An Analysis of Dag-Consistent Distributed Shared-Memory Algorithms

Robert D. Blumofe* Matteo Frigo† Christopher F. Joerg†
Charles E. Leiserson† Keith H. Randall†

*Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
rdb@cs.utexas.edu

†MIT Laboratory for Computer Science
545 Technology Square
Cambridge, Massachusetts 02139
{athena,cfj,cel,randall}@lcs.mit.edu

Abstract

In this paper, we analyze the performance of parallel multithreaded algorithms that use dag-consistent distributed shared memory. Specifically, we analyze execution time, page faults, and space requirements for multithreaded algorithms executed by a work-stealing thread scheduler and the BACKER algorithm for maintaining dag consistency. We prove that if the accesses to the backing store are random and independent (the BACKER algorithm actually uses hashing), the expected execution time $T_P(C)$ of a "fully strict" multithreaded computation on P processors, each with a LRU cache of C pages, is $O(T_1(C)/P + mT_\infty)$, where $T_1(C)$ is the total work of the computation including page faults, T_∞ is its critical-path length excluding page faults, and m is the minimum page transfer time. As a corollary to this theorem, we show that the expected number $F_P(C)$ of page faults incurred by a computation executed on P processors can be related to the number $F_1(C)$ of serial page faults by the formula $F_P(C) \leq F_1(C) + O(CPT_\infty)$. Finally, we give simple bounds on the number of page faults and the space requirements for "regular" divide-and-conquer algorithms. We use these bounds to analyze parallel multithreaded algorithms for matrix multiplication and LU-decomposition.

1 Introduction

In recent work [8, 17], we have proposed dag-consistent distributed shared memory as a virtual-memory model for multithreaded parallel-programming systems such as *Cilk*, a C-based multithreaded language and runtime system [7, 9, 17]. A multithreaded program defines a partial execution order on its instructions, and we view this partial order as a directed acyclic graph or *dag*. Informally, in the dag-consistency model, a read instruction can "see" a write instruction only if there is some serial execution order of the dag in which the read sees that write. Moreover, dag consistency allows different reads to return values that are based on different serial orders, as long as the values returned are consistent with the dependencies given by the dag. Our previous work provides a description of the model, coherence algorithms for maintaining dag consistency, and empirical evidence for their efficiency. In this paper, we analyze the execution time, page faults, and space requirements of multithreaded algorithms written with this consistency model when the execution is scheduled by the randomized work-stealing scheduler from [7, 10] and dag consistency is maintained by the BACKER coherence algorithm from [8].

A *multithreaded algorithm* is a collection of thread definitions. Analogous to a procedure definition, a thread definition is a block of serial code, possibly with conditional and looping constructions. Unlike a procedure, however, a thread definition may contain various types of "spawn" and "synchronization" statements that allow the algorithm to exhibit concurrency as follows. To specify parallelism, a thread may *spawn* child threads. A spawn is the parallel analogue of a procedure call, but in the case of a spawn, the parent and child may execute concurrently. From the time that a thread is spawned until the time that the thread returns, we say the thread is *living* or *alive*. In addition a thread may *synchronize* with some or all of its spawned children by suspending its execution until the specified children return. When the last of the specified children returns, it *enables* its parent to resume execution. A thread that is suspended waiting for children to return is said to be *stalled*, and otherwise, a thread is said to be *ready*. In general, a thread may synchronize with other threads that are not its children, but in our analysis, we shall focus on the class of *fully strict multithreaded algorithms* in which a thread may only synchronize with its children as just described.

The resource requirements of a multithreaded algorithm when used to solve a given input problem are modeled, in graph-theoretic terms, by a *multithreaded computation* [7]. A multithreaded computation is composed of two structures: a "spawn tree" of threads and a dag of instructions. The *spawn tree* of threads is the parallel analogue of a call tree of procedures. The spawn tree is rooted at the "main" thread where algorithm execution begins, and in general, each spawned thread is a node in the spawn tree with the parent-child relationships defined by the spawn operations. The dag of instructions is the parallel analogue of a serial instruction stream. We think of the dag of instructions as being "embedded" in the spawn tree, since each executed instruction is part of a spawned thread. This embedding has the following properties. All of the instructions in any given thread are totally ordered by dag edges that we call *continue* edges. For each thread, except the root thread, its first instruction has exactly one incoming edge that we call a *spawn* edge, and this edge comes from an instruction (the spawning instruction) in the parent thread. For each thread, except the root thread, its last instruction has exactly one outgoing edge that we call a *return* edge, and this edge goes to an instruction (the synchronizing instruction) in the parent thread. In the case of a fully strict multithreaded algo-

This research was supported in part by the Advanced Research Projects Agency (ARPA) under Grants N00014-94-1-0985 and N00014-92-J-1310. Robert Blumofe was supported in part by an ARPA High-Performance Computing Graduate Fellowship. Chris Joerg is now at Digital Equipment Corporations's Cambridge Research Laboratory. Charles Leiserson is currently Shaw Visiting Professor at the National University of Singapore. Keith Randall was supported in part by a Department of Defense NDSEG Fellowship.

This paper appears in the *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, June 24–26, 1996, Padua, Italy.

rithm, for any input problem, the resulting *fully strict multithreaded computation* contains only continue, spawn, and return edges as just described.

Before discussing how the BACKER coherence algorithm affects the performance of fully strict multithreaded algorithms that use dag consistent shared memory, let us first review some of the theory of multithreaded algorithms that do not use shared memory. Any multithreaded algorithm can be measured in terms of its "work" and "critical-path length" [5, 9, 10, 20]. Consider the multithreaded computation that results when a given multithreaded algorithm is used to solve a given input problem. The *work* of the computation, denoted T_1, is the number of instructions in the dag, which corresponds to the amount of time required by a one-processor execution.[1] The *critical-path length* of the computation, denoted T_∞, is the maximum number of instructions on any directed path in the dag, which corresponds to the amount of time required by an infinite-processor execution. Now, for a given multithreaded algorithm, we define the work $T_1(n)$ and critical-path length $T_\infty(n)$ of the algorithm as the work and critical-path length respectively of the multithreaded computation that results when the algorithm is used to solve an input problem of size n. With any number P of (homogeneous) processors, the time to solve a problem of size n cannot be less than $T_1(n)/P$ or less than $T_\infty(n)$. Notice that multithreaded algorithms and multithreaded computations do not specify when or on what processor to execute any instruction. Such scheduling decisions must be made at runtime, and a good runtime scheduler strives to achieve performance close to these lower bounds.

The randomized work-stealing scheduler achieves performance close to these lower bounds for the case of fully strict multithreaded algorithms that do not use shared memory. Specifically, for any such algorithm and any number P of processors, the randomized work-stealing scheduler executes the algorithm in expected time $O(T_1(n)/P + T_\infty(n))$ [7, 10]. The randomized work-stealing scheduler operates as follows. Each processor maintains a *ready deque* (doubly-ended queue) of threads from which work is obtained. When a thread is spawned, the parent thread is suspended and put on the bottom of the deque and execution commences on the spawned child thread. When a thread returns, execution of the parent resumes by removing it from the bottom of the deque. On one processor, this execution order is the standard, depth-first serial execution order. A processor that finds its deque empty becomes a "thief" and sends a steal request to a randomly chosen "victim" processor. If the victim has a thread in its deque, it sends the topmost thread to the thief to execute. Otherwise, the victim has no threads and the thief tries again with a new random victim. Finally, when a thread executing on a processor enables a thread that was stalled on another processor, the newly enabled thread is sent to the enabling processor to be resumed.

All of the threads of a multithreaded algorithm should have access to a single, shared virtual address space, and in order to support such a shared-memory abstraction on a computer with physically distributed memory, the runtime scheduler must be coupled with a coherence algorithm. For our BACKER coherence algorithm, we assume that each processor's memory is divided into two regions, each containing pages of shared-memory objects. One region is a *page cache* of C pages of objects that have been recently accessed by that processor. The rest of each processors' memory is maintained as a *backing store* of pages that have been allocated in the virtual address space. Each allocated page is assigned to the backing store of a processor chosen by hashing the page's virtual address. In order for a processor to operate on an object, the object must be resident in the processor's page cache; otherwise, a page fault occurs, and BACKER must "fetch" the object's page from backing store into the page cache. We assume that when a page fault occurs, no progress can be made on the computation during the time it takes to service the fault, and the fault time may vary due to congestion of concurrent accesses to the backing store. We shall further assume that pages in the cache are maintained using the popular LRU (least-recently-used) [19] heuristic. In addition to servicing page faults, BACKER must "reconcile" pages between the processor page caches and the backing store so that the semantics of the execution obey the assumptions of dag consistency. The BACKER coherence algorithm and the work-stealing scheduler have been implemented in the Cilk runtime system with encouraging empirical results [8].

In order to model performance for multithreaded algorithms that use dag-consistent shared memory, we observe that running times will vary as a function of the cache size C, so we must introduce measures that account for this dependence. Consider again the multithreaded computation that results when a given multithreaded algorithm is used to solve a given input problem. We shall define a new work measure, the "total work," that accounts for the cost of page faults in the serial execution, as follows. Let m be the time to service a page fault in the serial execution. We now weight the instructions of the dag. Each instruction that generates a page fault in the one-processor execution with the standard, depth-first serial execution order and with a cache of size C has weight $m+1$, and all other instructions have weight 1. The *total work*, denoted $T_1(C)$, is the total weight of all instructions in the dag, which corresponds to the serial execution time if page faults take m units of time to be serviced. We shall continue to let T_1 denote the number of instructions in the dag, but for clarity, we shall refer to T_1 as the *computational work*. (The computational work T_1 corresponds to the serial execution time if all page faults take zero time to be serviced.) To relate these measures, we define the *serial page faults*, denoted $F_1(C)$, to be the number of page faults taken in the serial execution (that is, the number of instructions with weight m). Thus, we have $T_1(C) = T_1 + mF_1(C)$. Now, for a given multithreaded algorithm, we define the total work $T_1(C,n)$ and serial page faults $F_1(C,n)$ of the algorithm as the work and serial page faults respectively of the multithreaded computation that results when the algorithm is used to solve an input problem of size n.

The quantity $T_1(C)$ is an unusual measure. Unlike T_1, it depends on the serial execution order of the computation. The quantity $T_1(C)$ further differs from T_1 in that $T_1(C)/P$ is not a lower bound on the execution time for P processors. It is possible to construct a computation containing P subcomputations that run on P separate processors in which each processor repeatedly accesses C different pages in sequence. Consequently, with caches of size C, no processor ever faults, except to warm up the cache at the start of the computation. If we run the same computation serially with a cache of size C (or any size less than CP), however, the necessary multiplexing among tasks can cause numerous page faults. Consequently, for this computation, the execution time with P processors is much less than $T_1(C)/P$. In this paper, we shall forgo the possibility of obtaining such superlinear speedup on computations. Instead, we shall simply attempt to obtain linear speedup.

Critical-path length can likewise be split into two notions. We define the *total critical-path length*, denoted $T_\infty(C)$, to be the maximum over all directed paths in the computational dag, of the time, including page faults, to execute along the path by a single processor with cache size C. The *computational critical-path length* T_∞ is the same, but where faults cost zero time. Both T_∞ and $T_\infty(C)$ are lower bounds on execution time. Although $T_\infty(C)$ is the stronger lower bound, it appears difficult to compute and analyze, and our upper-bound results will be characterized in terms of T_∞, which we shall continue to refer to simply as the critical-path length.

In this paper, we analyze the execution time of fully strict multi-

[1] For nondeterministic algorithms whose computation dag depends on the scheduler, we define T_1 to be the number of instructions in the actually occurring computation dag, and similarly for the other measures we shall define.

threaded algorithms that use dag consistent shared memory. The algorithm is executed on a parallel computer with P processors, each with a cache of size C, and a page fault that encounters no congestion is serviced in m units of time. The execution is scheduled by the work-stealing scheduler and dag consistency is maintained by the BACKER coherence algorithm. In addition, we assume that accesses to shared memory are distributed uniformly and independently over the backing store—often a plausible assumption, since BACKER hashes pages to the backing store. Then, the expected execution time to solve a problem of size n is $O(T_1(C,n)/P + mCT_\infty(n))$. In addition, we give a high-probability bound.

This result is not as strong as we would like to prove, because accesses to the backing store are not necessarily independent. For example, threads may concurrently access the same pages by algorithm design. We can artificially solve this problem by insisting, as does the EREW-PRAM model, that the algorithm performs exclusive accesses only. More seriously, however, congestion delay in accessing the backing store can cause the computation to be scheduled differently than if there were no congestion, thereby perhaps causing more congestion to occur. It may be possible to prove our bounds for a hashed backing store without making this independence assumption, but we do not know how at this time. The problem with independence does not seem to be serious in practice, and indeed, given the randomized nature of our scheduler, it is hard to conceive of how an adversary can actually take advantage of the lack of independence implied by hashing to slow the execution. Although our results are imperfect, we are actually analyzing the effects of congestion, and thus our results are much stronger than if we assume, for example, that accesses to the backing store independently suffer Poisson-distributed delays.

In this paper, we also analyze the number of page faults that occur during algorithm execution. Again, execution is scheduled with the work-stealing scheduler and dag consistency is maintained by the BACKER coherence algorithm, and we assume that accesses to backing store are random and independent. Under this assumption, we show that the expected number of page faults to solve a problem of size n on P processors, each with a LRU cache of size C, is at most $F_1(C,n) + O(CPT_\infty(n))$. In addition, for "regular" divide-and-conquer multithreaded algorithms, we derive a good upper bound on $F_1(C,n)$. For example, we show that the total number of page faults incurred by a divide-and-conquer matrix multiplication algorithm when multiplying $n \times n$ matrices using P processors is $O(n^3/(m^{3/2}\sqrt{C}) + CP\lg^2 n)$, assuming that the independence assumption for the backing store holds.

Finally, in this paper, we analyze the space requirements of "simple" multithreaded algorithms that use dag-consistent shared memory. We assume that the computation is scheduled by a scheduler, such as the work-stealing algorithm, that maintains the "busy-leaves" property [10]. For a given simple multithreaded algorithm, let $S_1(n)$ denote the space required by the standard, depth-first serial execution of the algorithm to solve a problem of size n. In previous work, we have shown that the space used by a P-processor execution is at most $S_1(n)P$ in the worst case [10]. We improve this characterization of the space requirements, and we provide a much stronger upper bound on the space requirements of regular divide-and-conquer multithreaded algorithms. For example, we show that a divide-and-conquer matrix multiplication algorithm multiplying $n \times n$ matrices on P processors uses only $\Theta(n^2P^{1/3})$ space, which is tighter than the $O(n^2P)$ result obtained by directly applying the $S_1(n)P$ bound.

The remainder of this paper is organized as follows. Section 2 gives a precise definition of dag consistency and describes the BACKER coherence algorithm for maintaining dag consistency. Section 3 analyzes the execution time of fully strict multithreaded algorithms when the execution is scheduled by the randomized work-stealing scheduler and dag consistency is maintained by the BACKER coherence algorithm. Section 4 analyzes the number of page faults taken by parallel divide-and-conquer algorithms. Section 5 analyzes the space requirements of parallel divide-and-conquer algorithms. Section 6 presents some sample analyses of algorithms that use dag-consistent shared memory. Finally, Section 7 offers some comparisons with other consistency models and some ideas for the future.

2 Dag consistency and the Backer algorithm

In this section we give a precise definition of dag consistency, and we describe the BACKER coherence algorithm for maintaining dag consistency. Dag consistency is a relaxed consistency model for distributed shared memory, and the BACKER algorithm collaborates with the work-stealing scheduler in order to maintain dag consistency for multithreaded computations executing on a parallel computer with physically distributed memory.

Shared memory consists of a set of *objects* that instructions can read and write. When an instruction performs a read of an object, it receives some value, but the particular value it receives depends upon the consistency model. Dag consistency is defined separately for each object in shared memory.

In order to give the definition, we first define some terminology. Let $G = (V,E)$ be the dag of a multithreaded computation. For $i, j \in V$, if a path of nonzero length from instruction i to j exists in G, we say that i (strictly) *precedes* j, which we write $i \prec j$. We say that two instructions $i, j \in V$ with $i \neq j$ are *incomparable* if we have $i \not\prec j$ and $j \not\prec i$. To track which instruction is responsible for an object's value, we imagine that each shared-memory object has a tag which the write operation sets to the name of the instruction performing the write. We make the technical assumption that an initial sequence of instructions writes a value to every object. We can now define dag consistency.

Definition 1 *The shared memory M of a multithreaded computation $G = (V,E)$ is* **dag consistent** *if for every location x in the shared memory, there exists a function $f_x : V \mapsto V$ such that the following conditions hold.*

1. *For all instructions $i \in V$, the instruction $f_x(i)$ writes on x.*
2. *If an instruction i writes on x, then we have $f_x(i) = i$.*
3. *If an instruction i reads x, it receives a value tagged with $f_x(i)$.*
4. *For all instructions $i \in V$, we have $i \not\prec f_x(i)$.*
5. *For each triple of instructions i, j and k, such that $i \prec j \prec k$, if $f_x(j) \neq i$ holds, then we have $f_x(k) \neq i$.*

Informally, the function $f_x(i)$ represents the viewpoint of instruction i on the contents of location x, that is, the tag of x from i's perspective. Therefore, if an instruction i writes, the tag of x becomes i (part 2 of the definition), and when it reads, it reads something tagged with $f_x(i)$ (part 3). Moreover, part 4 requires that future execution does not have any influence on the current value of the memory. The rationale behind part 5 is shown in Figure 1. When there is a path from i to k through j, then j "masks" i, in the sense that if i's view of x is no longer current when j executes, then it cannot be current when k executes. Instruction k can still have a different viewpoint on the memory than j, for instance, it can see writes performed by other instructions (such as l in the figure) incomparable with j.

In previous work [8, 17], we presented a definition of dag consistency different from Definition 1. Our Definition 1 is stronger than

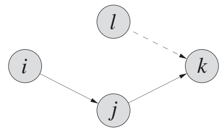

Figure 1: Illustration of the definition of dag consistency. When there is a path from i to k through j, then j "masks" i. Instruction k, however, is allowed to see writes performed by an instruction l incomparable with i and j, and possibly with k itself.

the previous definition, that is, if the shared memory M is dag consistent in the sense of Definition 1, it also satisfies the previous definition. The converse is not true. The reason we updated our definition is that Definition 1 "confines" nondeterminism, in the following sense: consider the case of two instructions i_1 and i_2, writing to a memory location x, and having a common successor j. Suppose that no instruction other than i_1 and i_2 writes to x. In Definition 1, j is forced to have a view of x, and see one of the two values. All of j's successors will then see that same value. With the old definition, two of j's successors could see different values. A full justification of Definition 1 and explanation of its properties is beyond the scope of this paper.

We now describe the BACKER coherence algorithm from [8], in which the global backing store holds shared-memory objects that are fetched into local processor caches. We state without proof a theorem that BACKER maintains dag consistency (Definition 1).

In the BACKER coherence algorithm, versions of shared-memory objects can reside simultaneously in any of the processor caches and the backing store. Each processor's cache contains objects recently used by the threads that have executed on that processor, and the backing store provides default global storage for each object. In order for a thread executing on the processor to read or write an object, the object must be in the processor's cache. Each object in the cache has a *dirty bit* to record whether the object has been modified since it was brought into the cache.

BACKER uses three basic operations to manipulate shared-memory objects: fetch, reconcile, and flush. A *fetch* copies an object from the backing store to a processor cache and marks the cached object as clean. A *reconcile* copies a dirty object from a processor cache to the backing store and marks the cached object as clean. Finally, a *flush* removes a clean object from a processor cache.

The BACKER coherence algorithm operates as follows. When the user code performs a read or write operation on an object, the operation is performed directly on a cached copy of the object. If the object is not in the cache, it is fetched from the backing store before the operation is performed. If the operation is a write, the dirty bit of the object is set. To make space in the cache for a new object, a clean object can be removed by flushing it from the cache. To remove a dirty object, it is reconciled and then flushed.

Besides performing these basic operations in response to user reads and writes, BACKER performs additional reconciles and flushes to enforce dag consistency. For each edge $i \to j$ in the computation dag, if instructions i and j are executed on different processors, say p and q, then BACKER causes p to reconcile all its cached objects after executing i but before enabling j, and it causes q to reconcile and flush its entire cache before executing j. Note that if q's cache is flushed for some other reason after p has reconciled its cache but before q executes j (perhaps because of another interprocessor dag edge), it need not be flushed again before executing j.

The following theorem states that BACKER is correct. A proof of the theorem is beyond the scope of the present paper.

Theorem 2 *If the shared memory M of a multithreaded computation is maintained using* BACKER*, then M is dag consistent.*

To understand the performance of BACKER, we posit the following environment in which BACKER operates. We have a homogeneous parallel computer with P processors. The backing store is distributed across the processors by hashing, with each processor managing a proportional share of the objects which are grouped into fixed-size pages. In addition to backing store, each processor has a cache of C pages that is maintained using the LRU replacement heuristic. When pages are transferred between processors, congestion may occur at a destination processor, in which case we assume that the transfers are serviced at the destination in FIFO (first-in, first-out) order. A multithreaded computation itself is executed by the randomized, work-stealing scheduler, but with a small technical modification described in the analysis.

3 Analysis of execution time

In this section, we bound the execution time of fully strict multithreaded algorithms when the parallel execution is scheduled by the work-stealing scheduler and dag consistency is maintained by the BACKER algorithm, under the assumption that accesses to the backing store are random and independent. Specifically, for a given fully strict multithreaded algorithm, let $T_P(C,n)$ denote the time taken by the algorithm to solve a problem of size n on a parallel computer with P processors, each with a LRU cache of C pages, when the execution is scheduled by the work-stealing scheduler in conjunction with the BACKER coherence algorithm. In addition, let $T_\infty(n)$ denote the critical-path length of the algorithm. In this section, we show that if accesses to backing store are random and independent, then the expected value of $T_P(C,n)$ is $O(T_1(C,n)/P + mCT_\infty(n))$ where m denotes the minimum time to transfer a page. In addition, we bound the number of page faults. The exposition of the proofs in this section makes heavy use of results and techniques from [10].

Our analysis of execution time is organized as follows. First, we prove a lemma describing how the BACKER algorithm adds page faults to a parallel execution. Then, we obtain a bound on the number of "rounds" that a parallel execution contains. Each round contains a fixed amount of scheduler overhead, so bounding the number of rounds bounds the total amount of scheduler overhead. To complete the analysis, we use an accounting argument to add up the total execution time.

In the following analysis, we consider the fully strict multithreaded computation that results when a given fully strict multithreaded algorithm is executed to solve a given input problem. The computation is executed by the work-stealing scheduler in conjunction with the BACKER coherence algorithm on a parallel computer with P processors. Each processor has a LRU cache of C pages. We assume that a minimum of m time steps are required to transfer a page. We let T_1 denote the computational work of the computation, $T_1(C)$ denote the total work of the computation, T_∞ denote the critical-path length of the computation, and $F_1(C)$ denote the serial page faults of the computation.

Before proceeding, we must make a slight technical modification to the work-stealing scheduler. Between successful steals, we wish to guarantee that a processor performs at least C page transfers (fetches or reconciles) so that it is not stealing too often. Consequently, whenever a processor runs out of work, if it has not performed C page transfers since its last successful steal, it performs enough additional "idle" transfers until it has transferred C pages. At that point, it can steal again. Similarly, we require that each processor perform one idle transfer after each unsuccessful steal request to ensure that steal requests do not happen too often.

We now define some new terminology. A *task* is the fundamental building block of a computation and is either a local instruction (one that does not access shared memory) or a shared-memory operation. If a task is a local instruction or references an object in the local cache, it takes 1 step to execute. Otherwise, the task is referencing an object not in the local cache, and a page transfer occurs, taking at least m steps to execute. A *synchronization* task is a task in the dag that forces BACKER to perform a cache flush in order to maintain dag consistency. Remember that for each interprocessor edge $i \rightarrow j$ in the dag, a cache flush is required by the processor executing j sometime after i executes but before j executes. A synchronization task is thus a task j having an incoming interprocessor edge $i \rightarrow j$ in the dag, where j executes on a processor that has not flushed its cache since i was executed. A *subcomputation* is the computation that one processor performs from the time it obtains work to the time it goes idle or enables a synchronization task. We distinguish two kinds of subcomputations: *primary* subcomputations start when a processor obtains work from a random steal request, and *secondary* subcomputations start when a processor starts executing from a synchronization task. We distinguish three kinds of page transfers. An *intrinsic* transfer is a transfer that would occur during a 1-processor depth-first execution of the computation. The remaining *extrinsic* page transfers are divided into two types. A *primary* transfer is any extrinsic transfer that occurs during a primary subcomputation. Likewise, a *secondary* transfer is any extrinsic transfer that occurs during a secondary subcomputation. We use these terms to refer to page faults as well.

Lemma 3 *Each primary transfer in an execution can be associated with a currently running primary subcomputation such that each primary subcomputation has at most $3C$ primary transfers associated with it. Similarly, each secondary transfer in an execution can be associated with a currently running secondary subcomputation such that each secondary subcomputation has at most $3C$ secondary transfers associated with it.*

Proof: For this proof, we use a fact shown in [8] that executing a subcomputation starting with an arbitrary cache can only incur C more page faults than the same block of code incurred in the serial execution. This fact follows from the observation that a subcomputation is executed in the same depth-first order as it would have been executed in the serial execution, and the fact that the cache replacement strategy is LRU.

We associate each primary transfer with a running primary subcomputation as follows. During a steal, we associate the (at most) C reconciles done by the victim with the stealing subcomputation. In addition, the stolen subcomputation has at most C extrinsic page faults, because the stolen subcomputation is executed in the same order as the subcomputation executes in the serial order. At the end of the subcomputation, at most C pages need be reconciled, and these reconciles may be extrinsic transfers. In total, at most $3C$ primary transfers are associated with any primary subcomputation.

A similar argument holds for secondary transfers. Each secondary subcomputation must perform at most C reconciles to flush the cache at the start of the subcomputation. The subcomputation then has at most C extrinsic page faults during its execution, because it executes in the same order as it executes in the serial order. Finally, at most C pages need to be reconciled at the end of the subcomputation. ∎

We now bound the amount of scheduler overhead by counting the number of rounds in an execution.

Lemma 4 *If each page transfer (fetch or reconcile) in the execution is serviced by a processor chosen independently at random, and each processor queues its transfer requests in FIFO order, then, for any $\varepsilon > 0$, with probability at least $1-\varepsilon$, the total number of steal requests and primary transfers is at most $O(CPT_\infty + CP\lg(1/\varepsilon))$.*

Proof: To begin, we shall assume that each access to the backing store takes one step regardless of the congestion. We shall describe how to handle congestion at the end of the proof.

First, we wish to bound the overhead of scheduling, that is, the additional work that the one-processor execution would not need to perform. We define an *event* as either the sending of a steal request or the sending of a primary-page-transfer request. In order to bound the number of events, we divide the execution into rounds. Round 1 starts at time step 1 and ends at the first time step at which at least $27CP$ events have occurred. Round 2 starts one time step after round 1 completes and ends when it contains at least $27CP$ events, and so on. We shall show that with probability at least $1-\varepsilon$, an execution contains only $O(T_\infty + \lg(1/\varepsilon))$ rounds.

To bound the number of rounds, we shall use a delay-sequence argument. We define a modified dag D' exactly as in [10]. (The dag D' is for the purposes of analysis only and has no effect on the computation.) The critical-path length of D' is at most $2T_\infty$. We define a task with no unexecuted predecessors in D' to be *critical*, and it is by construction one of the first two tasks to be stolen from the processor on which it resides. Given a task that is critical at the beginning of a round, we wish to show that it is executed by the start of the next round with constant probability. This fact will enable us to show that progress is likely to be made on any path of D' in each round.

We now show that at least $4P$ steal requests are initiated during the first $22CP$ events of a round. If at least $4P$ of the $22CP$ events are steal requests, then we are done. If not, then there are at least $18CP$ primary transfers. By Lemma 3, we know that at most $3CP$ of these transfers are associated with subcomputations running at the start of the round, leaving $15CP$ for steals that start in this round. Since at most $3C$ primary transfers can be associated with any steal, at least $5P$ steals must have occurred. At most P of these steals were requested in previous rounds, so there must be at least $4P$ steal requests in this round.

We now argue that any task that is critical at the beginning of a round has a probability of at least $1/2$ of being executed by the end of the round. Since there are at least $4P$ steal requests during the first $22CP$ events, the probability is at least $1/2$ that any task that is critical at the beginning of a round is the target of a steal request [10, Lemma 10], if it is not executed locally by the processor on which it resides. Any task takes at most $3mC+1 \leq 4mC$ time to execute, since we are ignoring the effects of congestion for the moment. Since the last $4CP$ events of a round take at least $4mC$ time to execute, if a task is stolen in the first part of the round, it is done by the end of the round.

We now want to show that with probability at least $1-\varepsilon$, the total number of rounds is $O(T_\infty + \lg(1/\varepsilon))$. Consider a possible delay sequence. Recall from [10] that a delay sequence of size R is a maximal path U in the augmented dag D' of length at most $2T_\infty$, along with a partition Π of R which represents the number of rounds during which each task of the path in D' is critical. We now show that the probability of a large delay sequence is tiny.

Whenever a task on the path U is critical at the beginning of a round, it has a probability of at least $1/2$ of being executed during the round, because it is likely to be the target of one of the $4P$ steals in the first part of the round. Furthermore, this probability is independent of the success of critical tasks in previous rounds, because victims are chosen independently at random. Thus, the probability is at most $(1/2)^{R-2T_\infty}$ that a particular delay sequence with size $R > 2T_\infty$ actually occurs in an execution. There are at most $2^{2T_\infty}\binom{R+2T_\infty}{2T_\infty}$ delay sequences of size R. Thus, the probability that any delay sequence of size R occurs is at most

$$\begin{aligned} & 2^{2T_\infty}\binom{R+2T_\infty}{2T_\infty}\left(\frac{1}{2}\right)^{R-2T_\infty} \\ \leq\ & 2^{2T_\infty}\left(\frac{e(R+2T_\infty)}{2T_\infty}\right)^{2T_\infty}\left(\frac{1}{2}\right)^{R-2T_\infty} \\ \leq\ & \left(\frac{4e(R+2T_\infty)}{2T_\infty}\right)^{2T_\infty}\left(\frac{1}{2}\right)^{R}, \end{aligned}$$

which can be made less than ε by choosing $R = 14T_\infty + \lg(1/\varepsilon)$. Therefore, there are at most $O(T_\infty + \lg(1/\varepsilon))$ rounds with probability at least $1-\varepsilon$. In each round, there are at most $28CP$ events, so there are at most $O(CPT_\infty + CP\lg(1/\varepsilon))$ steal requests and primary transfers in total.

Now, let us consider what happens when congestion occurs at the backing store. We still have at most $3C$ transfers per task, but these transfers may take more than $3mC$ time to complete because of congestion. We define the following indicator random variables to keep track of the congestion. Let x_{uip} be the indicator random variable that tells whether task u's ith transfer request is delayed by a transfer request from processor p. The probability is at most $1/P$ that one of these indicator variables is 1. Furthermore, we shall argue that they are nonpositively correlated, that is, $\Pr\left\{x_{uip} = 1 \,\middle|\, \bigwedge_{u'i'p'} x_{u'i'p'} = 1\right\} \leq 1/P$, as long as none of the (u',i') requests execute at the same time as the (u,i) request. That they are nonpositively correlated follows from an examination of the queuing behavior at the backing store. If a request (u',i') is delayed by a request from processor p' (that is, $x_{u'i'p'} = 1$), then once the (u',i') request has been serviced, processor p''s request has also been serviced, because we have FIFO queuing of transfer requests. Consequently, p''s next request, if any, goes to a new, random processor when the (u,i) request occurs. Thus, a long delay for request (u',i') cannot adversely affect the delay for request (u,i). Finally, we also have $\Pr\left\{x_{uip} = 1 \,\middle|\, \bigwedge_{p'\neq p} x_{uip'} = 1\right\} \leq 1/P$, because the requests from the other processors besides p are distributed at random.

The execution time X of the transfer requests for a path U in D' can be written as $X \leq \sum_{u\in U}(5mC + m\sum_{ip} x_{uip})$. Rearranging, we have $X \leq 10mCT_\infty + m\sum_{uip} x_{uip}$, because U has length at most $2T_\infty$. This sum is just the sum of $10CPT_\infty$ indicator random variables, each with expectation at most $1/P$. Since the tasks u in U do not execute concurrently, the x_{uip} are nonpositively correlated, and thus, their sum can be bounded using combinatorial techniques. The sum is greater than z only if some z-size subset of these $10CPT_\infty$ variables are all 1, which happens with probability:

$$\begin{aligned} \Pr\left\{\sum_{uip} x_{uip} \geq z\right\} &\leq \binom{10CPT_\infty}{z}\left(\frac{1}{P}\right)^{z} \\ &\leq \left(\frac{10eCPT_\infty}{z}\right)^{z}\left(\frac{1}{P}\right)^{z} \\ &\leq \left(\frac{10eCT_\infty}{z}\right)^{z}. \end{aligned}$$

This probability can be made less than $(1/2)^z$ by choosing $z \geq 20eCT_\infty$. Therefore, we have $X > (10+20e)mCT_\infty$ with probability at most $(1/2)^{X-10mCT_\infty}$. Since there are at most $2T_\infty$ tasks on the critical path, at most $2T_\infty + X/mC$ rounds can be overlapped by the long execution of page transfers of these critical tasks. Therefore, the probability of a delay sequence of size R is at most $(1/2)^{R-O(T_\infty)}$. Consequently, we can apply the same argument as for unit-cost transfers, with slightly different constants, to show that with probability at least $1-\varepsilon$, there are $O(T_\infty + \lg(1/\varepsilon))$ rounds, and hence $O(CPT_\infty + CP\lg(1/\varepsilon))$ events, during the execution. ■

We now bound the running time of a computation.

Theorem 5 *Consider any fully strict multithreaded computation executed on P processors, each with a LRU cache of C pages, using the work-stealing scheduler in conjunction with the* BACKER *coherence algorithm. Let m be the service time for a page fault that encounters no congestion, and assume that accesses to the backing store are random and independent. Suppose the computation has T_1 computational work, $F_1(C)$ serial page faults, $T_1(C) = T_1 + mF_1(C)$ total work, and T_∞ critical-path length. Then for any $\varepsilon > 0$, the execution time is $O(T_1(C)/P + mCT_\infty + m\lg P + mC\lg(1/\varepsilon))$ with probability at least $1-\varepsilon$. Moreover, the expected execution time is $O(T_1(C)/P + mCT_\infty)$.*

Proof: As in [10], we shall use an accounting argument to bound the running time. During the execution, at each time step, each processor puts a dollar into one of 5 buckets according to its activity at that time step. Specifically, a processor puts a dollar in the bucket labeled:

WORK if the processor executes a task;

STEAL if the processor sends a steal request;

STEALWAIT if the processor waits for a response to a steal request;

XFER if the processor sends a page-transfer request; and

XFERWAIT if the processor waits for a page transfer to complete.

When the execution completes, we add up the dollars in each bucket and divide by P to get the running time.

We bound the amount of money in each of the buckets at the end of the computation by using the fact, from Lemma 4, that with probability at least $1-\varepsilon'$, there are $O(CPT_\infty + CP\lg(1/\varepsilon'))$ events.

WORK. The WORK bucket contains exactly T_1 dollars, because there are exactly T_1 tasks in the computation.

STEAL. We know that there are $O(CPT_\infty + CP\lg(1/\varepsilon'))$ steal requests, so there are $O(CPT_\infty + CP\lg(1/\varepsilon'))$ dollars in the STEAL bucket.

STEALWAIT. We use the analysis of the *recycling game* ([10, Lemma 5]) to bound the number of dollars in the STEALWAIT bucket. The recycling game says that if N requests are distributed randomly to P processors for service, with at most P requests outstanding simultaneously, the total time waiting for the requests to complete is $O(N + P\lg P + P\lg(1/\varepsilon'))$ with probability at least $1-\varepsilon'$. Since steal requests obey the assumptions of the recycling game, if there are $O(CPT_\infty + CP\lg(1/\varepsilon'))$ steals, then the total time waiting for steal requests is $O(CPT_\infty + P\lg P + CP\lg(1/\varepsilon'))$ with probability at least $1-\varepsilon'$. We must add to this total an extra $O(mCPT_\infty + mCP\lg(1/\varepsilon'))$ dollars because the processors initiating a successful steal must also wait for the cache of the victim to be reconciled, and we know that there are $O(CPT_\infty + CP\lg(1/\varepsilon'))$ such reconciles. Finally, we must add $O(mCPT_\infty + mCP\lg(1/\varepsilon))$ dollars because each steal request might also have up to m idle steps associated with it. Thus, with probability at least $1-\varepsilon'$, we have a total of $O(mCPT_\infty + P\lg P + mCP\lg(1/\varepsilon'))$ dollars in the STEALWAIT bucket.

XFER. We know that there are $O(F_1(C) + CPT_\infty + CP\lg(1/\varepsilon'))$ transfers during the execution: a fetch and a reconcile for each intrinsic fault, $O(CPT_\infty + CP\lg(1/\varepsilon'))$ primary transfers from Lemma 4, and $O(CPT_\infty + CP\lg(1/\varepsilon'))$ secondary transfers. We have this bound on secondary transfers, because each secondary subcomputation can be paired with a unique primary subcomputation. We construct this pairing as follows. For each synchronization task j, we examine each interprocessor edge entering j. Each of these edges corresponds to some child of j's thread in the spawn tree, because the computation is fully strict. At least one of these children (call it k) is not finished executing at the time of the last cache flush by j's processor, since j is a synchronization task. We now show that there must be a random steal of j's thread just after k is spawned. If not, then k is completed before j's thread continues executing after the spawn. There must be a random steal somewhere between when k is spawned and when j is executed, however, because j and k execute on different processors. On the last such random steal, the processor executing j must flush its cache, but this cannot happen because k is still executing when the last flush of the cache occurs. Thus, there must be a random steal just after k is spawned. We pair the secondary subcomputation that starts at task j with the primary subcomputation that starts with the random steal after k is spawned. By construction, each primary subcomputation has at most one secondary subcomputation paired with it, and since each primary subcomputation does at least C extrinsic transfers and each secondary subcomputation does at most $3C$ extrinsic transfers, there are at most $O(CPT_\infty + CP\lg(1/\varepsilon'))$ secondary transfers. Since each transfer takes m time, the number of dollars in the XFER bucket is $O(mF_1(C) + mCPT_\infty + mCP\lg(1/\varepsilon'))$.

XFERWAIT. To bound the dollars in the XFERWAIT bucket, we use the recycling game as we did for the STEALWAIT bucket. The recycling game shows that there are $O(mF_1(C) + mCPT_\infty + mP\lg P + mCP\lg(1/\varepsilon'))$ dollars in the XFERWAIT bucket with probability at least $1-\varepsilon'$.

With probability at least $1-3\varepsilon'$, the sum of all the dollars in all the buckets is $T_1 + O(mF_1(C) + mCPT_\infty + mP\lg P + mCP\lg(1/\varepsilon'))$. Dividing by P, we obtain a running time of $T_P \le O((T_1 + mF_1(C))/P + mCT_\infty + m\lg P + mC\lg(1/\varepsilon'))$ with probability at least $1-3\varepsilon'$. Using the identity $T_1(C) = T_1 + mF_1(C)$ and substituting $\varepsilon = 3\varepsilon'$ yields the desired high-probability bound. The expected bound follows similarly. ∎

We now bound the number of page faults.

Corollary 6 *Consider any fully strict multithreaded computation executed on P processors, each with a LRU cache of C pages, using the work-stealing scheduler in conjunction with the* BACKER *coherence algorithm. Assume that accesses to the backing store are random and independent. Suppose the computation has $F_1(C)$ serial page faults and T_∞ critical-path length. Then for any $\varepsilon > 0$, the number of page faults is at most $F_1(C) + O(CPT_\infty + CP\lg(1/\varepsilon))$ with probability at least $1-\varepsilon$. Moreover, the expected number of page faults is at most $F_1(C) + O(CPT_\infty)$.*

Proof: In the parallel execution, we have one fault for each intrinsic fault, plus an extra $O(CPT_\infty + CP\lg(1/\varepsilon))$ primary and secondary faults. The expected bound follows similarly. ∎

4 Analysis of page faults

This section provides upper bounds on the number of page faults for "regular" divide-and-conquer multithreaded algorithms when the parallel execution is scheduled by the randomized work-stealing scheduler and dag consistency is maintained by the BACKER algorithm. In a *regular divide-and-conquer multithreaded algorithm*, each thread, when spawned to solve a problem of size n, operates as follows. To solve a problem of size n where n is larger than some constant, the thread divides the problem into a subproblems each of size n/b for some constants $a \ge 1$ and $b > 1$, and then recursively spawns child threads to solve each subproblem. When all a of the children have completed, the thread merges their results, and then returns. In the base case, when n is smaller than some constant, the thread directly solves the problem, and then returns.

In Corollary 6, we analyzed the number of page faults that a fully strict multithreaded algorithm incurs when run on P processors using the randomized work-stealing scheduler and the BACKER coherence algorithm. Specifically, for a given fully strict multithreaded algorithm, let $F_1(C,n)$ denote the number of page faults that occur when the algorithm is used to solve a problem of size n with the standard, depth-first serial execution order on a single processor with a LRU cache of C pages. In addition, for any number $P \ge 2$ of processors, let $F_P(C,n)$ denote the number of page faults that occur when the algorithm is used to solve a problem of size n with the work-stealing scheduler and BACKER on P processors, each with a LRU cache of C pages. Corollary 6 then says that the expectation of $F_P(C,n)$ is at most $F_1(C,n) + O(CPT_\infty(n))$. The $O(CPT_\infty(n))$ term represents faults due to "warming up" the processors' caches.

Generally, one must implement and run an algorithm to get a good estimate of $F_1(C,n)$ before one can predict whether it will run well in parallel. For regular divide-and-conquer multithreaded algorithms, however, analysis can provide good asymptotic bounds on $F_1(C,n)$, and hence on $F_P(C,n)$.

Theorem 7 *Consider any regular divide-and-conquer multithreaded algorithm executed on 1 processor with a LRU cache of C pages, using the standard, depth-first serial execution order. Let n_C be the largest problem size that can be solved wholly within the cache. Suppose that each thread, when spawned to solve a problem of size n larger than or equal to n_C, divides the problem into a subproblems each of size n/b for some constants $a \ge 1$ and $b > 1$. Additionally, suppose each thread solving a problem of size n makes $p(n)$ page faults in the worst case. Then, the number $F_1(C,n)$ of page faults taken by the algorithm when solving a problem of size n can be determined as follows.*[2]

1. *If $p(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $F_1(C,n) = O(C(n/n_C)^{\log_b a})$, if $p(n)$ further satisfies the regularity condition that $p(n) \le a\gamma p(n/b)$ for some constant $\gamma < 1$.*
2. *If $p(n) = \Theta(n^{\log_b a})$, then $F_1(C,n) = O(C(n/n_C)^{\log_b a}\lg(n/n_C))$.*
3. *If $p(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, then $F_1(C,n) = O(C(n/n_C)^{\log_b a} + p(n))$, if $p(n)$ further satisfies the regularity condition that $p(n) \ge a\gamma p(n/b)$ for some constant $\gamma > 1$.*

Proof: If a problem of size n does not fit in the cache, then the number $F_1(C,n)$ of faults taken by the algorithm in solving the problem is at most the number $F_1(C,n/b)$ of faults for each of the a subproblems of size n/b plus an additional $p(n)$ faults for the top thread itself. If the problem can be solved in the cache, the data for it need only be paged into memory at most once. Consequently, we obtain the recurrence

$$F_1(C,n) \le \begin{cases} aF_1(C,n/b) + p(n) & \text{if } n > n_C\,, \\ C & \text{if } n \le n_C\,. \end{cases} \qquad (1)$$

[2] Other cases exist besides the three given here.

We can solve this recurrence using standard techniques [12, Section 4.4]. We iterate the recurrence, stopping as soon as we reach the first value of the iteration count k such that $n/b^k \le n_C$ holds, or equivalently when $k = \lceil \log_b(n/n_C) \rceil$ holds. Thus, we have

$$\begin{aligned} F_1(C,n) &\le a^k F_1(C,n/b^k) + \sum_{i=0}^{k-1} a^i p(n/b^i) \\ &\le Ca^k + \sum_{i=0}^{k-1} a^i p(n/b^i) \\ &= O\left(C(n/n_C)^{\log_b a} + \sum_{i=0}^{\log_b(n/n_C)} a^i p(n/b^i) \right) . \end{aligned}$$

If $p(n)$ satisfies the conditions of Case 1, the sum is geometrically increasing and is dominated by its last term. For $p(n)$ satisfying Case 2, each term in the sum is the same. Finally, for $p(n)$ satisfying Case 3, the first term of the sum dominates. Using the inequality $p(n_C) \le C$, we get the stated results. ■

5 Analysis of space utilization

This section provides upper bounds on the memory requirements of regular divide-and-conquer multithreaded algorithms when the parallel execution is scheduled by a "busy-leaves" scheduler, such as the work-stealing scheduler used by Cilk. A *busy-leaves* scheduler is a scheduler with the property that at all times during the execution, if a thread has no living children, then that thread has a processor working on it. The work-stealing scheduler is a busy-leaves scheduler [10]. We shall proceed through a series of lemmas that provide an exact characterization of the space used by "simple" multithreaded algorithms when executed by a busy-leaves scheduler. A *simple multithreaded algorithm* is a fully strict multithreaded algorithm in which each thread's control consists of allocating memory, spawning children, waiting for the children to complete, deallocating memory, and returning, in that order. We shall then specialize this characterization to provide space bounds for regular divide-and-conquer algorithms.

Previous work [10] has shown that a busy-leaves scheduler can efficiently execute a fully strict multithreaded algorithm on P processors using no more space than P times the space required to execute the algorithm on a single processor. Specifically, for a given fully strict multithreaded algorithm, if $S_1(n)$ denotes the space used by the algorithm to solve a problem of size n with the standard, depth-first, serial execution order, then for any number P of processors, a busy leaves scheduler uses at most $PS_1(n)$ space. The basic idea in the proof of this bound is that a busy-leaves scheduler never allows more than P leaves in the spawn tree of the resulting computation to be living at one time. If we look at any path in the spawn tree from the root to a leaf and add up all the space allocated on that path, the largest such value we can obtain is $S_1(n)$. The bound then follows, because each of the at most P leaves living at any time is responsible for at most $S_1(n)$ space, for a total of $PS_1(n)$ space. For many algorithms, however, the bound $PS_1(n)$ is an overestimate of the true space, because space near the root of the spawn tree may be counted multiple times. In this section, we tighten this bound for the case of regular divide-and-conquer algorithms. We start by considering the more general case of simple multithreaded algorithms.

We first introduce some terminology. Consider any simple multithreaded algorithm and input problem, and let T be the spawn tree of the simple multithreaded computation that results when the given algorithm is executed to solve the given problem. Let Λ be any nonempty set of the leaves of T. A node (thread) $u \in T$ is *covered* by Λ if u lies on the path from some leaf in Λ to the root of T. The *cover* of Λ, denoted $\mathcal{C}(\Lambda)$, is the set of nodes covered by Λ. Since all nodes on the path from any node in $\mathcal{C}(\Lambda)$ to the root are covered, it follows that $\mathcal{C}(\Lambda)$ is connected and forms a subtree of T. If each node u allocates $f(u)$ memory, then the space used by Λ is defined as

$$\mathcal{S}(\Lambda) = \sum_{u \in \mathcal{C}(\Lambda)} f(u) .$$

The following lemma shows how the notion of a cover can be used to characterize the space required by a simple multithreaded algorithm when executed by a busy leaves scheduler.

Lemma 8 *Let T be the spawn tree of a simple multithreaded computation, and let $f(u)$ denote the memory allocated by node $u \in T$. For any number P of processors, if the computation is executed using a busy-leaves scheduler, then the total amount of allocated memory at any time during the execution is at most $\mathcal{S}^*$, which we define by the identity*

$$\mathcal{S}^* = \max_{|\Lambda| \le P} \mathcal{S}(\Lambda) ,$$

with the maximum taken over all sets Λ of leaves of T of size at most P.

Proof: Consider any given time during the execution, and let Λ denote the set of leaves living at that time, which by the busy-leaves property has cardinality at most P. The total amount of allocated memory is the sum of the memory allocated by the leaves in Λ plus the memory allocated by all their ancestors. Since both leaves and ancestors belong to $\mathcal{C}(\Lambda)$ and $|\Lambda| \le P$ holds, the lemma follows. ■

The next few definitions will help us characterize the structure of $\mathcal{C}(\Lambda)$ when Λ maximizes the space used. Let T be the spawn tree of a simple multithreaded computation, and let $f(u)$ denote the memory allocated by node $u \in T$, where we shall henceforth make the technical assumption that $f(u) = 0$ holds if u is a leaf and $f(u) > 0$ holds if u is an internal node. When necessary, we can extend the spawn tree with a new level of leaves in order to meet this technical assumption. Define the *serial-space function* $S(u)$ inductively on the nodes of T as follows:

$$S(u) = \begin{cases} 0 & \text{if } u \text{ is a leaf;} \\ f(u) + \max\{S(v) : v \text{ is a child of } u\} \\ & \text{if } u \text{ is an internal node of } T. \end{cases}$$

The serial-space function assumes a strictly increasing sequence of values on the path from any leaf to the root. Moreover, for each node $u \in T$, there exists a leaf such that if π is the unique simple path from u to that leaf, then we have $S(u) = \sum_{v \in \pi} f(v)$. We shall denote that leaf (or an arbitrary such leaf, if more than one exists) by $\lambda(u)$. The *u-induced dominator* of a set Λ of leaves of T is defined by

$$\begin{aligned} \mathcal{D}(\Lambda,u) \;=\; &\{v \in T : \exists w \in \mathcal{C}(\Lambda) \text{ such that } w \text{ is a child} \\ &\text{of } v \text{ and } S(w) < S(u) \le S(v)\} . \end{aligned}$$

The next lemma shows that every induced dominator of Λ is indeed a "dominator" of Λ.

Lemma 9 *Let T be the spawn tree of a simple multithreaded computation encompassing more than one node, and let Λ be a nonempty set of leaves of T. Then, for any internal node $u \in T$, removal of $\mathcal{D}(\Lambda,u)$ from T disconnects each leaf in Λ from the root of T.*

Proof: Let r be the root of T, and consider the path π from any leaf $l \in \Lambda$ to r. We shall show that some node on the path belongs to $\mathcal{D}(\Lambda,u)$. Since u is not a leaf and S is strictly increasing on the

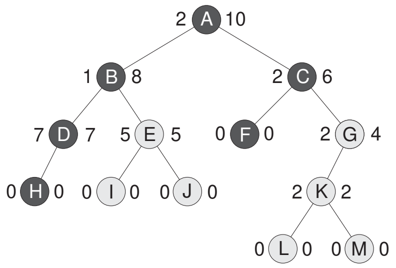

Figure 2: An illustration of the definition of a dominator set. For the tree shown, let f be given by the labels at the left of the nodes, and let $\Lambda = \{F, H\}$. Then, the serial space S is given by the labels at the right of the nodes, $\mathcal{C}(\Lambda) = \{A, B, C, D, F, H\}$ (the shaded nodes), and $\mathcal{D}(\Lambda, G) = \{C, D\}$. The space required by Λ is $\mathcal{S}(\Lambda) = 12$.

nodes of the path π, we must have $0 = S(l) < S(u) \leq S(r)$. Let w be the node lying on π that maximizes $S(w)$ such that $S(w) < S(u)$ holds, and let v be its parent. We have $S(w) < S(u) \leq S(v)$ and $w \in \mathcal{C}(\Lambda)$, because all nodes lying on π belong to $\mathcal{C}(\Lambda)$, which implies that $v \in \mathcal{D}(\Lambda, u)$ holds. ∎

The next lemma shows that whenever we have a set Λ of leaves that maximizes space, every internal node u not covered by Λ induces a dominator that is at least as large as Λ.

Lemma 10 *Let T be the spawn tree of a simple multithreaded computation encompassing more than one node, and for any integer $P \geq 1$, let Λ be a set of leaves such that $\mathcal{S}(\Lambda) = \mathcal{S}^*$ holds. Then, for all internal nodes $u \notin \mathcal{C}(\Lambda)$, we have $|\mathcal{D}(\Lambda, u)| \geq |\Lambda|$.*

Proof: Suppose, for the purpose of contradiction, that $|\mathcal{D}(\Lambda, u)| < |\Lambda|$ holds. Lemma 9 implies that each leaf in Λ is a descendant of some node in $\mathcal{D}(\Lambda, u)$. Consequently, by the pigeonhole principle, there must exist a node $v \in \mathcal{D}(\Lambda, u)$ that is ancestor of at least two leaves in Λ. By the definition of induced dominator, a child $w \in \mathcal{C}(\Lambda)$ of v must exist such that $S(w) < S(u)$ holds.

We shall now show that a new set Λ' of leaves can be constructed such that we have $\mathcal{S}(\Lambda') > \mathcal{S}(\Lambda)$, thus contradicting the assumption that $\mathcal{S}$ achieves its maximum value on Λ. Since w is covered by Λ, the subtree rooted at w must contain a leaf $l \in \Lambda$. Define $\Lambda' = \Lambda - \{l\} \cup \{\lambda(u)\}$. Adding $\lambda(u)$ to Λ causes the value of $\mathcal{S}(\Lambda)$ to increase by at least $S(u)$, and the removal of l causes the path from l to some descendant of w (possibly w itself) to be removed, thus decreasing the value of $\mathcal{S}(\Lambda)$ by at most $S(w)$. Therefore, we have $\mathcal{S}(\Lambda') \geq \mathcal{S}(\Lambda) - S(w) + S(u) > \mathcal{S}(\Lambda)$, since $S(w) < S(u)$ holds. ∎

We now restrict our attention to regular divide-and-conquer multithreaded algorithms, as introduced in Section 4. In a regular divide-and-conquer multithreaded algorithm, each thread, when spawned to solve a problem of size n, allocates an amount of space $s(n)$ for some function s of n. The following lemma characterizes the structure of the worst-case space usage for this class of algorithms.

Lemma 11 *Let T be the spawn tree of a regular divide-and-conquer multithreaded computation encompassing more than one node, and for any integer $P \geq 1$, let Λ be a set of leaves such that $\mathcal{S}(\Lambda) = \mathcal{S}^*$ holds. Then, $\mathcal{C}(\Lambda)$ contains every node at every level of the tree with P or fewer nodes.*

Proof: If T has fewer than P leaves, then Λ consists of all the leaves of T and the lemma follows trivially. Thus, we assume that T has at least P leaves, and we have $|\Lambda| = P$.

Suppose now, for the sake of contradiction, that there is a node u at a level of the tree with P or fewer nodes such that $u \notin \mathcal{C}(\Lambda)$ holds. Since all nodes at the same level of the spawn tree allocate the same amount of space, the set $\mathcal{D}(\Lambda, u)$ consists of all covered nodes at the same level as u, all of which have the same serial space $S(u)$. Lemma 10 then says that there are at least P nodes at the same level as u that are covered by Λ. This fact contradicts our assumption that the tree has P or fewer nodes at the same level as u. ∎

Finally, we state and prove a theorem that bounds the worst-case space used by a regular divide-and-conquer multithreaded algorithm when it is scheduled using a busy-leaves scheduler.

Theorem 12 *Consider any regular divide-and-conquer multithreaded algorithm executed on P processors using a busy-leaves scheduler. Suppose that each thread, when spawned to solve a problem of size n, allocates $s(n)$ space, and if n is larger than some constant, then the thread divides the problem into a subproblems each of size n/b for some constants $a \geq 1$ and $b > 1$. Then, the total amount $S_P(n)$ of space taken by the algorithm in the worst case when solving a problem of size n can be determined as follows.*[3]

1. *If $s(n) = \Theta(\lg^k n)$ for some constant $k \geq 0$, then $S_P(n) = \Theta(P \lg^{k+1}(n/P))$.*
2. *If $s(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $S_P(n) = \Theta(P s(n/P^{1/\log_b a}))$, if, in addition, $s(n)$ satisfies the regularity condition $\gamma_1 s(n/b) \leq s(n) \leq a\gamma_2 s(n/b)$ for some constants $\gamma_1 > 1$ and $\gamma_2 < 1$.*
3. *If $s(n) = \Theta(n^{\log_b a})$, then $S_P(n) = \Theta(s(n) \lg P)$.*
4. *If $s(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, then $S_P(n) = \Theta(s(n))$, if, in addition, $s(n)$ satisfies the regularity condition that $s(n) \geq a\gamma s(n/b)$ for some constant $\gamma > 1$.*

Proof: Consider the spawn tree T of the multithreaded computation that results when the algorithm is used to solve a given input problem of size n. The spawn tree T is a perfectly balanced a-ary tree. A node u at level k in the tree allocates space $f(u) = s(n/b^k)$. From Lemma 8 we know that the maximum space usage is bounded by $\mathcal{S}^*$, which we defined as the maximum value of the space function $\mathcal{S}(\Lambda)$ over all sets Λ of leaves of the spawn tree having size at most P.

In order to bound the maximum value of $\mathcal{S}(\Lambda)$, we shall appeal to Lemma 11 which characterizes the set Λ at which this maximum occurs. Lemma 11 states that for this set Λ, the set $\mathcal{C}(\Lambda)$ contains every node in the first $\lfloor \log_a P \rfloor$ levels of the spawn tree. Thus, we have

$$S_P(n) \leq \sum_{i=0}^{\lfloor \log_a P \rfloor - 1} a^i s(n/b^i) + \Theta(P S_1(n/P^{1/\log_b a})) . \qquad (2)$$

To determine which term in Equation (2) dominates, we must evaluate $S_1(n)$, which satisfies the recurrence

$$S_1(n) = S_1(n/b) + s(n) ,$$

because with serial execution the depth-first discipline allows each of the a subproblems to reuse the same space. The solution to this recurrence [12, Section 4.4] is

- $S_1(n) = \Theta(\lg^{k+1} n)$, if $s(n) = \Theta(\lg^k n)$ for some constant $k \geq 0$, and

[3] Other cases exist besides those given here.

- $S_1(n) = \Theta(s(n))$, if $s(n) = \Omega(n^\varepsilon)$ for some constant $\varepsilon > 0$ and in addition satisfies the regularity condition that $s(n) \ge \gamma s(n/b)$ for some constant $\gamma > 1$.

The theorem follows by evaluating Equation (2) for each of the cases. We only sketch the essential ideas in the algebraic manipulations. For Cases 1 and 2, the serial space dominates, and we simply substitute appropriate values for the serial space. In Cases 3 and 4, the space at the top of the spawn tree dominates. In Case 3, the total space at each level of the spawn tree is the same. In Case 4, the space at each level of the spawn tree decreases geometrically, and thus, the space allocated by the root dominates the entire tree. ∎

6 Example analyses of algorithms

In this section we show how to apply the analysis techniques of this paper to specific algorithms. We focus first on analyzing matrix multiplication, and then we examine LU-decomposition. We show that both of these matrix problems can be solved efficiently with respect to the measures of time, page faults, and space using recursive divide-and-conquer algorithms. In our analyses, we shall assume that the cache memory of each of the P processors contains C pages, and each page can hold m matrix elements. We shall also assume that the accesses to backing store behave as if they were random and independent, so that the expected bounds $T_P(C,n) = O(T_1(C,n)/P + mCT_\infty(n))$ and $F_P(C,n) = F_1(C,n) + O(CPT_\infty(n))$ are good models for the performance of algorithms.

Multiplying two $n \times n$ matrices (using the ordinary algorithm, and not a variant of Strassen's algorithm [28]) can be performed using $\Theta(n^3)$ work and can be done in $\Theta(\lg n)$ time [24]. Thus, for this problem, we have $T_1(n) = \Theta(n^3)$ and $T_\infty(n) = \Theta(\lg n)$. If there were no page faults, we would see $T_P(n) = O(n^3/P + \lg n)$.

We must also account for page faults, however. Let us consider first the number of page faults for the naive "blocked" serial algorithm for computing $C = AB$ in which the three matrices A, B, and C are partitioned into $\sqrt{m} \times \sqrt{m}$ submatrix blocks. We perform the familiar triply nested loop on the blocked matrix—indexing i through the row blocks of C, j through the column blocks of C, and k through the row blocks of A and column blocks of B—updating $C[i,j] \leftarrow C[i,j] + A[i,k] \cdot B[k,j]$ on the matrix blocks. If the matrix B does not fit into the cache, that is, $mC < n^2$, then every access to a block of B causes a page fault. Consequently, the number of serial page faults is $F_1(C,n) = (n/\sqrt{m})^3 = n^3/m^{3/2}$, even if we assume that A and C never fault.

The divide-and-conquer `matrixmul` algorithm from [8] uses the processor cache much more effectively. To multiply the $n \times n$ matrix A by similar matrix B, `matrixmul` divides each matrix into four $n/2 \times n/2$ submatrices and uses the identity

$$\begin{aligned} &\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \\ &\quad = \begin{bmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{bmatrix} + \begin{bmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{bmatrix} . \end{aligned}$$

The idea of `matrixmul` is to recursively compute the 8 products of the submatrices of A and B in parallel, and then add the subproducts together in pairs to form the result using recursive matrix addition. We can apply Theorem 7 to analyze `matrixmul` using $a = 8$, $b = 2$, $n_C = \sqrt{mC}$, and $p(n) = \Theta(n^2/m)$. Case 1 of the theorem applies with $\varepsilon = 1$, which yields $F_1(C,n) = O(C(n/\sqrt{mC})^3) = O(n^3/(m^{3/2}\sqrt{C}))$, a factor of $\sqrt{C}$ fewer faults than the naive algorithm.

To analyze the space for `matrixmul`, we use Theorem 12. For this algorithm, we obtain a recurrence with $a = 8$, $b = 2$, and $s(n) = \Theta(n^2)$. Case 2 applies, yielding a worst-case space bound of $S_P(n) = \Theta(P(n/P^{1/3})^2) = \Theta(n^2P^{1/3})$.[4]

The work and critical-path length for `matrixmul` can also be computed using recurrences. The computational work $T_1(n)$ to multiply $n \times n$ matrices satisfies $T_1(n) = 8T_1(n/2) + \Theta(n^2)$, since adding two matrices in parallel can be done using $O(n^2)$ computational work, and thus, $T_1(n) = \Theta(n^3)$. Consequently, the total work is $T_1(C,n) = T_1(n) + mF_1(C,n) = \Theta(n^3)$.

To derive a recurrence for the critical-path length $T_\infty(n)$, we observe that with an infinite number of processors, only one of the 8 submultiplications is the bottleneck, because the 8 multiplications can execute in parallel. Consequently, the critical-path length $T_\infty(n)$ satisfies $T_\infty(n) = T_\infty(n/2) + \Theta(\lg n)$, because the parallel addition can be accomplished recursively with a critical path of length $\Theta(\lg n)$. The solution to this recurrence is $T_\infty(n) = \Theta(\lg^2 n)$.

Using our performance model, the total expected time for `matrixmul` on P processors is therefore $T_P(C,n) = O(T_1(C,n)/P + mCT_\infty(n)) = O(n^3/P + mC\lg^2 n)$. Consequently, if we have $P = O(n^3/(mC\lg^2 n))$, the algorithm runs in $O(n^3/P)$ time. A parallel version of the naive algorithm has a slightly shorter critical path, and therefore it can achieve $O(n^3/P)$ time even with slightly more processors. But `matrixmul` commits fewer page faults, which in practice may mean better actual performance. Moreover, the code is more portable, because it requires no knowledge of the page size m. What is important, however, is that the performance models for dag consistency allow us to analyze algorithm behavior.

With a simple change, `matrixmul` can be modified to use no auxiliary space, but at the cost of a longer critical path. The idea is to spawn 4 of the 8 subproducts which place their results in the output matrix, wait for them to complete, and then spawn the other 4 to add their results into the output matrix. Since we must wait for the first 4 to complete, the critical-path length for this computation is $T_\infty(n) = 2T_\infty(n/2) + \Theta(1)$, which has solution $T_\infty(n) = \Theta(n)$. If the number P of processors is not too large, this algorithm may be preferable to `matrixmul` because it uses only $\Theta(n^2)$ space.

Let us now examine the more complicated problem of performing an LU-decomposition of an $n \times n$ matrix A without pivoting. The ordinary parallel algorithm for this problem pivots on the first diagonal element. Next, in parallel it updates the first column of A to be the first column of L and the first row of A to be the first row of U. Then, it forms a Schur complement to update the remainder of A, which it recursively (or iteratively) factors. This standard algorithm requires $\Theta(n^3)$ computational work and it has a critical path of length $\Theta(n)$. Unfortunately, even when implemented in a blocked fashion, the algorithm does not display good locality for a hierarchical memory system. Each step causes updates to the entire matrix, resulting in $F_1(C,n) = \Theta(n^3/m^{3/2})$ serial page faults, similar to blocked matrix multiplication.

A divide-and-conquer algorithm for the problem uses fewer page faults, at the cost of a slightly longer critical path. Divide the matrix A and its factors L and U into four parts so that $A = LU$ is written as

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} = \begin{bmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{bmatrix} \cdot \begin{bmatrix} U_{00} & U_{01} \\ 0 & U_{11} \end{bmatrix} .$$

The parallel algorithm computes L and U as follows. It recursively factors A_{00} into $L_{00} \cdot U_{00}$. Then, it uses back substitution to solve

[4] In recent work, Blelloch, Gibbons, and Matias [6] have shown that "series-parallel" dag computations can be scheduled to achieve substantially better space bounds than we report here. For example, they give a bound of $S_P(n) = O(n^2 + P\lg^2 n)$ for matrix multiplication. Their improved space bounds come at the cost of substantially more communication and overhead than is used by our scheduler, however.

for U_{01} in the formula $A_{01} = L_{00}U_{01}$, while simultaneously using forward substitution to solve for L_{10} in $A_{10} = L_{10}U_{00}$. Finally, it recursively factors the Schur complement $A_{11} - L_{10}U_{01}$ into $L_{11} \cdot U_{11}$. If the variant of `matrixmul` that uses no auxiliary storage is used in forming the Schur complement, the entire algorithm can be performed in place with no extra storage.

To analyze this algorithm, we must first understand the performance of back and forward substitution. To solve these problems on an $n \times n$ matrix, we can use a straightforward parallel divide-and-conquer algorithm to achieve computational work $\Theta(n^2)$, critical-path length $\Theta(n)$, and serial page faults $\Theta(n^2/m)$. Thus, we obtain the recurrence $T_1(n) = 2T_1(n/2) + \Theta(n^3)$ for the computational work of the algorithm, since the two recursive calls to the algorithm must execute in series and the matrix multiplication to compute the Schur complement requires $\Theta(n^3)$ computational work. This recurrence gives us a solution of $T_1(n) = \Theta(n^3)$ for the computational work. The critical-path length has recurrence $T_\infty(n) = 2T_\infty(n/2) + \Theta(n)$, which has solution $\Theta(n \lg n)$, slightly worse than the standard algorithm. More importantly, however, we can show that the number of serial page faults is $\Theta(n^3/(m^{3/2}\sqrt{C}))$. The result follows from the observation that the dominant page fault cost is due to the matrix multiplications. Therefore we can apply Case 3 of Theorem 7 with $a = b = 2$ and $p(n) = O(n^3/(m^{3/2}\sqrt{C}))$. Like `matrixmul`, many fewer page faults occur with the divide-and-conquer algorithm than with the standard algorithm, and the algorithm also uses much less communication for paging.

7 Conclusion

We briefly relate dag consistency to other distributed shared memories, and then we offer some ideas for the future.

Like Cilk's dag consistency, most distributed shared memories (DSM's) employ a relaxed consistency model in order to realize performance gains, but unlike dag consistency, most distributed shared memories take a low-level view of parallel programs and cannot give analytical performance bounds.

Relaxed shared-memory consistency models are motivated by the fact that sequential consistency [22] and various forms of processor consistency [16] are too expensive to implement in a distributed setting. (Even modern SMP's do not implement sequential consistency.) Relaxed models, such as location consistency [14] and various forms of release consistency [1, 13, 15], ensure consistency (to varying degrees) only when explicit synchronization operations occur, such as the acquisition or release of a lock. Causal memory [2] ensures consistency only to the extent that if a process A reads a value written by another process B, then all subsequent operations by A must appear to occur after the write by B. Most DSM's implement one of these relaxed consistency models [11, 18, 21, 27], though some implement a fixed collection of consistency models [4], while others merely implement a collection of mechanisms on top of which users write their own DSM consistency policies [23, 26]. All of these consistency models and the DSM's that implement these models take a low-level view of a parallel program as a collection of cooperating processes.

In contrast, dag consistency takes the high-level view of a parallel program as a dag, and this dag exactly defines the memory consistency required by the program. Like some of these other DSM's, dag consistency allows synchronization to affect only the synchronizing processors and does not require a global broadcast to update or invalidate data. Unlike these other DSM's, however, dag consistency requires no extra bookkeeping overhead to keep track of which processors might be involved in a synchronization operation, because this information is encoded explicitly in the dag. By leveraging this high-level knowledge, the BACKER algorithm in conjunction with the work-stealing scheduler is able to execute multithreaded algorithms with the performance bounds shown here. The BLAZE parallel language [25] and the Myrias parallel computer [3] define a high-level relaxed consistency model much like dag consistency, but we do not know of any efficient implementation of either of these systems. After an extensive literature search, we are aware of no other distributed shared memory with analytical performance bounds for any nontrivial algorithms.

We are currently working on various extensions of dag consistency and improvements to our implementation of dag consistency in Cilk. We are considering possible extensions to dag-consistent shared memory, since some operations are impossible to express with dag-consistent reads and writes alone. For example, concurrent threads cannot increment a shared counter with only dag-consistent reads and writes. We are considering the possibility of dag-consistent "atomic updates" in order to support such operations. In addition, the idea of dag consistency can be extended to the domain of file I/O. We anticipate that it should be possible to memory-map files and use our existing dag-consistency mechanisms to provide a parallel, asynchronous I/O capability for Cilk. We are also currently working on supporting dag-consistent shared memory in our Cilk-NOW runtime system [7] which executes Cilk programs in an adaptively parallel and fault-tolerant manner on networks of workstations. We expect that the "well-structured" nature of Cilk computations will allow the runtime system to maintain dag consistency efficiently, even in the presence of processor faults.

Finally, we observe that our work to date leaves open a number of analytical questions regarding the performance of multithreaded algorithms that use dag consistent shared memory. We would like to improve the analysis of execution time to directly account for the cost of page faults when pages are hashed to backing store instead of assuming that accesses to backing store "appear" to be independent and random as assumed here. We conjecture that the bound of Theorem 5 holds when pages are hashed to backing store provided the algorithm is EREW in the sense that concurrent threads never read or write to the same page. We would also like to obtain tight bounds on the number of page faults and the memory requirements for classes of multithreaded algorithms that are different from or more general than the class of regular divide-and-conquer algorithms analyzed here.

Acknowledgments

We gratefully acknowledge the work of Bradley C. Kuszmaul, Yuli Zhou, and Rob Miller, all formerly of MIT, for their efforts as part of the Cilk team. Thanks to the National University of Singapore for resources used to prepare this abstract. Thanks to Chee Chee Weng of Singapore's National Supercomputer Research Center for his contributions to the LU-decomposition algorithm. Thanks to Bruce Maggs of Carnegie Mellon University for his $o(1)$-time page transfers to the SPAA program committee. Thanks to Arvind and his dataflow group at MIT for helpful discussions and continual inspiration.

References

[1] Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, Seattle, Washington, May 1990.

[2] Mustaque Ahamad, Phillip W. Hutto, and Ranjit John. Implementing and programming causal distributed shared memory. In *Proceedings of the 11th International Conference on Distributed Computing systems*, pages 274–281, Arlington, Texas, May 1991.

[3] Monica Beltrametti, Kenneth Bobey, and John R. Zorbas. The control mechanism for the Myrias parallel computer system. *Computer Architecture News*, 16(4):21–30, September 1988.

[4] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway distributed shared memory system. In *Digest of Papers from the Thirty-Eighth IEEE Computer Society International Conference (Spring COMPCON)*, pages 528–537, San Francisco, California, February 1993.

[5] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3), March 1996.

[6] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, Santa Barbara, California, July 1995.

[7] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995.

[8] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. Dag-consistent distributed shared memory. In *Proceedings of the 10th International Parallel Processing Symposium*, Honolulu, Hawaii, April 1996.

[9] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[10] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, Santa Fe, New Mexico, November 1994.

[11] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 152–164, Pacific Grove, California, October 1991.

[12] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.

[13] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[14] Guang R. Gao and Vivek Sarkar. Location consistency: Stepping beyond the barriers of memory coherence and serializability. Technical Report 78, McGill University, School of Computer Science, Advanced Compilers, Architectures, and Parallel Systems (ACAPS) Laboratory, December 1993.

[15] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, Seattle, Washington, June 1990.

[16] James R. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherent Interface (SCI) Working Group, March 1989.

[17] Christopher F. Joerg. *The Cilk System for Parallel Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1996.

[18] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 213–228, Copper Mountain Resort, Colorado, December 1995.

[19] Edward G. Coffman Jr. and Peter J. Denning. *Operating Systems Theory*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1973.

[20] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science—Volume A: Algorithms and Complexity*, chapter 17, pages 869–941. MIT Press, Cambridge, Massachusetts, 1990.

[21] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *USENIX Winter 1994 Conference Proceedings*, pages 115–132, San Francisco, California, January 1994.

[22] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[23] James R. Larus, Brad Richards, and Guhan Viswanathan. LCM: Memory system support for parallel language implementation. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 208–218, San Jose, California, October 1994.

[24] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes*. Morgan Kaufmann Publishers, San Mateo, California, 1992.

[25] Piyush Mehrotra and John Van Rosendale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, 5:339–361, 1987.

[26] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, Chicago, Illinois, April 1994.

[27] Daniel J. Scales and Monica S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 101–114, Monterey, California, November 1994.

[28] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354–356, 1969.