] the total work spent performing CREATE-IDENTITY operations due to steals is $O(\tau P \cdot \text{Span}(A))$. The total work of A is therefore at most $\text{Work}(\widehat{A}) + O(\tau P\text{Span}(A))$. By Lemma 2, the total work of A is therefore at most $\text{Work}(\widehat{A}) + O(\tau^2 P\text{Span}(\widehat{A}))$. □

7. ANALYSIS OF PBFS

Using the results in Section 6, we can bound the expected running time of the locked version of PBFS. First we bound the work and span of the user dag for PBFS. We then apply Lemmas 3 and 2 to bound the work and span of PBFS's parallel execution, and finally we apply Lemma 1 to get the expected running time of PBFS.

LEMMA 4. *Suppose PBFS is run on a connected input graph $G = (V,E)$. The total work in PBFS's user dag is $O(V+E)$, and the total span of PBFS's user dag is $O(D\lg(V/D) + D\lg(\text{max-degree}(V)))$.*

PROOF. In each layer, PBFS evaluates every vertex v in that layer exactly once, and therefore PBFS checks every vertex u in v's adjacency list exactly once. In the locked 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 $O(D\lg V)$. 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. Because a steal will reset 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. Therefore 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 number of vertices in the dth layer. Since the series of syncs PBFS performs form a mirror of the split tree, the computation for dividing and the vertices in a layer and combining 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 $O(\lg(\text{max-degree}(V)))$. 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 $O(\lg V_d + \lg V_{d+1} + \lg \text{max-degree}(V))$. Summing this quantity over all layers in G, we find that the maximum span for PBFS is $O(D\lg(V/D) + D\lg(\text{max-degree}(V)))$. □

THEOREM 5. *Consider the parallel execution of PBFS on P processors. The expected running time of PBFS is $(V+E)/P + O(D\lg^2(V/D)(\lg(V/D) + \lg(\text{max-degree}(V))))$. Furthermore, if* $\text{max-degree}(V)$ *is bounded, then the expected running time of PBFS is $(V+E)/P + O(D\lg^3(V/D))$.*

PROOF. To maximize the cost of all CREATE-IDENTITY and REDUCE operations in PBFS, the worst-case cost of each of these operations is $O(\lg(V/D))$. By Lemma 2 the span of PBFS is $O(D\lg(V/D)(\lg(V/D) + \lg(\text{max-degree}(V)))$, and by Lemma 3 the work of PBFS is $V + E + O(DP\lg^2(V/D)(\lg(V/D) + \lg(\text{max-degree}(V))))$. By Lemma 1 the expected running time of PBFS is $(V+E)/P + O(D\lg^2(V/D)(\lg(V/D) + \lg(\text{max-degree}(V)))$. If $\text{max-degree}(V)$ is bounded, then the expected running time of PBFS is $(V+E)/P + O(D\lg^3(V/D))$. □

8. IMPLEMENTATION

We implemented both the PBFS algorithm and an optimized FIFO-based serial BFS algorithm in Cilk++ and compared their performance on a suite of benchmark graphs. In particular, we examined the performance of PBFS without locks.

As can be seen from Figure 8 in Appendix A, PBFS demonstrates excellent performance characteristics on these benchmark graphs. For many graphs PBFS run serially is nearly as fast or faster than serial BFS. Furthermore, the additional work PBFS performed due to its benign race is typically very small compared to the size of the graph.[6]

9. CONCLUSION

To be written.

10. REFERENCES

[1] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 119–129, New York, NY, USA, June 1998. ACM Press.

[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 *35th International Conference on Parallel Processing (ICPP)*, Columbus, Ohio, Aug. 2006.

[4] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3), Mar. 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. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

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

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

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

[9] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving large, irregular graph problems using adaptive work-stealing. *Parallel Processing, International Conference on*, 0:536–545, 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, 1994.

[12] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 1–10. ACM Press, 1990.

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

[14] P. A. Emrath and D. A. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 89–99, Madison, Wisconsin, May 1988.

[15] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–11, Newport, Rhode Island, June22–25 1997.

[16] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proceedings of the Twenty-First Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 79–90, Calgary, Canada, Aug. 2009. ACM. Won Best Paper award.

[17] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

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

[19] R. L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 45:1563–1581, Nov. 1966.

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

[21] G. L. S. Jr. Making asynchronous parallelism safe for the world. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 218–231. ACM Press, 1990.

[22] R. E. Korf and P. Schultze. Large-scale parallel breadth-first search. In *AAAI'05: Proceedings of the 20th national conference on Artificial intelligence*, pages 1380–1385. AAAI Press, 2005.

[23] C. Y. Lee. An algorithm for path connection and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, 1961.

[24] C. E. Leiserson. The Cilk++ concurrency platform. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 522–527. ACM, July 2009.

[25] J. Leskovec, D. Chakrabarti, J. M. Kleinberg, and C. Faloutsos. Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication. In *Conference on Principles and Practice of Knowledge Discovery in Databases*, pages 133–145, 2005.

[26] J. B. Lubos, L. Brim, and J. Chaloupka. Parallel breadth-first search ltl model-checking. In *In 18th IEEE International Conference on Automated Software Engineering (ASEâĂŹ03*, pages 106–115. IEEE Computer Society, 2003.

[27] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing'91*, pages 24–33. IEEE Computer Society Press, 1991.

[28] E. F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[29] R. H. B. Netzer and B. P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[30] I. Nudler and L. Rudolph. Tools for the efficient development of efficient parallel programs. In *Proceedings of the First Israeli Conference on Computer Systems Engineering*, May 1986.

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

[32] Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science. *Cilk 5.4.2.3 Reference Manual*, Apr. 2006.

[33] 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: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 25, Washington, DC, USA, 2005. IEEE Computer Society.

[34] F. Zaoui. *Large Eletrical Power Systems Optimization Using Automatic Differentiation*. Springer Berlin Heidelberg, 2008.

[35] Y. Zhang and E. A. Hansen. Parallel breadth-first heuristic search on a shared-memory architecture. In *AAAI-06 Workshop on Heuristic Search, Memory-Based Heuristics and Their Applications*, July 2006.

[6]We regret that we were unable for this draft to conduct the complete set of performance tests or measure the performance of PBFS with locks on our newest hardware.

APPENDIX

A. EXPERIMENTAL RESULTS

We chose several real-world graphs for testing these algorithms. Kkt_power, Cage14, and Cage15 are 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 [18]. The RMat23 matrix [25], which models scale-free graphs, is 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.

B. WORKED EXAMPLES

Reducers

The basic idea of a reducer can be understood from the example of a series of additive updates to a value x:

1 $x = 10$
2 $x{++}$
3 $x += 3$
4 $x += -2$
5 $x += 6$
6 $x{--}$
7 $x += 4$
8 $x += 3$
9 $x{++}$
10 $x += -9$

When executed serially, the resulting value is $x = 16$. Alternatively, the sequence could be executed as follows:

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

In this case, the computations for x and x' can proceed in parallel with an additional step to combine the results at the end. Of course, the computation could be split anywhere along the way, and the final result would be the same as long as x' is initialized to 0, the identity for $+$, which is an associative operator.

Hyperstrings

As an example, for the computation in Figure 3, a hyperstring describing the nondeterministic choices for an execution might be

$$(((abcdef,!ghijklm)\otimes no,!pqrstu)\otimes,!vwxy)\otimes z$$

where "!" represents an init strand and "$\otimes$" represents a reduce strand. To interpret this string, suppose we have a particular reducer object h. In this string, first the user strands $abcdef$ form a view h_1 of the reducer, and then h_1 is reduced into view h_2 — the view formed from a newly initialized reducer by $ghijklm$ — to form view h_3. Next, h_3 is updated according to the update strands in no to form h_4, which is subsequently reduced into h_5 — the view formed from a newly initialized reducer by $pqrstu$ — to get view h_6. Finally, h_6 is reduced into h_7 — the view formed from a newly initialized reducer by $vwxy$ — to get h_8, which is updated by z if z is an update strand.

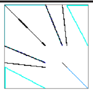
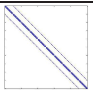
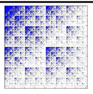
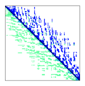
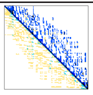
Name Description	Spy Plot	$\lvert V\rvert$ $\lvert E\rvert$ D	Work Span Parallelism	SERIAL-BFS T_1	PBFS T_1	PBFS T_4	Duplicates$_4$
Kkt_power Optimal power flow, nonlinear opt.		2.05M 12.76M 31	602.5M 37.2M 16.69	0.438	0.465	0.155	27
Grid3D200 3D 7-point finite-diff mesh		8M 55.8M 598	2,485.7M 166.5M 14.93	1.486	1.550	0.560	557
RMat23 Real-world graph model		2.3M 77.9M 8	11,821.4M 77.0M 153.61	1.226	1.181	0.425	4
Cage14 DNA electrophoresis		1.51M 27.1M 43	782.9M 26.6M 29.38	0.304	0.376	0.124	68
Cage15 DNA electrophoresis		5.15M 99.2M 50	2,802.9M 75.0M 37.35	1.256	1.494	0.486	86

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. Duplicates$_4$ refers to the average number of duplicate vertices rounded to the nearest integer that PBFS evaluated when run on 4 cores.