

**Design and Analysis of a Nondeterministic Parallel
Breadth-First Search Algorithm**

by

Tao Benjamin Schardl

Submitted to the Department of Electrical Engineering and Computer
Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2010

© Massachusetts Institute of Technology 2010. All rights reserved.

Author

Department of Electrical Engineering and Computer Science
May 21, 2010

Certified by

Charles E. Leiserson
Professor
Thesis Supervisor

Accepted by

Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

Design and Analysis of a Nondeterministic Parallel Breadth-First Search Algorithm

by

Tao Benjamin Schardl

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2010, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

I have developed a multithreaded implementation of breadth-first search (BFS) of a sparse graph using the Cilk++ extensions to C++. My 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. I 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, I 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 $P \ll (V + E)/D \lg^3(V/D)$.

Some parts of this thesis represent joint work with Professor Charles E. Leiserson.

Thesis Supervisor: Charles E. Leiserson
Title: Professor

Acknowledgments

Thanks to my advisor, Professor Charles E. Leiserson of MIT CSAIL, for his tremendous support and guidance in all aspects of this thesis work. Thanks to Aydın Buluç of University of California, Santa Barbara, who helped me obtain many of our benchmark tests. Pablo G. Halpern of Intel Corporation and Kevin M. Kelley of MIT CSAIL helped me debug PBFS's performance bugs. Matteo Frigo of Axis Semiconductor helped me weigh the pros and cons of reducers versus TLS. Thanks to the Cilk team at Intel and the Supertech Research Group at MIT CSAIL for their support. Thanks to all of my friends and family for their unending support and encouragement.

Contents

1	Introduction	11
2	Background on dynamic multithreading	15
3	The PBFS algorithm	19
4	The bag data structure	23
5	Implementation	29
6	The dag model of computation	35
7	Reducers	43
8	Analysis of programs with nonconstant-time reducers	53
9	Analysis of PBFS	69
10	Conclusion	73

List of Figures

1-1	A standard serial breadth-first search algorithm.	12
1-2	Characteristic performance of PBFS.	14
2-1	An example of the intuition behind reducers.	17
3-1	The PBFS algorithm.	20
3-2	Modification to the PBFS algorithm to resolve the benign race.	21
4-1	Pseudocode for PENNANT-UNION	24
4-2	Illustration of PENNANT-UNION operation.	24
4-3	Pseudocode for PENNANT-SPLIT.	24
4-4	An example bag.	25
4-5	Pseudocode for BAG-INSERT.	25
4-6	Table detailing the function $FA(x,y,z)$	26
4-7	Pseudocode for BAG-UNION.	26
4-8	Pseudocode for BAG-SPLIT.	27
5-1	Performance results for breadth-first search.	31
5-2	Multicore Murphi application speedup.	33
6-1	A dag representation of a multithreaded execution.	37
6-2	A user dag representation of a multithreaded computation without reducers.	38
7-1	A modified locking protocol for managing reducers.	46
7-2	A dag representation of a computation with a REDUCE operation executed opportunistically.	50

Chapter 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 [27] while studying the problem 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, 10, 22, 26, 33, 34]. 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 thesis, we shall see 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 [11, 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 1-1 gives a variant of the classical serial algorithm [11, Section 22.2] for computing BFS, which uses a FIFO queue as an auxiliary data structure. The FIFO can be

```

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

```

Figure 1-1: 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 .

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 *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

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

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

as fast as a FIFO, even when constant factors are considered. In addition, bags can be split and unioned efficiently.

I have implemented a parallel BFS algorithm in Cilk++ [21, 24]. The *PBFS* algorithm, which employs bags instead of a FIFO, uses the “reducer hyperobject” [15] feature of Cilk++. My 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 1-2 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 [32]. 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. My 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

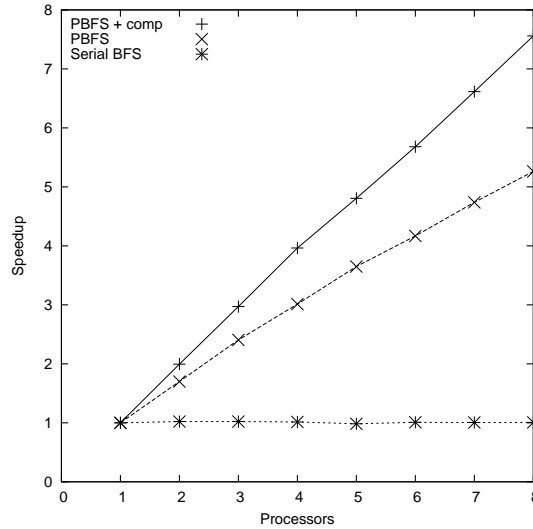


Figure 1-2: The performance of PBFS for the Cage15 graph showing speedup curves for serial BFS, PBFS, and a variant of PBFS where the computational intensity has been artificially enhanced and the speedup normalized.

that employ reducers. This thesis leaves 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 as follows. First, we shall examine the basic PBFS algorithm and analyze it empirically. Chapter 2 provides background on dynamic multithreading. Chapter 3 described the basic PBFS algorithm, and Chapter 4 describes the implementation of the bag data structure. Chapter 5 presents our empirical studies.

Second, we shall create a theoretical framework for analyzing programs with non-constant time reducers, and apply this framework to analyze PBFS. Chapter 6 provides background on the theory of dynamic multithreading. Chapter 7 gives a formal model for reducer behavior, and Chapter 8 develops a theory for analyzing programs that use reducers. Chapter 9 emplys this theory to analyze the theoretical performance of PBFS.

Finally, Chapter 10 concludes with a discussion of thread-local storage as an alternative to reducers.

Chapter 2

Background on dynamic multithreading

This chapter overviews the key attributes of dynamic multithreading. The PBFS software is implemented in Cilk++ [15, 21, 24], which is a linguistic extension to C++ [30], but most of the vagaries of C++ are unnecessary for understanding the issues. Thus, I describe Cilk-like pseudocode, as is exemplified in [11, Ch. 27], which the reader should find more straightforward than real code to understand and which can be translated easily to Cilk++. In this chapter we shall review the pseudocode keywords for creating fork-join parallel programs. We shall also see the basic intuition behind the reducer hyperobject.

Multithreaded pseudocode

The linguistic model for multithreaded pseudocode in [11, Ch. 27] follows MIT Cilk [16, 31] and Cilk++ [21, 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 *continuation* — the code that immediately follows the spawn — may execute in parallel with the spawned 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* [15], 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 Chapter 3, PBFS uses a reducer called a “bag,” which implements an unordered set and supports fast unioning as its **REDUCE** operator.

Figure 2-1 illustrates the basic idea of a reducer. The example involves a series of additive updates to a variable x . When the code in Figure 2-1(a) is executed serially, the resulting value is $x = 16$. Figure 2-1(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 2-1(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 2-1(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 **REDUCE** 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

<pre> 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 </pre>	<pre> 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' </pre>	<pre> 1 x = 10 2 x++ 3 x += 3 x' = 0 4 x' += -2 5 x' += 6 6 x'-- x'' = 0 7 x'' += 4 8 x'' += 3 9 x''++ 10 x'' += -9 x += x' x += x'' </pre>
(a)	(b)	(c)

Figure 2-1: 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'' .

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 Chapter 7.

Chapter 3

The PBFS algorithm

PBFS uses *layer synchronization* [3, 34] 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:

- $bag = \text{BAG-CREATE}()$: Create a new empty bag.
- $\text{BAG-INSERT}(bag, x)$: Insert element x into bag .
- $\text{BAG-UNION}(bag_1, bag_2)$: Move all the elements from bag_2 to bag_1 , and destroy bag_2 .
- $bag_2 = \text{BAG-SPLIT}(bag_1)$: Remove half (to within some constant amount `GRAIN-SIZE` of granularity) of the elements from bag_1 , and put them into a new bag bag_2 .

As Chapter 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 3-1. 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 $d = 0, 1, \dots, D$, where D is the diameter of the input graph G . To process $V_d = in\text{-}bag$, `PROCESS-LAYER` uses parallel divide-and-

```

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   $V_0 = \text{BAG-CREATE}()$ 
6   $\text{BAG-INSERT}(V_0, v_0)$ 
7  while  $\neg \text{BAG-IS-EMPTY}(V_d)$ 
8       $V_{d+1} = \text{new reducer BAG-CREATE}()$ 
9       $\text{PROCESS-LAYER}(\text{revert } V_d, V_{d+1}, d)$ 
10      $d = d + 1$ 

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

```

Figure 3-1: 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.

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

```

15.1  if TRY-LOCK( $v$ )
15.2      if  $v.dist == \infty$ 
15.3           $v.dist = d + 1$ 
15.4          BAG-INSERT( $out-bag, v$ )
15.5          RELEASE-LOCK( $v$ )

```

Figure 3-2: Modification to the PBFS algorithm to resolve the benign race.

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 $V_{d+1} = out-bag$ 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 Chapter 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 $V_{d+1} = out-bag$. 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 Chapter 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 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 3-2. In the code, the call TRY-LOCK(v) in line 15.1 attempts to acquire a

lock on the vertex v . If it is successful, we proceed to execute lines 15.2–15.5. 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 15.2 avoid the overhead of lock acquisition when $v.dist$ has already been set.

Chapter 4

The bag data structure

This chapter describes the bag data structure for implementing a dynamic unordered set. We first examine an auxiliary data structure called a “pennant.” We then see how bags can be implemented using pennants, and we provide algorithms for BAG-CREATE, BAG-INSERT, BAG-UNION, and BAG-SPLIT. Finally, we consider 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 PENNANT-UNION function described in Figure 4-1, which is illustrated in Figure 4-2.

The function PENNANT-SPLIT, whose pseudocode is given in Figure 4-3, performs the inverse operation of PENNANT-UNION in $O(1)$ time. The PENNANT-SPLIT function assumes that the input pennant x contains at least 2 elements. After this function, each of the pennants x and y contain half of the elements.

PENNANT-UNION(x, y)

- 1 $y.right = x.left$
- 2 $x.left = y$
- 3 **return** x

Figure 4-1: Pseudocode for PENNANT-UNION

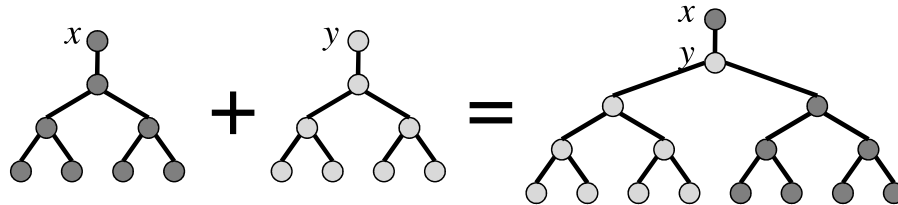


Figure 4-2: Two pennants, each of size 2^k , can be unioned in constant time to form a pennant of size 2^{k+1} .

PENNANT-SPLIT(x)

- 1 $y = x.left$
- 2 $x.left = y.right$
- 3 $y.right = \text{NULL}$
- 4 **return** y

Figure 4-3: Pseudocode for PENNANT-SPLIT.

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 4-4 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.

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 method shown in Figure 4-5.

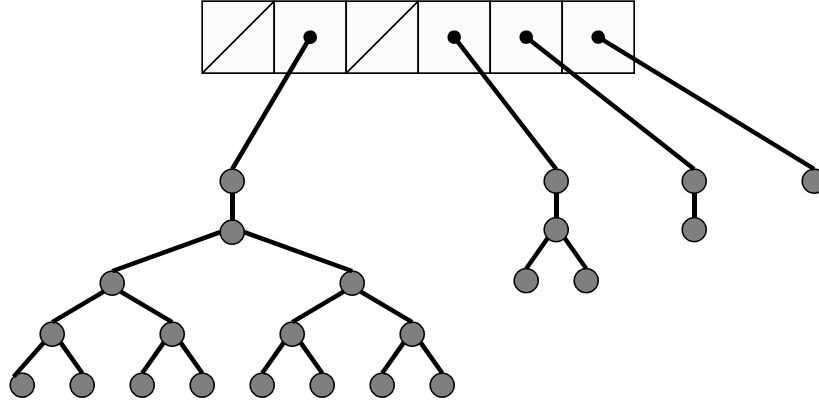


Figure 4-4: A bag with $23 = 010111_2$ elements.

```

BAG-INSERT( $S, x$ )
1   $k = 0$ 
2  while  $S[k] \neq \text{NULL}$ 
3       $x = \text{PENNANT-UNION}(S[k], x)$ 
4       $S[k++] = \text{NULL}$ 
5   $S[k] = x$ 

```

Figure 4-5: Pseudocode for BAG-INSERT. This function assumes the input pennant x has unit size.

The analysis of BAG-INSERT mirrors the analysis for incrementing a binary counter [11, 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 table in Figure 4-6 details the full-adder function $\text{FA}(x, y, z)$ in which (s, c) is computed from (x, y, z) . Using this full-adder function, BAG-UNION can be implemented as shown in Figure 4-7.

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 backbone of the resulting

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)

Figure 4-6: Table detailing the function $FA(x,y,z)$. This function takes three input pennants x , y , and z , each of which either has size 2^k or is empty, and merges 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. A 0 in the left-hand-side of the table designates an empty pennant, while a 1 designates a pennant with size 2^k .

```

BAG-UNION( $S_1, S_2$ )
1   $y = \text{NULL}$  // The “carry” bit.
2  for  $k = 0$  to  $r$ 
3       $(S_1[k], y) = FA(S_1[k], S_2[k], y)$ 

```

Figure 4-7: Pseudocode for BAG-UNION. This function uses the function $FA(x,y,z)$ detailed in Figure 4-6 as a subroutine.

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 is shown in Figure 4-8. This function operates like an arithmetic right shift to divide the elements in one bag evenly between two bags.

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.

```

BAG-SPLIT( $S_1$ )
1   $S_2 = \text{BAG-CREATE}()$ 
2   $y = S_1[0]$ 
3   $S_1[0] = \text{NULL}$ 
4  for  $k = 1$  to  $r$ 
5      if  $S_1[k] \neq \text{NULL}$ 
6           $S_2[k-1] = \text{PENNANT-SPLIT}(S_1[k])$ 
7           $S_1[k-1] = S_1[k]$ 
8           $S_1[k] = \text{NULL}$ 
9  if  $y \neq \text{NULL}$ 
10      $\text{BAG-INSERT}(S_1, y)$ 
11 return  $S_2$ 

```

Figure 4-8: Pseudocode for BAG-SPLIT.

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. My PBFS software implementation 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.

Chapter 5

Implementation

I implemented optimized versions of both the PBFS algorithm in Cilk++ and a FIFO-based serial BFS algorithm in C++. This chapter compares their performance on a suite of benchmark graphs. Figure 5-1 summarizes the results.

Implementation and Testing

My implementation of PBFS differs from the abstract algorithm in some notable ways. First, this implementation of PBFS does not use locks to resolve the benign races described in Chapter 3. Second, this implementation of PBFS does not use the BAG-SPLIT routine described in Chapter 4. Instead, this 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, this implementation assumes that all vertices have bounded out-degree, and indeed most of the vertices in our benchmark graphs have relatively small degree. Finally, this implementation of PBFS sets `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 5-1. `Kkt_power`, `Cage14`, `Cage15`, `Freescale1`, `Wikipedia` (as of February 6, 2007), and `Nlp-`

kkt160 are all from the University of Florida sparse-matrix collection [12]. Grid3D200 is a 7-point finite difference mesh generated using the Matlab Mesh Partitioning and Graph Separator Toolbox [17]. The RMat23 matrix [25], 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. These implementations store these graphs in a compressed-sparse-rows (CSR) format in main memory.

Results

I ran these 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 5-1 presents the performance of PBFS on eight different benchmark graphs. (The parallelism was computed using the Cilkview tool [19] and does not take into account effects from reducers.) As can be seen in Figure 5-1, 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 5-1 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, I 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. My studies indicate that the

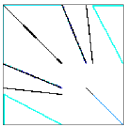
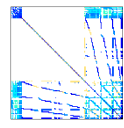
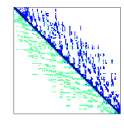

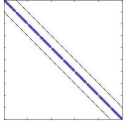
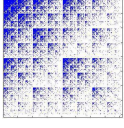
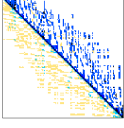
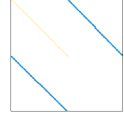
<i>Name</i>		$ V $	<i>Work</i>	SERIAL-BFS T_1
<i>Description</i>	<i>Spy Plot</i>	$ E $	<i>Span</i>	PBFS T_1
		D	<i>Parallelism</i>	PBFS T_1/T_8
Kkt_power Optimal power flow, nonlinear opt.		2.05M 12.76M 31	241M 2.3M 104.09	0.511 0.360 6.102
Freescall1 Circuit simulation		3.43M 17.1M 128	349M 2.3M 153.06	0.285 0.319 5.145
Cage14 DNA electrophoresis		1.51M 27.1M 43	390M 1.6M 246.35	0.267 0.283 5.442
Wikipedia Links between Wikipedia pages		2.4M 41.9M 460	606M 3.4M 179.02	0.918 0.738 6.833
Grid3D200 3D 7-point finite-diff mesh		8M 55.8M 598	1,009M 12.7M 79.27	1.469 1.098 4.902
RMat23 Scale-free graph model		2.3M 77.9M 8	1,049M 11.3M 93.22	1.107 0.924 6.794
Cage15 DNA electrophoresis		5.15M 99.2M 50	1,410M 2.1M 675.22	1.099 1.163 5.486
Nlpkkt160 Nonlinear optimization		8.35M 225.4M 163	3,060M 9.2M 331.57	1.286 1.463 6.096

Figure 5-1: 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.

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 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. I 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 [20], formerly of Cilk Arts and Intel Corporation, used PBFS to parallelize the Murphi model-checking tool [13]. 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 5-2, a parallel Murphi using PBFS scales well, even outperforming a version based on parallel depth-first search and attaining the relatively large speedup of 15.5 times on 16 cores.

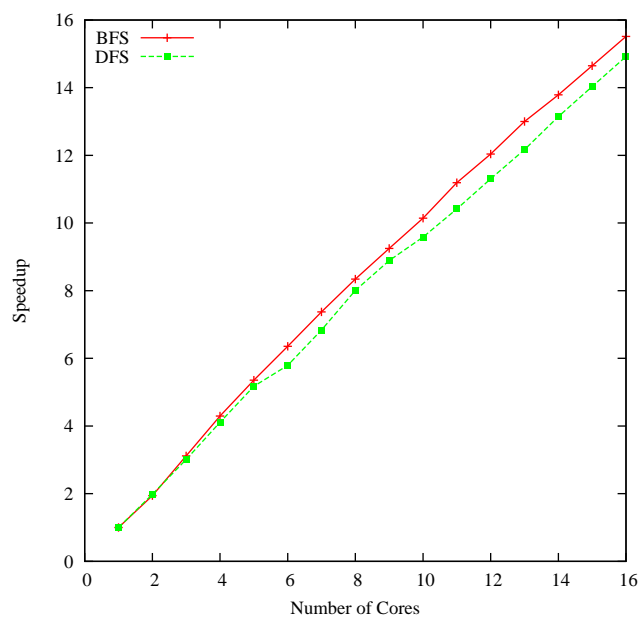


Figure 5-2: Multicore Murphi application speedup on a 16-core AMD processor [20]. Even though the DFS implementation uses a parallel depth-first search for which Cilk++ is particularly well suited, the BFS implementation, which uses the PBFS library, outperforms it.

Chapter 6

The dag model of computation

This chapter overviews the theoretical model of multithreaded computation. We shall see how a multithreaded program execution can be modeled theoretically as a dag using the framework of Blumofe and Leiserson [7], and we shall make some assumptions about the runtime environment. We shall derive a “user dag” from the dag model of computation to facilitate measuring the performance of a program with no reducers. We shall also define deterministic and nondeterministic computations. Chapter 7 describes how to extend this model to account for reducer hyperobjects.

The dag model

We 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 a multithreaded program with spawns and syncs. Although we incorporate instructions executed to manage reducers in this model, this model does not accurately represent the effect of reducers on the running time of the computation. We extend this model in Chapter 7 to account for the cost of dealing with reducers.

The dag model views the executed computation resulting from the running of a multithreaded program¹ as a *dag (directed acyclic graph)* A , where the vertex set $V(A)$ consists of *strands* — sequences of serially executed instructions containing no parallel control —

¹When I refer to the running of a program, you should generally assume that I mean “on a given input.”

and the edge set $E(A)$ represents dependencies between strands. Specifically, $V(A)$ contains all strands of two different types:

- **User strands** arising from the execution of code explicitly invoked by the programmer. We denote the set of strands of this type with V_v .
- **Runtime strands** arising from the execution of code run implicitly by the runtime system to create and reduce views of a reducer. We shall denote these runtime strands as $V_l \cup V_p$ in Chapter 7, which discusses these strands in more detail.

The edge set $E(A)$ contains edges representing two types of dependencies: parallel-control dependencies and scheduling dependencies.

The **parallel-control dependencies**² $E(A)$, denoted E_χ , represent the dependencies that are described by **spawn** and **sync** statements in the code of the multithreaded program. A strand that has 2 outgoing control dependencies is a **spawn strand**, and a strand that resumes the caller after a spawn is called a **continuation strand**. A strand with at least 2 incoming control dependencies is a **sync strand**. We assume that no continuation strand is also a sync strand. We do not represent any control dependencies on runtime strands.

The scheduling dependencies in $E(A)$, denoted E_σ , may be understood in terms of a schedule for A . A **schedule** of a computation A on a P -processor computer is a mapping $C : V(A) \rightarrow \{0, 1, \dots, P-1\} \times \mathbf{N}$ where, for each strand $u \in V(A)$, if $C(u) = (w_u, t_u)$ then worker w_u begins executing u at time step t_u . A scheduler creates a schedule for A 's execution such that all scheduling dependencies are obeyed, where, for two strands $u, v \in V(A)$, a **scheduling dependency** $(u, v) \in E_\sigma$ from u to v exists if v cannot legally execute before u has completed execution. For simplicity, we only consider a minimal set of scheduling dependencies, meaning that if $(u, v) \in E_\sigma$, then there does not exist a strand $w \in V(A)$ such that $(u, w) \in E_\sigma$ and $(w, v) \in E_\sigma$. This set E_σ of scheduling dependencies is included in the edge set $E(A)$. If a strand u depends on every strand in a set $\{v_1, v_2, \dots, v_k\} \subseteq V(A)$, we say that the last strand in $\{v_1, v_2, \dots, v_k\}$ to finish executing **enables** u .

We will often want to examine precedence relations between strands in $V(A)$ with respect to only one edge set. We therefore define two precedence operators, \succ_χ and \succ_σ as

²We shall also refer to these dependencies simply as **control dependencies**.

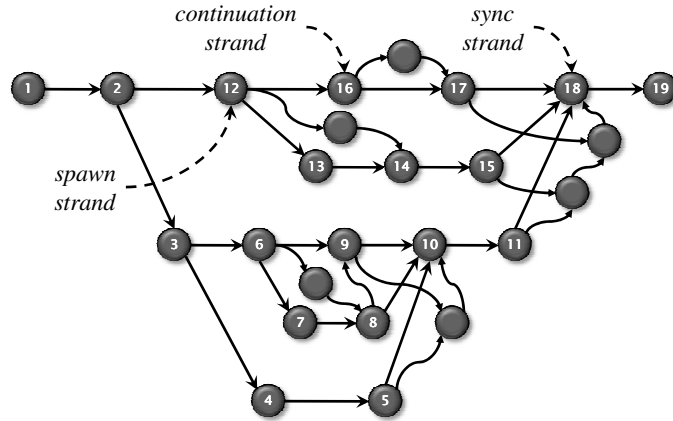


Figure 6-1: A dag representation of a multithreaded execution. Each vertex represents a strand. Straight edges represent parallel-control dependencies between strands, while curved edges represent scheduling dependencies between strands. For visual simplicity, only scheduling dependencies that differ from parallel-control dependencies are shown. Vertices with no number depict strands executed implicitly by the runtime system to manage reducers.

follows. For two strands $u, v \in V(A)$, we have $u \succ_{\chi} v$ in A if u precedes v in A according to only the edges in E_{χ} . Similarly, we have $u \succ_{\sigma} v$ in A if u precedes v in A according to only the edges in E_{σ} .

Although the scheduling dependencies are more restrictive than the parallel-control dependencies, they are related to the parallel-control dependencies by the following invariant:

Invariant 1. For two strands $u, v \in V(A)$, we have $u \succ_{\chi} v \rightarrow u \succ_{\sigma} v$.

For two strands $u, v \in V(A)$, we have $u \succ_{\chi} v \rightarrow v \not\succeq_{\chi} u$, and we have $u \succ_{\sigma} v \rightarrow v \not\succeq_{\sigma} u$. Together with Invariant 1, these implications guarantee that A is indeed a dag.

Figure 6-1 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 that calling or spawning a function terminates a strand, as does returning from a function. Thus, each strand belongs to exactly one function instantiation.

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

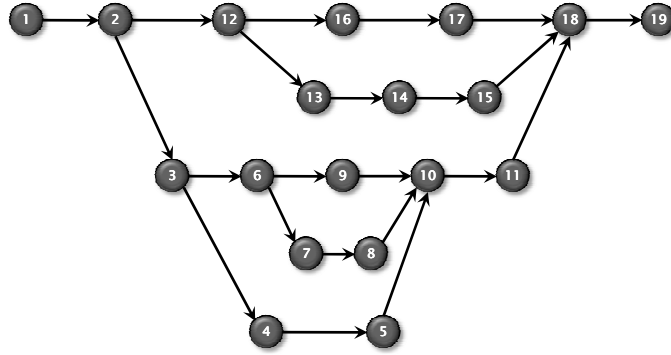


Figure 6-2: A user dag representation of a multithreaded computation without reducers. Each vertex represents a strand, and edges represent parallel-control dependencies between strands.

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.

For convenience we derive two distinct dags from a computation A .

First, to simplify the task of bounding the performance of a computation A with no reducers, we derive a *user dag* $\text{User}(A)$ from A by setting $V(\text{User}(A)) = V_0$ and setting $E(\text{User}(A)) = E_\chi$. Figure 6-2 shows an example user dag for a computation containing no reducers. The convenience of the user dag is motivated by the following two facts about the performance of A . First, Blumofe and Leiserson prove in [7] that the performance of A may be bounded by only on the control dependencies in $E(A)$. Second, since A does not contain reducers, then we have $V(A) = V_0$.

The second we derive is called a *scheduling dag* $\text{Sched}(A)$. We derive $\text{Sched}(A)$ from a computation A by setting $V(\text{Sched}(A)) = V(A)$ and setting $E(\text{Sched}(A)) = E_\sigma$. We shall use the scheduling dag in Chapter 8 in order to construct a “delay-sequence” argument on computations that use reducers.

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, 14, 18] on performance and speedup. The *work* of a dag A , denoted by $\text{Work}(A)$, is the sum of the lengths of all of the strands in $V(A)$. For a computation that does not contain reducers, this equals the sum of the lengths of all strands in $V(\text{User}(A))$. For example, consider the user dag of a reducer-free computation modeled in Figure 6-2. Assuming for simplicity that it takes unit time to execute a strand, the work for the example dag in Figure 6-2 is 19. The *span*³ of A , denoted by $\text{Span}(A)$, is the length of the longest path of control dependencies in A . For computations that do not contain reducers, this is simply the longest path in $\text{User}(A)$. Assuming unit-time strands, the span of the dag in Figure 6-2 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 [11, Ch. 27] and [24].

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

$$T_P(A) \geq \text{Work}(A)/P, \quad (6.1)$$

$$T_P(A) \geq \text{Span}(A). \quad (6.2)$$

Inequality (6.1), 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 P instructions per unit time. Inequality (6.2), called the **Span Law**, holds because no execution that respects the partial order of the dag according to parallel-control dependencies can execute faster than the longest serial chain of instructions.

We define the *speedup* of a program as $T_1(A)/T_P(A)$ — how much faster the P -processor execution is than the serial execution. Since all executions of a deterministic

³The literature also uses the terms *depth* [4] and *critical-path length* [5].

program produce the same dag A , we have that $T_1(A) = \text{Work}(A)$, and $T_\infty(A) = \text{Span}(A)$ (assuming no overhead for scheduling). Rewriting the Work Law, we obtain $T_1(A)/T_P(A) \leq P$, 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 $T_1(A)/T_\infty(A)$. The parallelism represents the maximum possible speedup on any number of processors, which follows from the Span Law, because $T_1(A)/T_P(A) \leq T_1(A)/\text{Span}(A) = \text{Work}(A)/\text{Span}(A)$. For example, the parallelism of the dag in Figure 6-2 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.

For a program that does not involve reducers, the dag model presented is sufficient for measuring the work and span of the computation. This model, however, does not represent any control dependencies involving runtime strands, although they do contribute to the span. In Chapter 7 we extend this model of computation to represent control dependencies involving runtime strands to portray their contribution to the work and span of a computation.

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 $T_P(A)$ can be bounded as

$$T_P(A) \leq \text{Work}(A)/P + O(\text{Span}(A)) , \quad (6.3)$$

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 (6.3) 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 (6.3) yields $T_P(A) \approx \text{Work}(A)/P$, and the speedup is $T_1(A)/T_P(A) \approx P$.

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 Chapter 9 shows, however, PBFS achieves

$$T_P(A) \leq \text{Work}(\text{User}(A))/P + O(\tau^2 \cdot \text{Span}(\text{User}(A))) , \quad (6.4)$$

where $\text{User}(A)$ is the user dag of A , τ is an upper bound on the time it takes to perform a REDUCE, which may be a function of the input size, and the work and span of $\text{User}(A)$ are the sum of the lengths of the strands in $V(\text{User}(A))$ and the length of the longest path in $\text{User}(A)$ respectively. (We shall formalize these concepts for computations with reducers in Chapters 7 and 8.) For nondeterministic computations satisfying Inequality (6.4), we define the *effective parallelism* as $\text{Work}(\text{User}(A))/(\tau^2 \cdot \text{Span}(\text{User}(A)))$. 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.

Chapter 7

Reducers

This chapter reviews the definition of reducer hyperobjects from Frigo *et al.* [15] and extends the dag model to incorporate them. We first formalize the semantic definition of a reducer. We then examine how Cilk implements reducers, and we propose a modified locking protocol for managing views of a reducer within the runtime system. We shall clarify the notion of a user dag for a computation with reducers, and we shall define “performance dags,” which include the strands that the runtime system implicitly invokes. Finally, we shall consider the inaccuracies in the performance dag representation of reducer operations, and we shall define a way to track the views of a reducer through a scheduling dag.

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.

In order to analyze programs that use reducers with nonconstant-time REDUCE operations, we must address the non-trivial task of representing operations on reducers within our model of computation. In particular, although the dag model of computation presented in Chapter 6 represents the CREATE-IDENTITY and REDUCE operations that the runtime system performs implicitly, it does not model any parallel-control dependencies on these strands, and therefore it fails to model the effect of these operations on the work and span. Since the runtime system performs CREATE-IDENTITY and REDUCE operations nondeterministically during the computation, how they affect the work and span of the computation is not straightforward.

We shall address this problem as follows. First, we shall examine how the runtime system implements and manages reducers. We shall then clarify the definition of a user dag for computations with reducers and show how to augment this user dag to get a “performance dag” containing the implicit operations on reducers where we would like to charge for them. The model of reducers in the performance dag does not faithfully represent where these reducer operations are performed in the schedule, however, and in order to prove that the performance dag is sufficient for accounting for reducers, we need to examine a faithful representation of reducer operations. We shall therefore show how to account for reducer operations in the scheduling dag, and we shall use the scheduling-dag representation in Chapter 8 to prove that the performance dag suffices to account for operations on reducers.

Implementation of reducers

To model and prove performance bounds on multithreaded computations with nonconstant-time reducers, we must first understand how the runtime system implements reducers. We also provide an alternative locking protocol to that proposed by Frigo *et al.* [15] for the runtime system to manage reducers, which guarantees that locks are held for constant time regardless of the runtime of any REDUCE operation.

First, we examine how the runtime system handles spawns and steals, as described in [15]. Every time a Cilk function is stolen, the runtime system creates a new *frame*.¹ Although frames are created and destroyed dynamically during a program execution, the

¹When we refer to frames in this paper, we specifically mean the “full” frames described in [15].

ones that exist always form a rooted 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 this 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 obtain 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.* [15] uses an intricate protocol to avoid long waits on locks, but the analysis of its performance assumes that each REDUCE 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 use these abstract locks to eliminate a frame F according to the locking protocol

```

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  $L == \emptyset$  and  $R == \emptyset$ 
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      $R = \emptyset; L = \emptyset$ 
15     Release the abstract locks
16     for each reducer  $h \in R'$ 
17       if  $h \in F.hu$ 
18          $F.hu(h) \otimes = R'(h)$ 
19       else  $F.hu(h) = R'(h)$ 
20     for each reducer  $h \in L'$ 
21       if  $h \in F.hu$ 
22          $F.hu(h) = L'(h) \otimes F.hu(h)$ 
23       else  $F.hu(h) = L'(h)$ 
24

```

Figure 7-1: A modified locking protocol for eliminating a frame F containing reducers with nonconstant-time REDUCE operations. This locking protocol guarantees that locks are held for $O(1)$ time, regardless of the running time of any REDUCE operation.

shown in Figure 7-1.

Modeling reducers

To specify the nondeterministic behavior encapsulated by reducers precisely, consider a computation A of a multithreaded program, where $V(A)$ 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 can partition $V(A)$ into three classes of strands:

- V_v : User strands arising from the execution of code explicitly invoked by the programmer, including calls to UPDATE.
- V_i : *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_p : *Reduce strands* arising from the execution of REDUCE, which occur implicitly when the runtime system combines views.

Notice that $V_i \cup V_p$ is the set of runtime strands in $V(A)$, as Chapter 6 mentions.

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 therefore maintain the definition of the user dag exactly as it was in Chapter 6. For a computation A , we set $V(\text{User}(A)) = V_v$, and we set $E(\text{User}(A)) = E_\chi$. If A involves reducers, then the user dag omits all runtime strands in A . For example, the user dag associated with the computation in Figure 6-1 looks exactly like the dag shown in Figure 6-2. The work of $\text{User}(A)$ is the sum of the lengths of the strands in $V(\text{User}(A))$, and the span of $\text{User}(A)$ is the length of the longest path through $\text{User}(A)$.

To track the views of a reducer h in the user dag, let $h(v)$ denote the view of h seen by a strand $v \in V(\text{User}(A))$. The runtime system maintains the following invariants:

Invariant 2. If $u \in V(\text{User}(A))$ has out-degree 1 and $(u, v) \in E(\text{User}(A))$, then $h(v) = h(u)$.

Invariant 3. Suppose that $u \in V(\text{User}(A))$ is a spawn strand with outgoing edges $(u, v), (u, w) \in E(\text{User}(A))$, where $v \in V(\text{User}(A))$ is the first strand of the spawned subroutine and $w \in V(\text{User}(A))$ is the continuation in the parent. Then, we have $h(v) = h(u)$ and

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

Invariant 4. If $v \in V(\text{User}(A))$ 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 3, we say that the old view $h(u)$ *dominates* $h(w)$, which we denote by $h(u) > h(w)$. For a set H of views, we say that two views $h_1, h_2 \in H$, where $h_1 > h_2$, are *adjacent* if there does not exist $h_3 \in H$ such that $h_1 > h_3 > h_2$. We can define the *dominates function* $\text{dom}\{h_1, h_2, \dots, h_k\}$ to be the view $h_i \in \{h_1, h_2, \dots, h_k\}$ such that for all $j \in \{1, 2, \dots, k\}$, if $j \neq i$ then $h_i > h_j$.

A useful property of sync strands is that the views of strands entering a sync strand $v \in V(\text{User}(A))$ are totally ordered by the dominates relation. That is, if k strands each have an edge in $E(\text{User}(A))$ to the same sync strand $v \in V(\text{User}(A))$, then the strands can be numbered $u_1, u_2, \dots, u_k \in V(\text{User}(A))$ such that $h(u_1) \geq h(u_2) \geq \dots \geq h(u_k)$. Moreover, we have $h(u_1) = h(v) = h(u)$, 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 $\text{User}(A)$ to bound the P -processor execution time: $T_P(A) \leq \text{Work}(\text{User}(A))/P + O(\text{Span}(\text{User}(A)))$. Chapter 8 applies a similar delay-sequence argument to computations containing runtime strands. To do so, we augment $\text{User}(A)$ with the runtime strands to produce a *performance dag* $\text{Perf}(A)$ for the computation A , where

- $V(\text{Perf}(A)) = V(A) = V_v \cup V_l \cup V_p$,
- $E(\text{Perf}(A)) = E_v \cup E_l \cup E_p$,

where the edge sets E_l and E_p are constructed as follows.

The edges in E_l are created in pairs. For each init strand $v \in V_l$, we include (u, v) and (v, w) in E_l , where $u, w \in V_v$ 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_p are created in groups corresponding to the set of REDUCE functions that must execute before a given sync. Suppose that $v \in V_v$ is a sync strand, that k strands

$u_1, u_2, \dots, u_k \in V_0$ join at v , and that $k' < k$ reduce strands $r_1, r_2, \dots, r_{k'} \in V_\rho$ execute before the sync. Consider the set $U = \{u_1, u_2, \dots, u_k\}$, and let $h(U) = \{h(u_1), h(u_2), \dots, h(u_k)\}$ be the set of $k' + 1$ views that must be reduced. Construct a **reduce tree** as follows:

- 1 **while** $|h(U)| \geq 2$
- 2 Let $r \in \{r_1, r_2, \dots, r_{k'}\}$ 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 $r' \in \{r_1, r_2, \dots, r_{k'}\}$ reduces adjacent strands $h_i, h_{i+1} \in h(U)$, we have $h_i > h_j$
- 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 dag $\text{User}(A)$ that are already in series, the performance dag $\text{Perf}(A)$ is indeed a dag. The work of $\text{Perf}(A)$ is the sum of the lengths of all the strands in $\text{Perf}(A)$. Since we shall consider all the edges in $E(\text{Perf}(A))$ to be control dependencies, the span of $\text{Perf}(A)$ is the length of the longest path in $\text{Perf}(A)$. Chapter 8 proves that we can use the longest path in $\text{Perf}(A)$ to bound the running time of a computation A that contains reducers.

The edges in a performance dag reflect the guarantees that the runtime system makes on when runtime strands occur relative to user strands in a schedule of the computation. Consequently, for a computation A involving reducers, we have two invariants that relate $\text{User}(A)$ and $\text{Perf}(A)$ to $\text{Sched}(A)$, which are similar to Invariant 1.

Invariant 5. If a user strand $u \in \text{User}(V)$ precedes a strand $v \in V(A)$ in $\text{User}(A)$, then u precedes v in $\text{Sched}(A)$.

Invariant 6. If a strand $u \in V(A)$ precedes a sync strand $v \in V(A)$ in $\text{Perf}(A)$, then u precedes v in $\text{Sched}(A)$.

Faithfully representing REDUCE operations

Although the performance dag is useful for bounding the work and span of computations with reducers, it does not faithfully model when reduce strands occur within the computa-

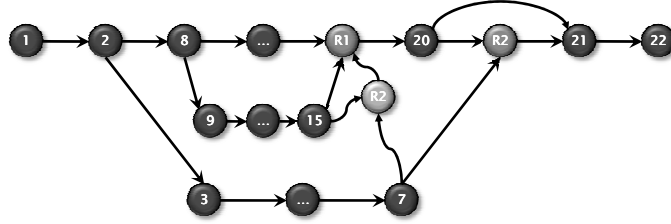


Figure 7-2: A dag representation of a computation with a REDUCE operation executed opportunistically. The set of strands and the set of straight edges represent the performance dag of the computation. In particular, the numbered, dark-colored vertices represent user and init strands within the computation, and each dark-colored vertex labeled with “...” represents a sequence of user and init strands. The gray vertices labeled R1 and R2 represent reduce operations where they are placed in the performance dag. The lighter-gray vertex labeled R2 and connected via curved arrows represents the position of R2 as dictated by scheduling dependencies only.

tion, as illustrated in Figure 7-2. Suppose that we are executing the illustrated computation, which involves a reducer h , and suppose that we have some time step when worker w_1 is executing strand 7, worker w_2 is executing strand 13, and worker w_3 is executing the continuation after strand 8. Furthermore, suppose that the view of h at strand 7 is nonempty, and the view at strand 13 is nonempty. If w_1 completes execution on strand 7 before w_2 completes strand 15, then once w_2 completes strand 15 the runtime system may opportunistically execute the reduce strand R2, even though execution has not yet reached the sync strand 21. The performance dag represents the reduce operation R2 as shown in Figure 7-2 when only parallel-control dependencies are considered, even though R2 was scheduled to execute before then.

We can formalize this phenomenon in terms of the operations of the runtime system. Suppose that during the execution of a parallel computation with a reducer h , there is some point when three workers w_1 , w_2 , and w_3 are operating on frames F_1 , F_2 , and F_3 , respectively, such that F_1 is F_2 's child, which syncs with F_2 at strand u , and F_2 is F_3 's child, which syncs with F_3 at strand v preceding u . In Figure 7-2, w_1 executing strand 7 syncs at sync strand $u = 21$, which follows the sync strand $v = 20$ where w_2 executing strand 13 syncs. We label the view of h in each hypermap such that $h_1 = F_1.hu(h)$, $h_2 = F_2.hu(h)$, and $h_3 = F_3.hu(h)$, and we assert that h_1 and h_2 are nonempty. Suppose that w_1 completes its computation quickly relative to w_2 as when, in Figure 7-2, w_1 completes strand 7 before w_2 completes strand 15. When w_1 returns from its spawn, it eliminates F_1 ,

setting $F_2.hc = h_1$. When w_2 subsequently returns from a spawn to the sync strand v , it must attempt to eliminate F_2 and executes a $\text{REDUCE}(h_1, h_2)$, though the execution has not yet reached u . The performance dag, however, represents the $\text{REDUCE}(h_1, h_2)$ strand in the reduce tree preceding u , although this reduce strand actually occurred before v . It is worth noting that a REDUCE operation may happen early only if the runtime system performs some sequence of operations like these, where a frame with a view h_1 of h is eliminated before its neighboring frame with view h_2 and $h_1 > h_2$.

Chapter 8 proves that the performance of a computation A can nevertheless be bounded in terms of $\text{Perf}(A)$. We shall prove this claim using a delay-sequence argument [28], but in order to construct a delay sequence for this argument, we must first consider a faithful representation of the runtime strands that occur. To prove this claim we first examine how to track the views of a reducer through the scheduling dependencies $\text{Sched}(E)$ in $E(A)$.

To track the views of a reducer h through $\text{Sched}(E) \in E(A)$ for computation A with schedule C , let $\eta(v)$ denote the view of h seen by a strand $v \in V(A)$ as dictated by $\text{Sched}(E)$. For simplicity we assume that every reduce strand $u \in V_p$ performing an operation $\text{REDUCE}(\eta(v_1), \eta(v_2))$ only has scheduling dependencies on strands v_1 and v_2 . We shall assign views of h to the strands of A by replaying A 's execution according to S and using the following assignment rules.

Rule 1. If $u \in V(A)$ enables $v \in V(A)$, where $v \notin R(V)$, v is not a sync strand, and u is not a spawn strand, then $\eta(v) = \eta(u)$.

Rule 2. Suppose that $u \in V(A)$ is a spawn strand, $v \in V(A)$ is the first strand of subroutine spawned by u , and $w \in V(A)$ is the continuation after u . Then set $\eta(v) = \eta(u)$ and

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

Rule 3. Each reduce strand $u \in R(V)$ depends on two strands $v_1, v_2 \in V(A)$ in order to execute $\text{REDUCE}(\eta(v_1), \eta(v_2))$. When a reduce strand $u \in R(V)$ is enabled, set $\eta(u) = \text{dom}\{\eta(v_1), \eta(v_2)\}$, and destroy the dominated view.

Rule 4. If $u \in V(A)$ is a sync strand that depends on strands $v_1, v_2, \dots, v_k \in V(A)$, then set $\eta(u) = \text{dom} \{\eta(v_1), \eta(v_2), \dots, \eta(v_k)\}$, and destroy all other views entering u .

From Rule 2 we infer a *dominates* relation between η views, which parallels the dominates relations between h views in the user dag due to Invariant 3. When a new view $\eta(w)$ is created, as is inferred by Rule 2, we say that the old view $\eta(u)$ *dominates* $\eta(w)$, which we denote by $\eta(u) > \eta(w)$. For a set H of views, we say that two views $\eta_1, \eta_2 \in H$, where $\eta_1 > \eta_2$, are *adjacent* if there does not exist $\eta_3 \in H$ such that $\eta_1 > \eta_3 > \eta_2$. We define the *dominates function* $\text{dom} \{\eta_1, \eta_2, \dots, \eta_k\}$ to be the view $\eta_i \in \{\eta_1, \eta_2, \dots, \eta_k\}$ such that for all $j \in \{1, 2, \dots, k\}$, if $j \neq i$, then $\eta_i > \eta_j$.

We must show that the η views entering a sync strand $v \in V(A)$ are totally ordered by the dominates relation, and thus Rule 4 assigns a well-defined label to each sync strand. Note that η views propagate through $\text{Sched}(A)$ by traversing edges in $\text{Sched}(E)$. Consequently, for a sync strand u to see a view η created by a strand v , there must exist a path in $\text{Sched}(A)$ from u to v along which η is never destroyed. The runtime system maintains the invariant that all threads that sync at a sync strand u in the scheduling dag were spawned along a single subpath terminating at u . Consequently, all of the views entering a given sync strand u must have been created and not destroyed along a single path ending at u . Since the dominance relation on η views implies that any views created and not destroyed along any path are totally ordered, the result of $\text{dom} \{\eta(v_1), \eta(v_2), \dots, \eta(v_k)\}$ in Rule 4 is well-defined.

Although the scheduling dag represents REDUCE operations faithfully, the nondeterminism of REDUCE operations makes analyzing the scheduling dag directly impractical. Our goal is to use the performance dag to account for the cost of operations on reducers. In Chapter 8 we shall use the accounting of reducer operations in the scheduling dag to prove that the performance dag is sufficient for analyzing programs with reducers.

Chapter 8

Analysis of programs with nonconstant-time reducers

This chapter provides a framework for analyzing programs that contain reducers whose REDUCE functions execute in more than constant time. We use an accounting argument to bound the expected running time of a computation A with performance dag $\text{Perf}(A)$. We then bound the work and span of $\text{Perf}(A)$ in terms of the work and span of the user dag $\text{User}(A)$ and the worst-case running time τ of any REDUCE or CREATE-IDENTITY operation. We combine these bounds to prove that the expected running time of A is $T_P(A) \leq \text{Work}(\text{User}(A))/P + O(\tau^2 \cdot \text{Span}(\text{User}(A)))$.

For simplicity, we assume that computation A makes use of at most a single reducer. The following proofs can be extended to handle many reducers within a computation.

Bounding the running time of a computation using its performance dag

We begin by bounding the time to execute a multithreaded computation A containing nonconstant-time reducers with a work-stealing scheduler in terms of its performance dag $\text{Perf}(A)$. The proof follows those of Blumofe and Leiserson [7] and Frigo *et al.* [15], with some salient differences. As in [7], we use a delay-sequence argument, as presented by Ranade [28], but we base it on the performance dag.

To analyze the running time of a work-stealing scheduler executing a multithreaded

computation, we use an accounting argument. Represent a unit of computation performed by a processor during a time step with a dollar. At each step, each processor spends its dollar by placing it into some bucket, depending on the type of task that processor performs at the step. If the execution takes time T_P , then at the end the total number of dollars in all the buckets is PT_P . Consequently, if we sum up all of the dollars in all of the buckets and divide by P , we obtain the running time.

We consider four different types of tasks a processor may perform at a step. If a processor executes an instruction at the step, then it places its dollar into the WORK bucket. If a processor initiates a steal attempt at the step, then it places its dollar into the STEAL bucket. If a processor waits for its steal request to be satisfied at the step, then it places its dollar into the WAIT-STEAL bucket. Finally, if a processor waits to acquire a lock on a data structure in the runtime system at the step, then it places its dollar into the WAIT-LOCK bucket.

First, we bound the total number of dollars in the WORK bucket.

Lemma 1 *The execution of a multithreaded computation A with work $\text{Work}(\text{Perf}(A))$ by a random work-stealing scheduler on a computer with P processors terminates with exactly $\text{Work}(\text{Perf}(A))$ dollars in the WORK bucket.*

PROOF. A processor places a dollar in the WORK bucket only when it executes an instruction. Since there are exactly $\text{Work}(\text{Perf}(A))$ instructions in the computation, the execution ends with exactly $\text{Work}(\text{Perf}(A))$ dollars in the WORK bucket. \square

Next, we bound the total dollars in the STEAL bucket using a delay-sequence argument, as presented by Ranade [28]. The idea of the argument is to construct a “delay sequence” with the scheduling dag $\text{Sched}(A)$, and then to use the performance dag $\text{Perf}(A)$ to bound the length of that delay sequence and, consequently, the expected number of dollars in the STEAL bucket. In other words, we first construct a delay sequence using $\text{Sched}(A)$, then we show how to translate that delay sequence into a delay sequence in $\text{Perf}(A)$, and finally we analyze the delay sequence in $\text{Perf}(A)$.

We begin this argument by identifying a “delaying path” — a path of “critical strands” through an augmented scheduling dag $\text{Sched}'(A)$. This augmented scheduling dag repre-

sents both the scheduling dependencies and the “deque dependencies” on $V(A)$. To construct a delay sequence, we divide the set of steal attempts that occur during A ’s execution into rounds, and associate each round with some strand on the delaying path. Next, we bound the probability that a strand remains critical across many steal attempts. We then bound the total length of the delay sequence by mapping this delay sequence to a path of strands in an augmented version of the performance dag that accounts for deque dependencies. We conclude that any delay sequence is unlikely to occur, and the execution of A is unlikely to suffer a large number of steal attempts.

We assume without loss of generality that every strand represents a single instruction. When we refer to an init or reduce strand, we mean the sequence of single-instruction strands that perform a particular CREATE-IDENTITY or REDUCE operation. We also assume for simplicity that every instruction takes a single time step to execute.

A *round* of steal attempts is a set of at least $3P$ but fewer than $4P$ consecutive steal attempts such that if a steal attempt that is initiated at time step t occurs in a particular round, then all other steal attempts initiated at time step t occur in the same round. We can partition all of the steal attempts that occur during an execution as follows. The first round contains all steal attempts initiated at time steps $1, 2, \dots, t_1$, where t_1 is the first time step such that at least $3P$ steal attempts were initiated at or before t_1 . For $i > 1$ we say that the i th round of steals begins at time step $t_{i-1} + 1$ and ends at time step t_i , where at least $3P$ consecutive steal attempts were initiated between time step $t_{i-1} + 1$ and time step t_i . Each round necessarily contains at least $3P$ consecutive steal attempts, and because fewer than P steal attempts can be initiated at a single time step, each round contains fewer than $4P$ steal attempts, and each round takes at least 4 steps.

The criticality of a strand $v \in V(A)$ is defined in terms of scheduling dependencies and deque dependencies. A *deque dependency* exists from a strand u to a strand v if v is the first child strand after a spawn strand w and u is the continuation strand after w . We augment $\text{Sched}(A)$ to form a new dag $\text{Sched}'(A)$ that reflects both the scheduling and deque dependencies between strands. Formally, the augmented scheduling dag $\text{Sched}'(A)$ has the same vertex set as $V(\text{Sched}(A))$ and all of the edges in $E(\text{Sched}(A))$ plus an edge (u, v) for every pair of strands $u, v \in V(A)$ such that a deque dependency exists from u to v — u is a

continuation after a spawn strand w and v is the first child strand after w . Since we assumed in Chapter 6 that no continuation strand is also a sync strand, the augmented scheduling dag is indeed a dag. We assume for simplicity that for any spawn strand w , neither the first spawned child of w nor the continuation after w is a reduce strand.

A strand $u \in V(A)$ is **critical** at time step t if it is unexecuted at time step t and all immediate predecessor strands $v \in \{v_1, v_2, \dots, v_k\}$ of u in $\text{Sched}'(A)$ have been executed. If a strand u is critical, then u must be enabled since all of u 's scheduling dependencies have been executed.

We use the augmented scheduling dag $\text{Sched}'(A)$ to create a delay sequence. A **delay sequence** of a computation A with schedule C is a triple (p, R, Π) consisting of the following.

- A **delaying path** $p = \langle u_1, u_2, \dots, u_L \rangle$ — a maximal path through $\text{Sched}'(A)$, meaning a path through the strands of A such that u_1 is the first strand in $\text{Sched}'(A)$, u_L is the last strand in $\text{Sched}'(A)$, and for each $i = 1, 2, \dots, L-1$, strand u_{i+1} follows u_i if $(u_i, u_{i+1}) \in E(\text{Sched}'(A))$;
- A positive number R of steal-attempt rounds; and
- A partition $\Pi = (\pi_1, \pi'_1, \pi_2, \pi'_2, \dots, \pi_L, \pi'_L)$ of the R steal-attempt rounds such that $|\pi'_i| \in \{0, 1\}$ for $i = 1, 2, \dots, L$.

For each $i = 1, 2, \dots, L$, we define π_i , the i th **group** of steal-attempt rounds, to be the $|\pi_i|$ consecutive rounds that begin after the r_i th round, where $r_i = \sum_{j=1}^{i-1} |\pi_j \cup \pi'_j|$. Because Π is a partition of the R steal-attempt rounds and $|\pi_i| \in \{0, 1\}$ for $i = 1, 2, \dots, L$, we have

$$\sum_{i=1}^L |\pi_i| \geq R - L, \quad (8.1)$$

We say that a given round of steal attempts **occurs** while strand v is critical if all of the steal attempts that comprise the round are initiated at time steps when v is critical. In other words, v must be critical throughout the entire round. A delay sequence **occurs** during the execution of A if, for each $i = 1, 2, \dots, L$, strand u_i is critical throughout all rounds in π_i .

The following lemma shows that if at least R rounds take place during an execution,

then some delay sequence (p, R, Π) must occur. In particular, if we look at any execution in which at least R rounds occur, then we can identify a path $p = \langle u_1, u_2, \dots, u_L \rangle$ in $\text{Sched}'(A)$ and a partition $\Pi = (\pi_1, \pi'_1, \pi_2, \pi'_2, \dots, \pi_L, \pi'_L)$ of the first R steal-attempt rounds such that for each $i = 1, 2, \dots, L$, all of the rounds in π_i occur while u_i is critical. Each π'_i contains at most one round, which is the round that began when u_i was critical and ended after u_i began executing. Such a round cannot be part of any group because no instruction is critical throughout.

Lemma 2 *Consider the execution of a multithreaded computation A by a work-stealing scheduler on an ideal parallel computer with P processors. If at least $4PR$ steal attempts occur during A 's execution, then some delay sequence (p, R, Π) must occur.*

PROOF. For a given execution of A in which at least $4PR$ steal attempts occur, we construct a delay sequence (p, R, Π) and show that it occurs. Because at least $4PR$ steal attempts occur, there must be at least R rounds of steal attempts.

We construct a delaying path in the augmented scheduling dag $\text{Sched}'(A)$ as follows. Let C be the schedule generated by the execution of A . Starting with the last strand v_1 in C and the trivial path $p' = \langle v_1 \rangle$, we find the immediate predecessor v_2 of v_1 in $\text{Sched}'(A)$ with the latest completion time in C , and prepend v_2 to p' to obtain $p' = \langle v_2, v_1 \rangle$. On each subsequent iteration $i > 1$, from strand v_i with associated path $p' = \langle v_i, v_{i-1}, \dots, v_2, v_1 \rangle$, we find the immediate predecessor v_{i+1} of v_i in $\text{Sched}'(A)$ with the latest completion time in C and prepend v_{i+1} to p' to obtain $p' = \langle v_{i+1}, v_i, v_{i-1}, \dots, v_2, v_1 \rangle$.

In the event of a tie between the latest completion times of strands preceding a reduce strand or a sync strand, we impose two special-case rules on this process of constructing a delaying path. First, if the predecessors w_1 and w_2 of a reduce strand u — which have views $\eta(w_1) > \eta(w_2)$ of the reducer h — have the same completion time, then we choose the immediate predecessor of u in the delaying path to be w_2 . Similarly, if the k predecessors w_1, w_2, \dots, w_k of a sync strand u with reducer views $\eta(w_1) > \eta(w_2) > \dots > \eta(w_k)$ all have the same completion time, then we choose the immediate predecessor of u in the delaying path to be w_k . These rules for constructing a delaying path preclude a sequence of strands that enables an early reduce operation or return from a spawn from appearing on a delaying

path, which makes intuitive sense since such a sequence executing a little bit later should not delay the computation as a whole.

After L iterations, this path has been extended to v_L , the first strand executed in C . At this point the delaying path is $p = \langle u_1, u_2, \dots, u_L \rangle$, where $u_i = v_{L-i+1}$ for $i = 1, 2, \dots, L$. One can verify that at every time step of the execution, one of the u_i is critical.

Next, we construct the partition $\Pi = (\pi_1, \pi'_1, \pi_2, \pi'_2, \dots, \pi_L, \pi'_L)$ of the R steal-attempt rounds. To construct Π we walk through the partitions and the rounds of steal attempts in parallel in order by increasing index, and we place each round of steal attempts in exactly one partition π_i or π'_i for some i . Formally, for each $i = 1, 2, \dots, R$ in order, first we let π_i equal the set of rounds that occur while u_i is critical. Next, if the $(r_i + |\pi_i|)$ th round begins after u_i has executed, then $\pi'_i = \emptyset$; otherwise this round begins while u_i is critical and ends after u_i has executed, and thus π'_i is the set containing only the $(r_i + |\pi_i|)$ th round. Finally, we repeat on π_{i+1} .

We conclude this proof by verifying that the delay sequence (p, R, Π) as constructed is a delay sequence that occurs during A 's execution. By construction, p is a maximal path through the strands of A that obeys scheduling and deque dependencies. Since each of the R rounds is placed in exactly one of the $\pi_i \in \Pi$ or $\pi'_i \in \Pi$ by construction, Π is a partition of the R steal-attempt rounds. Moreover, for $i = 1, 2, \dots, L$, at most one round can begin while strand u_i is critical and end after u_i is executed, and thus we have $|\pi'_i| \in \{0, 1\}$. Therefore, (p, R, Π) is a delay sequence. Finally, by construction the rounds in π_i all occur while strand u_i is critical for $i = 1, 2, \dots, L$. Consequently, the delay sequence (p, R, Π) occurs. \square

We now show that a critical instruction is unlikely to remain critical across a modest number of rounds.

Lemma 3 *Consider the execution of a multithreaded computation A by a work-stealing scheduler on an ideal parallel computer with $P \geq 2$ processors. For any strand u and any number $r \geq 2$ of steal-attempt rounds, the probability that any particular set of r rounds occur while the strand u is critical is at most the probability that only 0 or 1 of the steal attempts initiated in the first $r - 1$ of these rounds chose u 's worker, which is at most e^{-2r+3} .*

PROOF.

Let C be the schedule generated by the execution of A . If u is a user strand, then Blumofe and Leiserson prove this claim in [7, Lemma 11]. Thus, suppose that u is a runtime strand, and let $C(u) = (w_u, t_u)$. In this case, because runtime strands are executed opportunistically by the scheduler, u was never placed on w_u 's deque. Instead, w_u begins executing u at the time step after u is enabled. Since at most $P - 1$ steal attempts can occur during a single time step, and a round of steal attempts consists of at least $3P$ steal attempts, less than 1 round of steal attempts can occur while u is critical. Therefore, u is not critical throughout any rounds of steal attempts, and thus the probability bound in [7, Lemma 11] holds. □

To complete the delay-sequence argument, we show how to translate a delay sequence in $\text{Sched}'(A)$ into a delay sequence in an augmented version of $\text{Perf}(A)$. Much like the augmented scheduling dag, the augmented performance dag $\text{Perf}'(A)$ is the performance dag $\text{Perf}(A)$ augmented with additional edges to represent deque dependencies. For every set of instructions $u, v, w \in \text{Perf}(V)$ such that u is a spawn strand, v is the first strand in the spawned child of u , and w is the continuation strand after u , the deque edge (w, v) is included in $\text{Perf}'(A)$. Since we assumed in Chapter 6 that no continuation strand is also a sync strand, the augmented dag is indeed a dag. Moreover, we have $\text{Span}(\text{Perf}'(A)) \leq 2 \cdot \text{Span}(\text{Perf}(A))$.

We now prove that for a computation A and any delay sequence (p, R, Π) in $\text{Sched}'(A)$, a corresponding delay sequence in $\text{Perf}'(A)$ exists containing all of the strands in p .

Lemma 4 *Consider the execution of a multithreaded computation A by a work-stealing scheduler on an ideal P -processor parallel computer. For any delay sequence (p, R, Π) in $\text{Sched}'(A)$, a corresponding delay sequence exists in $\text{Perf}'(A)$. Moreover, the length L of the delay sequence satisfies $L \leq \text{Span}(\text{Perf}'(A)) \leq 2 \cdot \text{Span}(\text{Perf}(A))$.*

PROOF. We consider the same set of R steal-attempt rounds in the delay sequence in the augmented performance dag, as well as the same partition of those rounds and association of each round of steal-attempts to some strand in p . Our goal is to find a delaying path q

in $\text{Perf}'(A)$ such that $p \subseteq q$. To do so, we treat the user strands, init strands, and reduce strands in p separately. We assume that no steal-attempt rounds occur during strands $v \in q$ for $v \notin p$.

First, consider all user strands $u \in p$. The insertion of runtime strands to create $\text{Perf}(A)$ from $\text{User}(A)$ does not change the partial order on user strands in $\text{Perf}(A)$, and by Invariant 5 the partial order of strands in $\text{User}(A)$ is maintained in $\text{Sched}(A)$. Moreover, by construction of p , a user strand u is never the immediate predecessor of a sync strand v in p unless u is v 's immediate predecessor in $\text{User}(A)$. Finally, each deque dependency in $\text{Sched}'(A)$ is represented with a deque edge in $\text{Perf}'(A)$. Therefore, for any two user strands $u, v \in p$ such that $u \succ v$ in p , there must exist a path from u to v in $\text{Perf}'(A)$. By structural induction, for all strands $u \in V_v$, we have $u \in p \rightarrow u \in q$.

Second, consider the init strands $v \in p$. Every init strand is inserted into $\text{Perf}(A)$ between the user strands $u, w \in V_v$ that comprise the instructions in UPDATE whose execution invoked the CREATE-IDENTITY operation corresponding to v . These inserted edges therefore enforce the scheduling dependencies on each init strand v . Consequently, for any two nonreduce strands $u, v \in p$ such that $u \succ v$, there must exist a path from u to v in $\text{Perf}'(A)$. By structural induction, for all strands $u \in V_v \cup V_l$ we have $u \in p \rightarrow u \in q$.

Third, we must account for the reduce strands in p . We start by stating some invariants on reduce operations performed on a reducer h .

Invariant 7. For a reduce strand $v \in p$ to execute, all strands $u \in V(\text{Perf}(A))$ with view $h(u) = h(v)$ such that $u \succ v$ must have already completed execution. The edge set $E(\text{Sched}(A))$ maintains this constraint.

Invariant 8. Suppose that a strand u with reducer view $h(u)$ exists on some path p' in $\text{Sched}(A)$ or $\text{Perf}(A)$ that reaches the last strand in A , and suppose that the view $h(u)$ is eventually destroyed at strand v . Then we have $v \in p'$.

Invariant 9. Suppose that a strand $v \in V_p$ has the immediate predecessors u_1 and u_2 in $\text{Sched}(A)$, and suppose that v has the immediate predecessors u'_1 and u'_2 in $\text{Perf}(A)$. By Invariant 7 and because reduces occur eagerly, one of the immediate predecessors of v in $\text{Sched}(A)$ must be v 's immediate predecessor in $\text{Perf}(A)$. Without loss of

generality, assume $u_1 = u'_1$. If and only if $u_2 \neq u'_2$, then u_2 is a reduce strand that executed early with respect to its dependencies in $\text{Perf}(A)$. In particular, u_2 reduces $h(v)$ with a view that dominates $h(v)$ immediately after u'_2 executes and immediately before v reduces $h(v)$ with view $h(u'_2)$ where $h(v) > h(u'_2)$. Consequently, we have $h(u_2) \neq h(v)$ and $h(u_2) \neq h(u'_2)$.

Moreover, recall that we assume that no reduce strand is involved in a deque dependency. Consequently, the predecessor of a reduce strand in p can only be the destination of a deque edge.

Suppose that A involves a reducer h , and suppose that p contains some reduce strand v that reduces two performance-dag views h_i and h_j , where $h_i > h_j$. Consider v 's immediate predecessor u in p , which has the performance-dag view $h(u)$. We have three cases concerning the relation of view $h(u)$ to h_i and h_j .

Case $h(u) = h_i$: By Invariants 7 and 9, the strand u must be v 's immediate predecessor in $\text{Perf}'(A)$. Therefore, v is u 's immediate successor in q , and we have $u \in q \rightarrow v \in q$.

Case $h(u) = h_j$: Since v destroys h_j , it follows that v destroys the view $h(u)$. By Invariant 8, v is on every path in $\text{Perf}(A)$ from u to the last strand in A . Deque dependencies only add continuation strands to a path in $\text{Perf}(A)$ from u to the last strand in A that is otherwise realizable in $\text{Perf}(A)$. Therefore, we have $u \in q \rightarrow v \in q$.

Case $h(u) \neq h_j$ and $h(u) \neq h_i$: By Invariant 9, strand u is a reduce strand that executed early. In particular, u executed immediately before v and immediately after strand u' , where u' is v 's immediate predecessor in $\text{Perf}'(A)$. By construction, u' must be u 's immediate predecessor in p , and we have $u \in q \rightarrow u' \in q \rightarrow v \in q$.

By structural induction, for every strand $v \in V_p$, we have $v \in p \rightarrow v \in q$.

For any delaying path p in $\text{Sched}'(A)$, there exists a maximal path in $\text{Perf}'(A)$ containing all strands in p . Consequently, the length of the delaying path $L \leq \text{Span}(\text{Perf}'(A))$.

□

We now complete the delay-sequence argument and bound the total dollars in the STEAL bucket.

Lemma 5 Consider the execution of a multithreaded computation A by a work-stealing scheduler on an ideal parallel computer with P processors. For any $\varepsilon > 0$, with probability $1 - \varepsilon$, the execution terminates with at most $O(P(\text{Span}(\text{Perf}(A)) + \lg(1/\varepsilon)))$ dollars in the STEAL bucket, and the expected number of dollars in this bucket is $O(P \cdot \text{Span}(\text{Perf}(A)))$.

PROOF. From Lemma 2, we know that if at least $4PR$ steal attempts occur, then some delay sequence (p, R, Π) must occur. Consider a particular delay sequence (p, R, Π) having $p = \langle u_1, u_2, \dots, u_L \rangle$ and $\Pi = (\pi_1, \pi'_1, \pi_2, \pi'_2, \dots, \pi_L, \pi'_L)$. From Lemma 4 we know that $L \leq 2 \cdot \text{Span}(\text{Perf}(A))$. We compute the probability that (p, R, Π) occurs.

Such a sequence occurs if, for $i = 1, 2, \dots, L$, each strand u_i is critical throughout all rounds in π_i . Form Lemma 3, we know that the probability that all $|\pi_i|$ rounds in π_i occur while a given strand u_i is critical is at most the probability that only 0 or 1 of the steal attempts initiated in the first $|\pi_i| - 1$ of these rounds chose u_i 's worker, which is at most $e^{-2|\pi_i|+3}$ for $|\pi_i| \geq 2$. For $|\pi_i| < 2$ we use 1 as an upper bound on this probability. Moreover, since the work stealing scheduler chooses all targets of work-steal attempts independently, we can bound the probability of the delay sequence (p, R, Π) occurring as follows:

$$\begin{aligned}
\Pr \{(p, R, \Pi) \text{ occurs}\} &= \prod_{1 \leq i \leq L} \Pr \{\text{the rounds in } \pi_i \text{ occur while } u_i \text{ is critical}\} \\
&\leq \prod_{1 \leq i \leq L; |\pi_i| \geq 2} e^{-2|\pi_i|+3} \\
&\leq \exp \left[-2 \left(\sum_{1 \leq i \leq L; |\pi_i| \geq 2} |\pi_i| \right) + 3L \right] \\
&= \exp \left[-2 \left(\sum_{1 \leq i \leq L} |\pi_i| - \sum_{1 \leq i \leq L; |\pi_i| < 2} |\pi_i| \right) + 3L \right] \\
&\leq e^{-2((R-L)-L)+3L} \\
&= e^{-2R+7L},
\end{aligned}$$

where the last inequality follows from Inequality (8.1).

To bound the probability of some delay sequence (p, R, Π) occurring, we need to count the number of such delay sequences and multiply by the probability that a particular de-

lay sequence occurs. From Lemma 4, every delay sequence in the execution of A has a corresponding sequence in $\text{Perf}'(A)$ that contains all strands in p . Since the maximum out-degree of any strand in $\text{Perf}'(A)$ is 2, there are at most $2^{2 \cdot \text{Span}(\text{Perf}(A))}$ distinct maximal paths in $\text{Perf}'(A)$. Therefore, there are at most $2^{2 \cdot \text{Span}(\text{Perf}(A))}$ distinct choices for a delaying path p . There are at most $\binom{2L+R}{R} \leq \binom{4 \cdot \text{Span}(\text{Perf}(A)) + R}{R}$ ways to choose Π , since Π partitions R into $2 \cdot \text{Span}(\text{Perf}(A))$ pieces. As we have just shown, a given delay sequence has at most an $e^{-2R+14 \cdot \text{Span}(\text{Perf}(A))}$ chance of occurring. Multiplying these three factors together bounds the probability that any delay sequence (p, R, Π) occurs by

$$2^{2 \cdot \text{Span}(\text{Perf}(A))} \binom{4 \cdot \text{Span}(\text{Perf}(A)) + R}{R} e^{-2R+14 \cdot \text{Span}(\text{Perf}(A))},$$

which is at most ε for $R = c \cdot \text{Span}(\text{Perf}(A)) + \lg(1/\varepsilon)$, where c is a sufficiently large positive constant. Thus the probability that at least $4PR = \Theta(P(\text{Span}(\text{Perf}(A)) + \lg(1/\varepsilon)))$ steal attempts occur is at most ε . The expectation bound follows since the tail of the distribution decreases exponentially. \square

Next, we bound the number of dollars in the WAIT-STEAL bucket.

Lemma 6 *Consider the execution of any multithreaded computation A on an ideal parallel computer with P processors using a work-stealing scheduler. For any $\varepsilon > 0$, with probability at least $1 - \varepsilon$, the number of dollars in the WAIT-STEAL bucket is at most a constant times the number of dollars in the STEAL bucket plus $O(P \lg P + P \lg(1/\varepsilon))$, and the expected number of dollars in the WAIT-STEAL bucket is at most the number in the STEAL bucket.*

PROOF. The proof of this lemma follows from the proof given by Blumofe and Leiserson in [7, Lemma 6]. \square

The next lemma analyzes the work required to perform all eliminations using the locking protocol in Figure 7-1.

Lemma 7 *Consider the execution of a multithreaded computation A on an ideal parallel computer with P processors using a work-stealing scheduler. If M is the number of successful steals during the execution of A , then the expected number of dollars in the WAIT-LOCK*

bucket is $O(M)$, and with probability at least $1 - \epsilon$, at most $O(M + \lg P + \lg(1/\epsilon))$ dollars end up in the WAIT-LOCK bucket.

PROOF.

We apply the analysis of the locking protocol presented in [15] to the locking protocol presented in Figure 7-1. Since lines 3–15 in Figure 7-1 all require $O(1)$ work, each abstract lock is held for a constant amount of time. Since only two workers can compete for any given abstract lock simultaneously, and assuming linear waiting on locks [9], the total amount of time workers spend waiting for workers holding two abstract locks is at most proportional the number M of successful steals. Consequently, we only need to analyze the time workers that are holding only one abstract lock spend waiting for their second abstract lock.

Consider the eliminations attempts performed by a given worker w , and assume that w performed m elimination attempts and hence $2m$ abstract lock acquisitions. Consider the steal tree at the time of the i th abstract lock acquisition by w , when w is trying to eliminate frame F . Each worker x that is trying to eliminate some other frame $F_x \neq F$ in the steal tree at the same time creates an arrow across F_x oriented in the direction from the first edge x abstractly locks to the second. These arrows create directed paths in the tree that represent chains of dependencies on a given abstract lock, and the delay for w 's i th lock acquisition can be at most the length of such a directed path starting at the edge that w is abstractly locking. Since the orientation of lock acquisition along this path is fixed, and each pair of acquisitions is correctly oriented with probability $1/2$, the waiting time for F acquiring one of its locks can be bounded by a geometric distribution:

$$\Pr\{w \text{ waits for } \geq k \text{ elimination attempts}\} \leq 2^{-k-1} .$$

We compute a bound on the total time Δ for all $2m$ abstract lock acquisitions by worker w . First, we must prove that the i th abstract lock acquisition by some worker w is independent of the time for the j th abstract lock acquisition for $i < j$. To prove this independence result, we argue that for two workers w and v , we have $\Pr\{v \text{ delays } w_j \mid v \text{ delays } w_i\} = \Pr\{v \text{ delays } w_j \mid v \text{ does not delay } w_i\} = \Pr\{v \text{ delays } w_j\}$,

where w_i and w_j are w 's i th and j th lock acquisitions, respectively.

We consider each of these cases separately. First, suppose that worker v delays acquisition w_i . After w_i , worker 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 acquisition 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 $\Pr\{v \text{ delays } w_j \mid v \text{ delays } w_i\} = \Pr\{v \text{ delays } w_j \mid v \text{ does not delay } w_i\} = \Pr\{v \text{ delays } w_j\}$.

For all workers $v \neq w$, the probability that v delays acquisition w_j is independent of whether v delays acquisition w_i . Consequently, every lock acquisition by some worker is independent of all previous acquisitions. Thus, the probability that the $2m$ acquisitions take time longer than Δ elimination attempts is at most

$$\begin{aligned} \binom{\Delta}{2m} 2^{-\Delta} &\leq \left(\frac{e\Delta}{2m}\right)^{2m} 2^{-\Delta} \\ &\leq \epsilon'/P \end{aligned}$$

by choosing $\Delta = c(m + \lg(1/\epsilon'))$, where c is a sufficiently large positive constant.

Next, we bound the number of elimination attempts that occur. 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 $3M$.

Since there are at most P workers, the total number of dollars in the WAIT-LOCK bucket is $O(M + \lg P + \lg(1/\epsilon))$ with probability at least $1 - \epsilon$ (letting $\epsilon = \epsilon'/P$). The expectation

bound follows directly. □

We combine these bounds on the dollars within each bucket to find the expected running time of a Cilk-like computation A .

Lemma 8 *Consider the execution of a multithreaded computation A on an ideal parallel computer with P processors using a work-stealing scheduler. The expected running time of A is $T_P(A) \leq \text{Work}(\text{Perf}(A))/P + O(\text{Span}(\text{Perf}(A)))$, and for any $\epsilon > 0$, with probability at least $1 - \epsilon$, the execution time on P processors is $\text{Work}(\text{Perf}(A))/P + O(\text{Span}(\text{Perf}(A))) + \lg P + \lg(1/\epsilon)$.*

PROOF. From Lemma 5 the number successful steal attempts is $O(P \cdot \text{Span}(\text{Perf}(A)))$ in expectation, and $O(P(\text{Span}(\text{Perf}(A)) + \lg(1/\epsilon)))$ with probability at least $1 - \epsilon$. From Lemma 7, the expected number of dollars in the WAIT-LOCK bucket is $O(P \cdot \text{Span}(\text{Perf}(A)))$. Summing over the expected number of dollars in the remaining buckets from Lemmas 1 and 6 and dividing by P , the expected running time of A is $T_P(A) \leq \text{Work}(\text{Perf}(A))/P + O(\text{Span}(\text{Perf}(A)))$, and with probability at least $1 - \epsilon$ the execution time on P processors is $\text{Work}(\text{Perf}(A))/P + O(\text{Span}(\text{Perf}(A))) + \lg P + \lg(1/\epsilon)$. □

Relating the performance dag to the user dag

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

Lemma 9 *Consider a multithreaded computation A , and let τ be the worst-case cost of any CREATE-IDENTITY or REDUCE operation for the given input. Then, we have $\text{Span}(\text{Perf}(A)) = O(\tau \cdot \text{Span}(\text{User}(A)))$.*

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 $\text{User}(A)$ 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 $\text{Perf}(A)$, and let p_U be the corresponding path in $\text{User}(A)$. Suppose that k steals occur along p_U . The length of that corresponding path in $\text{Perf}(A)$ is at most $2k\tau + |p_U| \leq 2\tau \cdot \text{Span}(\text{User}(A)) + |p_U| \leq 3\tau \cdot \text{Span}(\text{User}(A))$. Therefore, we have $\text{Span}(\text{Perf}(A)) = O(\tau \cdot \text{Span}(\text{User}(A)))$. \square

Lemma 10 *Consider a multithreaded computation A , and let τ be the worst-case cost of any CREATE-IDENTITY or REDUCE operation for the given input. Then, we have $\text{Work}(\text{Perf}(A)) = \text{Work}(\text{User}(A)) + O(\tau^2 P \cdot \text{Span}(\text{User}(A)))$ in expectation, and with probability at least $1 - \varepsilon$ for any $\varepsilon > 0$ we have $\text{Work}(\text{Perf}(A)) = \text{Work}(\text{User}(A)) + O(\tau P(\tau \cdot \text{Span}(\text{User}(A)) + \lg P + \lg(1/\varepsilon)))$.*

PROOF. The work in $\text{Perf}(A)$ is the work in $\text{User}(A)$ plus the work represented in the runtime strands. The total work in reduce strands equals the total work to join stolen strands, which is at most the work of each REDUCE operation times twice the number of elimination attempts. By Lemma 7, this value is $O(\tau P \cdot \text{Span}(A))$ in expectation, and $O(\tau P(\text{Span}(\text{Perf}(A)) + \lg P + \lg(1/\varepsilon)))$ with probability $1 - \varepsilon$. Similarly, each steal may create one init strand, and by Lemma 5 the total work in init strands is $O(\tau P \cdot \text{Span}(A))$ in expectation, and $O(\tau P(\text{Span}(\text{Perf}(A)) + \lg(1/\varepsilon)))$ with probability at least $1 - \varepsilon$. Thus, we have $\text{Work}(\text{Perf}(A)) = \text{Work}(\text{User}(A)) + O(\tau P \cdot \text{Span}(\text{Perf}(A)))$ in expectation, and $\text{Work}(\text{Perf}(A)) = \text{Work}(\text{User}(A)) + O(\tau P(\text{Span}(\text{Perf}(A)) + \lg P + \lg(1/\varepsilon)))$ with probability at least $1 - \varepsilon$. Applying Lemma 9 yields the lemma. \square

We now prove Inequality (6.4), which bounds the running time of a computation whose nondeterminism arises from reducers.

Theorem 11 *Consider the execution of a computation A on an ideal parallel computer with P processors using a work-stealing scheduler. Let $\text{User}(A)$ be the user dag of A . The total expected running time of A is $T_P(A) \leq \text{Work}(\text{User}(A))/P + O(\tau^2 \cdot \text{Span}(\text{User}(A)))$, and for any $\varepsilon > 0$, with probability at least $1 - \varepsilon$, the execution time on P processors is $\text{Work}(\text{User}(A))/P + O(\tau^2 \cdot \text{Span}(\text{User}(A)) + \tau \lg P + \tau \lg(1/\varepsilon))$.*

PROOF. The proof of this theorem follows from Lemmas 8, 9, and 10. \square

Chapter 9

Analysis of PBFS

This chapter applies the results of Chapter 8 to bound the expected running time of the locking version of PBFS. For an input graph $G = (V, E)$ with diameter D and bounded out-degree, we shall bound the work of PBFS's user dag with $O(V + E)$, and we shall bound the span of PBFS's user dag with $O(D \lg(V/D))$. Using $O(\lg(V/D))$ as the bound on the worst-case running time of any REDUCE or CREATE-IDENTITY in PBFS, we obtain $T_P(\text{PBFS}) \leq O(V + E)/P + O(D \lg^3(V/D))$ to be the expected running time of PBFS.

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

Lemma 12 *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 [11, 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 d th 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 d th 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 13 *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 $O(\lg(V/D))$. Apply-

ing Theorem 11 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

Chapter 10

Conclusion

We conclude with a discussion of thread-local storage. *Thread-local storage* [29], 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. I 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.

Bibliography

- [1] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA*, pages 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] David A. Bader and Kamesh Madduri. Designing multithreaded algorithms for breadth-first search and *st*-connectivity on the Cray MTA-2. In *ICPP*, pages 523–530, 2006.
- [4] Guy E. Blelloch. Programming parallel algorithms. *CACM*, 39(3), 1996.
- [5] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *JPDC*, 37(1):55–69, 1996.
- [6] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. on Comput.*, 27(1):202–229, 1998.
- [7] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.
- [8] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *JACM*, 21(2):201–206, 1974.
- [9] James E. Burns. Mutual exclusion with linear waiting using binary shared variables. *SIGACT News*, 10(2):42–47, 1978.
- [10] Guojing Cong, Sreedhar Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay Saraswat, and Tong Wen. Solving large, irregular graph problems using adaptive work-stealing. In *ICPP*, pages 536–545, 2008.
- [11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.
- [12] Timothy A. Davis. University of Florida sparse matrix collection, 2010. Available at <http://www.cise.ufl.edu/research/sparse/matrices/>.

- [13] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *ICCD*, pages 522–525, 1992.
- [14] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, 1989.
- [15] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *SPAA*, pages 79–90, 2009.
- [16] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.
- [17] John R. Gilbert, Gary L. Miller, and Shang-Hua Teng. Geometric mesh partitioning: Implementation and experiments. *SIAM J. on Sci. Comput.*, 19(6):2091–2110, 1998.
- [18] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell Sys. Tech. J.*, 45:1563–1581, 1966.
- [19] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview scalability analyzer. In *SPAA*, 2010.
- [20] Yuxiong He. Multicore-enabling the Murphi verification tool. Available from <http://software.intel.com/en-us/articles/multicore-enabling-the-murphi-verification-tool/>, 2009.
- [21] Intel Corporation. *Intel Cilk++ SDK Programmer’s Guide*, 2009. Document Number: 322581-001US.
- [22] Richard E. Korf and Peter Schultze. Large-scale parallel breadth-first search. In *AAAI*, pages 1380–1385, 2005.
- [23] C. Y. Lee. An algorithm for path connection and its applications. *IRE Trans. on Elec. Comput.*, EC-10(3):346–365, 1961.
- [24] Charles E. Leiserson. The Cilk++ concurrency platform. *J. Supercomput.*, 51(3):244–257, 2010.
- [25] Jure Leskovec, Deepayan Chakrabarti, Jon M. Kleinberg, and Christos Faloutsos. Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication. In *PKDD*, pages 133–145, 2005.
- [26] Jir Barnat Lubos, Lubos Brim, and Jakub Chaloupka. Parallel breadth-first search LTL model-checking. In *ASE*, pages 106–115, 2003.
- [27] Edward F. Moore. The shortest path through a maze. In *Int. Symp. on Th. of Switching*, pages 285–292, 1959.
- [28] Abhiram G. Ranade. The delay sequence argument. In *Handbook of Randomized Algorithms*, chapter 1. Kluwer Academic Publishers, 2001.

- [29] D. Stein and D. Shah. Implementing lightweight threads. In *USENIX*, pages 1–9, 1992.
- [30] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 2000.
- [31] Supertech Research Group, MIT/LCS. *Cilk 5.4.6 Reference Manual*, 1998. Available from <http://supertech.csail.mit.edu/cilk/>.
- [32] 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.
- [33] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *SC '05*, page 25, 2005.
- [34] Yang Zhang and Eric 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.