Cilk: Efficient Multithreaded Computing

by

Keith H. Randall

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1998

Author . . . .
Department of Electrical Engineering and Computer Science
May 21, 1998

Certified by . . . .
Charles E. Leiserson
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Cilk: Efficient Multithreaded Computing

by

Keith H. Randall

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 1998, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

This thesis describes Cilk, a parallel multithreaded language for programming contemporary shared memory multiprocessors (SMP's). Cilk is a simple extension of C which provides constructs for parallel control and synchronization. Cilk imposes very low overheads — the typical cost of spawning a parallel thread is only between 2 and 6 times the cost of a C function call on a variety of contemporary machines. Many Cilk programs run on one processor with virtually no degradation compared to equivalent C programs. We present the "work-first" principle which guided the design of Cilk's scheduler and two consequences of this principle, a novel "two-clone" compilation strategy and a Dijkstra-like mutual-exclusion protocol for implementing the ready queue in the work-stealing scheduler.

To facilitate debugging of Cilk programs, Cilk provides a tool called the Nondeterminator-2 which finds nondeterministic bugs called "data races". We present two algorithms, All-Sets and Brelly, used by the Nondeterminator-2 for finding data races. The All-Sets algorithm is exact but can sometimes have poor performance; the Brelly algorithm, by imposing a locking discipline on the programmer, is guaranteed to run in nearly linear time. For a program that runs serially in time T, accesses V shared memory locations, and holds at most k locks simultaneously, Brelly runs in $O(kT\,\alpha(V,V))$ time and $O(kV)$ space, where α is Tarjan's functional inverse of Ackermann's function.

Cilk can be run on clusters of SMP's as well. We define a novel weak memory model called "dag consistency" which provides a natural consistency model for use with multithreaded languages like Cilk. We provide evidence that Backer, the protocol that implements dag consistency, is both empirically and theoretically efficient. In particular, we prove bounds on running time and communication for a Cilk program executed on top of Backer, including all costs associated with Backer itself. We believe this proof is the first of its kind in this regard. Finally, we present the MultiBacker protocol for clusters of SMP's which extends Backer to take advantage of hardware support for shared memory within an SMP.

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Computer Science and Engineering

Acknowledgments

I would like to thank Charles Leiserson, my advisor, for the guidance and intellectual challenge he has given me over the past 9 years. One seldom finds a person with such boundless energy and enthusiasm. It's been a wild ride, from all of the world travel (watching the Superbowl in Singapore at 5:00 in the morning) to all of the late night paper writing sessions. It is nice to see that with all of his time commitments, Charles still has enough free time to write a book about golf.

I would also like to thank the rest of my thesis committee, Arvind and Martin Rinard, for being both patient and flexible with my hectic schedule.

This thesis would not be possible without the efforts of the Cilk team. These Cilk "worms" include Guang-Ien Cheng, Don Dailey, MingDong Feng, Matto Frigo, Chris Joerg, Bradely Kuszmaul, Phil Lisiecki, Alberto Medina, Rob Miller, Aske Plaat, Harald Prokop, Bin Song, Andrew Stark, Volker Strumpen, and Yuli Zhou. I would especially like to thank my grad-student-in-arms Matteo Frigo for being a wonderful collaborator. The rapport between Matteo and myself has made Cilk development a joy.

I would like to thank the Air Force Office of Scientific Research (AFOSR) for their generous support of my graduate studies through a National Defense Science and Engineering Graduate (NDSEG) Fellowship. Their support gave me the independence to focus on my research without distractions. Additional support was provided by the Defense Advanced Research Projects Agency (DARPA) under Grants N00014-92-J-1310, N00014-94-1-0985, and F30602-97-1-0270. I would also like to thank Sun Microsystems and Digital Equipment Corporation for donating equipment used in this thesis.

Finally, I would like to thank my family for giving me all the opportunity in the world to pursue my dreams.

Contents

List of Figures

Chapter 1

Introduction

Multiprocessor shared-memory machines (SMP's) are rapidly becoming commodity items in the computer industry. Despite the prevalence of such machines, exploiting their computing power remains difficult because programming environments are cumbersome or inefficient.

I am a principal designer of Cilk (pronounced "silk"), a parallel multithreaded language being developed at MIT, which is designed to make high-performance parallel programming easier. This thesis describes the Cilk language, its programming environment, and its implementations. The Cilk language is a simple extension to C that provides constructs for easily expressing parallelism in an application. Cilk insulates the programmer from many low-level implementation details, such as load balancing and communication. Nevertheless, the Cilk compiler and runtime system work together to execute Cilk code efficiently on an SMP, typically with only a few percent overhead and near linear speedup.

1.1 Performance

Figure 1-1 gives some performance numbers for some sample applications written in Cilk, run on a Sun Enterprise 5000 with 8 167MHz UltraSPARC processors. Three metrics for each application are measured. The metric T_1 is the running time of the Cilk program on 1 processor, and the metric T_8 is the running time of the Cilk program

Program	*Size*	T_1/T_S	T_1/T_8	T_S/T_8
`fib`	35	3.63	8.0	2.2
`blockedmul`	1024	1.05	7.0	6.6
`notempmul`	1024	1.05	7.6	7.2
`strassen`	1024	1.01	5.7	5.6
*`cilksort`	$4,100,000$	1.21	6.0	5.0
†`queens`	22	0.99	8.0	8.0
†`knapsack`	30	1.03	8.0	7.7
`lu`	2048	1.02	7.7	7.5
*`cholesky`	BCSSTK32	1.25	6.9	5.5
`heat`	4096×512	1.08	6.6	6.1
`fft`	2^{20}	0.93	5.6	6.0
`barnes-hut`	2^{16}	1.02	7.2	7.1

Figure 1-1: The performance of some example Cilk programs. Times are are accurate to within about 10%. T_1 and T_8 are the running times of the Cilk program on 1 and 8 processors, respectively (except for the nondeterministic programs, labeled by a dagger (†), where T_1 is the actual work of the computation on 8 processors, and not the running time on 1 processor). T_S is the running time of the serial algorithm for the problem, which is the C elision for all programs except those that are starred (*), where the parallel program implements a different algorithm than the serial program. The quantity T_1/T_S gives the overhead incurred in parallelizing the program. The quantity T_1/T_8 represents the speedup on 8 processors relative to the 1 processor run, and the quantity T_S/T_8 represents the speedup relative to the serial program.

on 8 processors. The metric T_S is the running time of the best serial program for the application. For applications which did not have to be reorganized to expose their parallelism, the best serial program is just the ***C elision*** of the Cilk program, the Cilk program with all of the Cilk keywords deleted. Cilk's semantics guarantee that the C elision is a semantically correct implementation of the Cilk program. For other applications, namely `cilksort` and `cholesky`, the best serial program implements a different algorithm than the parallel Cilk code.

The column T_1/T_S in Figure 1-1 gives the ratio of the running time of the Cilk program on 1 processor to the running time of the best serial program. The quantity T_1/T_S represents the ***work overhead***, or the extra work required when converting from a serial program to a parallel program. For most applications, the work overhead is only a few percent. Only `fib`, a program to compute Fibonacci numbers, experiences a high work overhead because of its extremely short threads. Even programs that were reorganized to expose parallelism have small work overheads, despite the

	Original	Cilk	SPLASH-2
lines	1861	2019	2959
Δ lines	0	158	1098
`diff` lines	0	455	3741
T_1/T_S	1	1.024	1.099
T_1/T_8	N/A	7.2	7.2
T_S/T_8	N/A	7.1	6.6

Figure 1-2: Comparision of codes for the Barnes-Hut algorithm. The three codes listed include the original Barnes-Hut C code, the Cilk parallelization, and the SPLASH-2 parallelization. The row "Δ lines" counts the number of lines of code that were added to the original code to parallelize it. The row "`diff` lines" counts the number of lines the `diff` utility outputs when comparing the original and parallel versions of the code, a rough measure of the number of lines of code that were changed. The last three lines show a comparison of the performance of the two parallel codes. The quantity T_S for the Cilk code is just the running time of the original C Barnes-Hut code. The quantity T_S for the SPLASH-2 code was obtained by removing the parallel constructs from the SPLASH-2 code by hand. (The SPLASH-2 code cannot be directly compared to the original C code because it contains some (serial) optimizations which are not present in the original C code.)

fact that the best serial program is more efficient than the C elision.

The example Cilk programs get good speedup. The quantity T_1/T_8 is the speedup obtained by running the Cilk program on 8 processors. The important metric T_S/T_8 measures the end-to-end application speedup that a programmer can expect when parallelizing his serial application for use on an 8 processor machine. Figure 1-1 shows that Cilk gives good end-to-end speedup for a wide variety of applications.

1.2 Programmibility

Programming in Cilk is simple. To back up this claim, Figure 1-2 presents some comparisions among three versions of Barnes-Hut (the last application listed in Figure 1-1), an application which simulates the motion of galaxies under the influence of gravity. The three versions are the serial C version obtained from Barnes's web page [5], the Cilk parallelization of that code, and the SPLASH-2 parallelization [101]. SPLASH-2 is a standard parallel library similar to POSIX threads developed at Stanford. The SPLASH-2 Barnes-Hut code, derived from the serial C code [92], is part of the SPLASH-2 parallel benchmark suite.

Figure 1-2 shows some rough measures of the effort required to parallelize the original C code using both Cilk and SPLASH-2. One simple measure of effort is the number of lines that were added to the code. The entry "Δ lines" counts the number of lines that were added to the original code to parallelize it using both parallelization techniques. This measure shows that Cilk requires almost an order of magnitude less additional code than SPLASH-2. The entry "`diff` lines" gives the number of lines that were changed, measured by counting the number of lines that the UNIX `diff` utility[1] outputs when comparing the original C code to each of the parallel codes. Again, there is an order of magnitude difference between the changes required for the Cilk parallelization versus the SPLASH-2 parallelization. In fact, almost all lines of the C code had to be changed for the SPLASH-2 parallelization. The changes required in order to parallelize the application using SPLASH-2 include explicit load balancing (requiring additional indirections and load-balancing phases), passing a thread ID to every function, and using the thread ID as an index for every private variable access. Automatic load balancing and linguistic support for private variables eliminate all of this programmer overhead in the Cilk parallelization. Although the measures of coding effort used here are arguably imprecise, they nevertheless suggest that less effort is required to parallelize the original C code using Cilk than using a thread library like SPLASH-2, and the resulting parallelization obtains better end-to-end speedup.

1.3 Debugging

Parallel codes are notoriously hard to debug [80]. Much of this difficulty arises from either intentional or unintentional nondeterministic behavior of parallel programs. Unintentional nondeterminism often occurs because of a "data race", when two parallel threads holding no locks in common access the same memory location and at least

[1] `diff -w` is used to eliminate any differences in whitespace.

one of the threads modifies the location. In order to help Cilk programmers debug their parallel code, Cilk provides a parallel debugger called the Nondeterminator-2 to identify possible unintentional nondeterminism caused by data races. The Nondeterminator-2 provides two algorithms, called ALL-SETS and BRELLY, that check a Cilk ***computation***, the result of executing a Cilk program on a particular input, for data races. The ALL-SETS algorithm is exact but may be too inefficient in the worst case. The BRELLY algorithm, by imposing a simple locking discipline on the programmer, can detect data races or violations of the discipline in nearly linear time. For a program that runs serially in time T, accesses V shared memory locations, and holds at most k locks simultaneously, BRELLY runs in $O(kT\,\alpha(V,V))$ time and $O(kV)$ space, where α is Tarjan's functional inverse of Ackermann's function. Like its predecessor, the Nondeterminator (which checks for simple "determinacy" races) the Nondeterminator-2 is a debugging tool, not a verifier, since it checks for data races only in a single computation, and a program can generate many computations.

For the class of "abelian" programs, ones whose critical sections commute, however, the Nondeterminator-2 can provide a guarantee of determinacy. We prove that any abelian Cilk program produces a determinate final state if it is deadlock free and if it generates any computation which is data-race free. Thus, either of the Nondeterminator-2's two algorithms can verify the determinacy of a deadlock-free abelian program running on a given input.

1.4 Distributed implementation

For Cilk programs that require more computing power than one SMP can provide, we give a distributed version of Cilk that can run on multiple SMP's. We define a weak consistency model for shared memory called "dag consistency" and a corresponding consistency protocol called BACKER. We argue that dag consistency is a natural consistency model for Cilk programs, and we give both theoretical and empirical evidence that the BACKER algorithm efficiently implements dag consistency. In particular, we prove strong bounds on the running time and number of page faults (cache misses) of Cilk running with BACKER. For instance, we prove that the number of page faults (cache misses) incurred by BACKER running on P processors, each with a shared-memory cache of C pages, is the number of page faults of the 1-processor execution plus at most $2C$ "warm-up" faults for each procedure migration. We present empirical evidence that this warm-up overhead is actually much smaller in practice than the theoretical bound, as typically less than 3% of the possible $2C$ faults actually occur.

We define the MULTIBACKER algorithm, an extension of BACKER for clusters of SMP's. The MULTIBACKER algorithm modifies BACKER by using a unified shared-memory cache for each SMP and implements a "local bias" scheduling policy for improving scheduling locality. We have implementations of BACKER on the Connection Machine CM5 and MULTIBACKER on clusters of Sun and Digital SMP's.

1.5 Contributions of this thesis

This thesis advocates the use of Cilk as a programming environment for parallel computers. This thesis supports this advocacy through the following contributions:

- *The Cilk language.* Cilk provides simple yet powerful constructs for expressing parallelism in an application. Important concepts like the C elision and novel features like "implicit atomicity" provide the programmer with parallel semantics that are easy to understand and use.

- *An efficient implementation of the Cilk language on an SMP.* A principled compilation and runtime strategy is proposed and used to guide the implementation of Cilk-5, our latest Cilk release. Figure 1-1 shows that a wide range of applications obtain good performance when written in Cilk. Speedups relative to the best serial programs are quite good, reflecting both the low work overhead and good speedup of the parallel program.

- *Demonstrated performance of Cilk on a variety of realistic parallel applications.* This thesis presents a set of interesting applications in Cilk to show Cilk's applicability and efficiency across a variety of problem domains. These applications include a Barnes-Hut algorithm, sparse and dense matrix algorithms including LU decomposition and Cholesky factorization, and the world's fastest Rubik's cube solver.

- *A parallel debugging tool called the Nondeterminator-2.* The Nondeterminator-2 checks a parallel program for data races, which are a potential source of unwanted nondeterminism. We present two algorithms, All-Sets and Brelly, used by the Nondeterminator-2 for finding data races in a Cilk computation. The All-Sets algorithm is exact but can sometimes have poor performance; the Brelly algorithm, by imposing a locking discipline on the programmer, is guaranteed to run in nearly linear time.

- *A proof that the Nondeterminator-2 can guarantee determinacy for the class of abelian program.* Although the Nondeterminator-2 is guaranteed to find any data races in a Cilk computation, different runs of a Cilk program on the same input may generate different computations, and thus the Nondeterminator-2 cannot in general guarantee determinacy of the program on that input. We prove, however, that for the class of abelian programs, the Nondeterminator-2 can provide a guarantee. Specifically, if the Nondeterminator-2 does not find a data race in a single execution of a deadlock-free abelian program run on a given input, then that progrm is determinate (always generates the same final memory state) for that input.

- *A weak memory-consistency model called "dag consistency" and* Backer*, a protocol that implements dag consistency.* Dag consistency is a novel memory model that escapes from the standard "processor-centric" style of definition to a more general "computation-centric" definition. Dag consistency provides a natural consistency model for use with multithreaded languages like Cilk. Backer, the protocol that implements dag consistency, is a simple protocol that nevertheless provides good performance on the Connection Machine CM5 for a variety of Cilk applications.

- *A proof that* BACKER*, together with the work-stealing scheduler from [16], gives provably good performance.* By weakening the memory consistency model to dag consistency, we are able to use BACKER, a memory consistency protocol that we can analyze theoretically. We are able to prove bounds on the running time and communication use of a Cilk program executed on top of BACKER, with all the costs of the protocol for maintaining memory consistency included. We believe this proof is the first of its kind in this regard.

- *An extension of* BACKER*, called* MULTIBACKER*, for multi-level memory hierarchies like those found in clusters of SMP's.* The MULTIBACKER protocol extends BACKER to take advantage of hardware support for shared memory within an SMP while still using the theoretically and empirically efficient BACKER protocol between SMP's. MULTIBACKER is implemented in our latest distributed version of Cilk for clusters of SMP's.

The remainder of this thesis is logically composed of three parts. The first part, Chapter 2, describes the Cilk language. We describe how parallelism and synchronization are expressed using `spawn` and `sync` statements, and we show how nondeterminism can be expressed using inlets and an abort mechanism. We describe a performance model for Cilk programs in terms of "work" and "critical path" that allows the user to predict the performance of his programs on a parallel computer.

The second part of the thesis, Chapters 3, 4, and 5, describes our Cilk-5 system for contemporary shared-memory SMP's. Chapter 3 describes our implementation of the Cilk language and the techniques we use to reduce the scheduling overhead. In Chapter 4 we describe some realistic applications that we have coded in Cilk and show how these applications illuminate many of Cilk's features. Lastly, in Chapter 5, we describe the Nondeterminator-2 debugging tool for Cilk programs.

In the third part of the thesis, Chapters 6, 7, and 8, we describe the implementation of Cilk on distributed platforms. In Chapter 6 we describe dag consistency, our memory model, and BACKER, its implementation. In Chapter 7, we prove bounds on the running time and communication used by Cilk programs running with BACKER.

We also demonstrate how these bounds can be applied to many of the Cilk applications. Finally, in Chapter 8 we describe the latest distributed version of Cilk for clusters of SMP's, and outline MultiBacker, our multilevel consistency protocol.

Chapter 2

The Cilk language

This chapter presents a overview of the Cilk extensions to C as supported by Cilk-5. (For a complete description, consult the Cilk-5 manual [24].) The key features of the language are the specification of parallelism and synchronization, through the `spawn` and `sync` keywords, and the specification of nondeterminism, using `inlet` and `abort`. In this chapter, we also describe a simple performance model based on "work" and "critical path" which allows a Cilk programmer to predict the performance of his Cilk programs on parallel machines.

2.1 The history of Cilk

Cilk is a multithreaded language for parallel programming that generalizes the semantics of C by introducing linguistic constructs for parallel control. The original Cilk-1 release [11, 14, 58] featured a provably efficient, randomized, "work-stealing" scheduler [11, 16], but the language was clumsy, because parallelism was exposed "by hand" using explicit continuation passing. The Cilk-2 language provided better linguistic support by allowing the user to write code using natural "spawn" and "sync" keywords, which the compiler then converted to the Cilk-1 continuation-passing form automatically.

With the introduction of Cilk-3, shared memory was added to the Cilk language. Shared memory was provided by our BACKER algorithm between processors on the

Connection Machine CM5. The shared memory support in Cilk-3 was explicit, requiring the user to denote which pointers referenced shared objects and which pointers referenced private objects. Only explicitly allocated memory (from the stack or heap) could be shared in Cilk-3.

The Cilk-4 and Cilk-5 languages witnessed the transition from the Connection Machine CM5 as our main development platform to an Enterprise 5000 UltraSPARC SMP. With shared memory provided in hardware, much of the explicit shared memory support could be made implicit, simplifying the language and enabling sharing of all memory, including global and stack variables.

The Cilk language implemented by our latest Cilk-5 release [24] still uses a theoretically efficient scheduler, but the language has been simplified considerably. It employs call/return semantics for parallelism and features a linguistically simple "inlet" mechanism for nondeterministic control. Cilk-5 is designed to run efficiently on contemporary symmetric multiprocessors (SMP's), which feature hardware support for shared memory. We have coded many applications in Cilk, including scientific applications like Barnes-Hut, dense and sparse linear algebra, and others. We have also coded non-scientific applications including a Rubik's cube solver, raytracing and radiosity programs, and the ⋆Socrates and Cilkchess chess-playing programs which have won prizes in international competitions.

The philosophy behind Cilk development has been to make the Cilk language a true parallel extension of C, both semantically and with respect to performance. On a parallel computer, Cilk control constructs allow the program to execute in parallel. When we elide the Cilk keywords for parallel control to create the C elision, however, a syntactically and semantically correct C program results. Cilk is a ***faithful*** extension of C, because the C elision of a Cilk program is a correct implementation of the semantics of the program. Moreover, on one processor, a parallel Cilk program "scales down" to run nearly as fast as its C elision.

The remainder of this chapter describes the features of the Cilk-5 language and a simple performance model which allows programmers to reason about the performance of their program. The intent of this chapter is to give a feel for the simplicity of the Cilk language. Readers familiar with the Cilk language and its performance model of "work" and "critical path" can skip to Chapter 3.

2.2 Spawn and sync

The basic Cilk language can be understood from an example. Figure 2-1 shows a Cilk program that computes the nth Fibonacci number.[1] Observe that the program would be an ordinary C program if the three keywords `cilk`, `spawn`, and `sync` are elided.

The type qualifier `cilk` identifies `fib` as a ***Cilk procedure***, which is the parallel analog to a C function. Parallelism is introduced within a Cilk procedure when the keyword `spawn` preceeds the invocation of a child procedure. The semantics of a spawn differs from a C function call only in that the parent procedure instance can continue to execute in parallel with the child procedure instance, instead of waiting for the child to complete as is done in C. Cilk's scheduler takes the responsibility of scheduling the spawned procedure instances on the processors of the parallel computer.

A Cilk procedure cannot safely use the values returned by its children until it executes a `sync` statement. The `sync` statement is a local "barrier," not a global one as, for example, is used in message-passing programming. In the Fibonacci example, a `sync` statement is required before the statement `return (x+y)` to avoid the anomaly that would occur if `x` and `y` are summed before they are computed. In addition to explicit synchronization provided by the `sync` statement, every Cilk procedure syncs implicitly before it returns, thus ensuring that all of its children terminate before it does.

Cilk imposes the following restrictions on the appearance of the keywords `cilk`, `spawn`, and `sync` in a Cilk program. The `spawn` and `sync` keywords can appear only in Cilk procedures, that is C functions annotated with the `cilk` keyword. A `spawn` statement can spawn only Cilk procedures, not C functions, and Cilk procedures can

[1] This program uses an inefficient algorithm which runs in exponential time. Although logarithmic-time methods are known [26, p. 850], this program nevertheless provides a good didactic example.

```
#include <stdlib.h>
#include <stdio.h>
#include <cilk.h>

cilk int fib (int n)
{
    if (n<2) return n;
    else {
        int x, y;
        x = spawn fib (n-1);
        y = spawn fib (n-2);
        sync;
        return (x+y);
    }
}

cilk int main (int argc, char *argv[])
{
    int n, result;
    n = atoi(argv[1]);
    result = spawn fib(n);
    sync;
    printf ("Result: %d\n", result);
    return 0;
}
```

Figure 2-1: A simple Cilk program to compute the nth Fibonacci number in parallel (using a very bad algorithm).

be invoked only via `spawn` statements.[2] These restrictions ensure that parallel code is invoked with parallel `spawn` calls and serial code is invoked with regular C calls.

2.3 Inlets

Ordinarily, when a spawned procedure returns, the returned value is simply stored into a variable in its parent's frame:

```
x = spawn foo(y);
```

[2]The keyword `cilk` enables static checking of this condition. Functions which are spawned must have the `cilk` qualifier in their type, and functions which are called must not have the `cilk` qualifier in their type.

```
cilk int fib (int n)
{
    int x = 0;
    inlet void summer (int result)
    {
        x += result;
        return;
    }

    if (n<2) return n;
    else {
        summer(spawn fib (n-1));
        summer(spawn fib (n-2));
        sync;
        return (x);
    }
}
```

Figure 2-2: Using an inlet to compute the nth Fibonnaci number.

Occasionally, one would like to incorporate the returned value into the parent's frame in a more complex way. Cilk provides an ***inlet*** feature for this purpose, which was inspired in part by the inlet feature of TAM [28].

An inlet is essentially a C function internal to a Cilk procedure. In the normal syntax of Cilk, the spawning of a procedure must occur as a separate statement and not in an expression. An exception is made to this rule if the spawn is performed as an argument to an inlet call. In this case, the procedure is spawned, and when it returns, the inlet is invoked. In the meantime, control of the parent procedure proceeds to the statement following the inlet call. In principle, inlets can take multiple spawned arguments, but Cilk-5 has the restriction that exactly one argument to an inlet may be spawned and that this argument must be the first argument. If necessary, this restriction is easy to program around.

Figure 2-2 illustrates how the `fib()` function might be coded using inlets. The inlet `summer()` is defined to take a returned value `result` and add it to the variable `x` in the frame of the procedure that does the spawning. All the variables of `fib()`

are available within `summer()`, since it is an internal function of `fib()`.[3]

No lock is required around the accesses to `x` by `summer`, because Cilk provides atomicity implicitly. The concern is that the two updates might occur in parallel, and if atomicity is not imposed, an update might be lost. Cilk provides implicit atomicity among the "threads" of a procedure instance, where a ***thread*** is a maximal sequence of instructions not containing a `spawn` or `sync` statement, or a return (either explicit or implicit) from a Cilk procedure. Threads are determined dynamically at runtime based on the actual control flow of a procedure instance. An inlet is precluded from containing `spawn` and `sync` statements (as it is only allowed to contain C code), and thus it operates atomically as a single thread. Implicit atomicity simplifies reasoning about concurrency and nondeterminism without requiring locking, declaration of critical regions, and the like.

Cilk provides syntactic sugar to produce certain commonly used inlets implicitly. For example, the statement `x += spawn fib(n-1)` conceptually generates an inlet similar to the one in Figure 2-2.

2.4 Abort

Sometimes, a procedure spawns off parallel work which it later discovers is unnecessary. This "speculative" work can be aborted in Cilk using the `abort` primitive inside an inlet. A common use of `abort` occurs during a parallel search, where many possibilities are searched in parallel. As soon as a solution is found by one of the searches, one wishes to abort any currently executing searches as soon as possible so as not to waste processor resources. The `abort` statement, when executed inside an inlet, causes all of the already-spawned children of the enclosing procedure instance to terminate.

[3]The C elision of a Cilk program with inlets is not ANSI C, because ANSI C does not support internal C functions. Cilk is based on Gnu C technology, however, which does provide this support.

```
typedef struct
{
    Cilk_lockvar lock;
    int count;
} entry;

cilk void histogram(int *elements, int num, entry *hist)
{
    if (num == 1) {
        entry *e = &hist[*elements];
        Cilk_lock(&e->lock);
        e->count++;
        Cilk_unlock(&e->lock);
    }
    else {
        spawn histogram(elements, num/2, hist);
        spawn histogram(elements + num/2, num - num/2, hist);
        sync;
    }
}
```

Figure 2-3: A Cilk procedure to compute the histogram of an array of elements. This procedure uses locks to protect parallel accesses to the `count` variables.

2.5 Locks

Release 5.1 of Cilk provides the user with mutual-exclusion locks. A lock has type `Cilk_lockvar`. The two operations on locks are `Cilk_lock` to acquire a lock, and `Cilk_unlock` to release a lock. Both functions take a single argument which is a pointer to an object of type `Cilk_lockvar`. Any number of locks may be held simultaneously. For a given lock `A`, the sequence of instructions from a `Cilk_lock(&A)` to its corresponding `Cilk_unlock(&A)` is called a ***critical section***, and we say that all accesses in the critical section are ***protected*** by lock `A`. Cilk guarantees that critical sections locked by the same lock act atomically with respect to each other. Acquiring and releasing a Cilk lock has the memory consistency semantics of release consistency [55, p. 716]. Locks must be initialized using the function `Cilk_lock_init`.

An example procedure that uses locks is shown in Figure 2-3. This program computes a simple histogram of the elements in the array `elements`. Locks are used to protect parallel updates to the `count` fields of the `hist` array.

Locks were a reluctant addition to the Cilk language. Locks force the programmer to follow a protocol, and therefore they make programming more difficult. The programmer needs to make sure his locking strategy is deadlock free and that it guards all of the appropriate accesses. The Nondeterminator-2 helps with some of these issues, but it is not a panacea. Unfortunately, we currently have no other mechanism for programmers to modify in parallel shared data structures other than with locks. In future research, we hope to provide a more principled mechanism to provide atomic operations on shared data.

Also, locks are not yet supported on our distributed implementations of Cilk. The stronger memory semantics of locks, together with the arbitrary nature of their acquire and release pattern, potentially make locking on a distributed platform very costly. We are currently investigating techniques to add locks to our distributed versions of Cilk.

2.6 Shared and private variables

Each global variable declaration in Cilk is shared by default. That is, all references to a global variable in a program refer to the same memory location. This memory location's value is maintained by default in a dag-consistent fashion, or in a stronger fashion if the hardware supports it. A shared declaration can be requested explicitly using the `shared` keyword.

Sometimes, however, it is useful to declare a variable which is "private". A private variable is declared by adding the keyword `private` to a variable declaration. Each thread in a Cilk program receives its own logical copy of each private variable. Each private variable is initialized to an undefined value when the thread starts. Once initialized, however, private variables cannot be changed except by the owning thread (unlike shared variables which can be changed by simultaneously executing threads). Private variables are useful for communicating between a Cilk thread and C functions it calls, because these C functions are completely contained in the Cilk thread. An

```
private char alternates[10][MAXWORDLEN];

int checkword(const char *word)
{
    /* Check spelling of <word>.  If spelling is correct,
     * return 0.  Otherwise, put up to 10 alternate spellings
     * in <alternates> array.  Return number of alternate spellings.
     */
}

cilk void spellcheck(const char **wordarray, int num)
{
    if (num == 1) {
        int alt = checkword(*wordarray);
        if (alt) {
            /* Print <alt> entries from <alternates> array as
             * possible correct spellings for the word <*wordarray>.
             */
        }
    }
    else {
        spawn spellcheck(wordarray, num/2);
        spawn spellcheck(wordarray + num/2, num - num/2);
        sync;
    }
}
```

Figure 2-4: An example of the use of private variables. The procedure `spellcheck` checks the spellings of the words in `wordarray` and prints alternate spellings for any words that are misspelled. The private variable `alternates` is used to pass alternate spellings from the C function `checkword` to the Cilk function `spellcheck`. If the variable `alternates` was shared, one instance of the `checkword` function could overwrite the alternate spellings that another instance had generated.

example of the use of private variables is shown in Figure 2-4.

2.7 Cactus stack

Some means of allocating memory must be provided in any useful implementation of a language with shared memory. We implement a heap allocator along the lines of C's `malloc` and `free`, but many times a simpler allocation model suffices. Cilk provides stack-like allocation in what is called a ***cactus-stack*** [52, 77, 94] to handle these simple allocations.

From the point of view of a single Cilk procedure, a cactus stack behaves much like

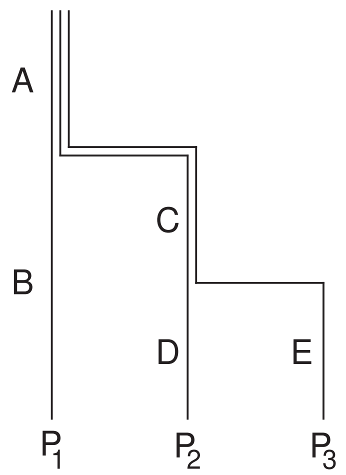


Figure 2-5: A cactus stack. Procedure P_1 spawns P_2, and P_2 spawns P_3. Each procedure sees its own stack allocations and the stack allocated by its ancestors. The stack grows downwards. In this example, the stack segment A is shared by all procedures, stack segment C is shared by procedures P_2 and P_3, and the other segments, B, D, and E, are private.

an ordinary stack. The procedure can access memory allocated by any of its ancestors in the "spawn tree" of the Cilk program. A procedure can itself allocate memory and pass pointers to that memory to its children. The cactus stack provides a natural place to allocate procedure local variables as well as memory explicitly allocated with a parallel version of `alloca`.

The stack becomes a cactus stack when multiple procedures execute in parallel, each with its own view of the stack that corresponds to its call history, as shown in Figure 2-5. In the figure, procedure P_1 allocates some memory A before procedure P_2 is spawned. Procedure P_1 then continues to allocate more memory B. When procedure P_2 allocates memory from the cactus stack, a new branch of the stack is started so that allocations performed by P_2 do not interfere with the stack being used by P_1. The stacks as seen by P_1 and P_2 are independent below the spawn point, but they are identical above the spawn point. Similarly, when procedure P_3 is spawned by P_2, the cactus stack branches again.

Cactus stacks have many of the same limitations as ordinary procedure stacks [77]. For instance, a child procedure cannot return to its parent a pointer to an object that it has allocated from the cactus stack. Similarly, sibling procedures cannot share

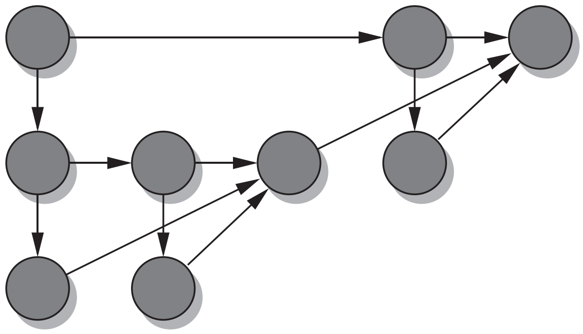


Figure 2-6: Dag of threads generated by the computation `fib(3)` from Figure 2-1. The threads of each procedure instance are ordered by horizontal ***continue*** edges. Downward ***spawn*** edges connect a procedure with its spawned child, and upward ***return*** edges connect the last thread of a procedure with the next sync in its parent procedure.

storage that they create on the stack. Just as with a procedure stack, pointers to objects allocated on the cactus-stack can be safely passed only to procedures below the allocation point in the call tree.

2.8 Computation model

The computation of a Cilk program on a given input can be viewed as a directed acyclic graph, or ***dag***, in which vertices are instructions and edges denote ordering constraints imposed by control statements. A Cilk `spawn` statement generates a vertex with out-degree 2, and a Cilk `sync` statement generates a vertex whose in-degree is 1 plus the number of subprocedures syncing at that point. Normal execution of serial code results in a linear chain of vertices, which can be grouped into threads. A computation can therefore be viewed either as a dag of instructions or a dag of threads. For example, the computation generated by the execution of `fib(3)` from the program in Figure 2-1 generates the thread dag shown in Figure 2-6.

Any computation can be measured in terms of its “work” and “critical-path length” [9, 15, 16, 60]. Consider the computation that results when a given Cilk program 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.[4] 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.

The theoretical analysis presented in [11, 16] cites two fundamental lower bounds as to how fast a Cilk program can run. For a computation with T_1 work, the lower bound $T_P \geq T_1/P$ must hold, because at most P units of work can be executed in a single step. In addition, the lower bound $T_P \geq T_\infty$ must hold, since a finite number of processors cannot execute faster than an infinite number.[5]

Cilk's randomized work-stealing scheduler [11, 16] executes a Cilk computation that does not use locks[6] on P processors in expected time

$$T_P = T_1/P + O(T_\infty) \ , \tag{2.1}$$

assuming an ideal parallel computer. This equation resembles "Brent's theorem" [19, 48] and is optimal to within a constant factor, since T_1/P and T_∞ are both lower bounds. We call the first term on the right-hand side of Equation (2.1) the ***work*** term and the second term the ***critical-path*** term. This simple performance model allows the programmer to reason about the performance of his Cilk program by examining the two simple quantities, work and critical-path, exhibited by his application. If the programmer knows the work and critical path of his application, he can use Equation (2.1) to predict its performance. Conveniently, the Cilk runtime system can measure the work and critical path of an application for the programmer. We shall revisit the running time bound in Equation (2.1) many times in this thesis

---

[4]For nondeterministic programs whose computation dag depends on the scheduler, we define T_1 to be the number of instructions that actually occur in the computation dag, and we define other measures similarly. This definition means, however, that T_1 does not necessarily correspond to the running time of the nondeterministic program on 1 processor.

[5]This abstract model of execution time ignores real-life details, such as memory-hierarchy effects, but is nonetheless quite accurate [14].

[6]The Cilk computation model provides no guarantees for scheduling performance when a program contains lock statements. If lock contention is low, however, the performance bounds stated here should still apply.

when examining our various Cilk implementations.

2.9 Memory model

To precisely define the behavior of a Cilk program, we must define a "memory model", which specifies the semantics of memory operations such as read and write. Every implementation of Cilk is guaranteed to provide at least dag-consistent shared memory. The definition of dag consistency is techical in nature and is discussed in detail in Chapter 6. Intuitively, a read can "see" a write in the dag-consistency model only if there is some serial execution order consistent with the dag in which the read sees the write. Two different locations, however, can observe different orderings of the dag. Dag consistency provides a natural "minimal" consistency model that is useful for many programs. As stated in Section 2.5, the use of locks in a Cilk program guarantees the stronger release consistency memory model for locations protected by a lock.

Of course, on machines where a stronger memory model is supported, the programmer may use that stronger consistency. Any program written assuming a stronger memory model than dag consistency, however, may not be portable across all Cilk implementations.

Chapter 3

SMP Cilk

This chapter describes our SMP implementation of Cilk-5.[1] Cilk-5 uses the same provably good "work-stealing" scheduling algorithm found in earlier versions of Cilk, but the compiler and runtime system have been completely redesigned. The efficiency of Cilk-5 was aided by a clear strategy that arose from the Cilk performance model: concentrate on minimizing overheads that contribute to the work, even at the expense of overheads that contribute to the critical path. Although it may seem counterintuitive to move overheads onto the critical path, this "work-first" principle has led to a portable Cilk-5 implementation in which the typical cost of spawning a parallel thread is only between 2 and 6 times the cost of a C function call on a variety of contemporary machines. Many Cilk programs run on one processor with virtually no degradation compared to their C elision. This chapter describes how the work-first principle was exploited in the design of Cilk-5's compiler and its runtime system. In particular, we present Cilk-5's novel "two-clone" compilation strategy and its Dijkstra-like mutual-exclusion protocol for implementing the ready deque in the work-stealing scheduler.

Unlike in Cilk-1, where the Cilk scheduler was an identifiable piece of code, in Cilk-5 both the compiler and runtime system bear the responsibility for scheduling. Cilk-5's compiler `cilk2c` is a source-to-source translator [74, 24] which converts the

[1] The contents of this chapter are joint work with Matteo Frigo and Charles Leiserson and will appear at PLDI'98 [41].

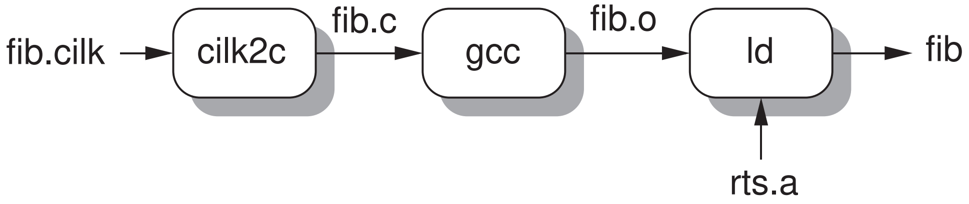


Figure 3-1: Generating an executable from a Cilk program. Our compiler `cilk2c` translates Cilk code into regular C code which we then compile with `gcc`. The result is linked with our runtime system library to create an executable.

Cilk constructs into regular C code. As shown in Figure 3-1, a Cilk executable is created from a Cilk program by first preprocessing the program using `cilk2c`, compiling the result with `gcc`,[2] and then linking with our runtime system. Importantly, our `cilk2c` source-to-source translator is machine independent and does not need to be changed when porting from one machine to another. Machine dependencies are isolated to one machine-dependent file in the runtime system.

To obtain an efficient implementation of Cilk, we have, of course, attempted to reduce scheduling overheads. Some overheads have a larger impact on execution time than others, however. A theoretical understanding of Cilk's scheduling algorithm and the performance model of work and critical path from Section 2.8 has allowed us to identify and optimize the common cases. Within Cilk's scheduler, we can identify a given cost as contributing to either work overhead or critical-path overhead. Much of the efficiency of Cilk derives from the following principle, which we shall justify in Section 3.1.

> ***The work-first principle:*** *Minimize the scheduling overhead borne by the work of a computation. Specifically, move overheads out of the work and onto the critical path.*

This principle is called the work-first principle because it emphasizes the importance of minimizing the work overhead in relation to other overheads. The work-first principle played an important role during the design of earlier Cilk systems, but Cilk-5 exploits the principle more extensively.

[2]We use some `gcc` extensions in the output of `cilk2c` which tie us to the `gcc` compiler. We hope to remedy this situation in the future.

The work-first principle inspired a "two-clone" strategy for compiling Cilk programs. Our `cilk2c` source-to-source translator produces two clones of every Cilk procedure—a "fast" clone and a "slow" clone. The fast clone, which is identical in most respects to the C elision of the Cilk program, executes in the common case where serial semantics suffice. The slow clone is executed in the infrequent case that parallel semantics and its concomitant bookkeeping are required. All communication due to scheduling occurs in the slow clone and contributes to critical-path overhead, but not to work overhead.

The work-first principle also inspired a Dijkstra-like [29], shared-memory, mutual-exclusion protocol as part of the runtime load-balancing scheduler. Cilk's scheduler uses a "work-stealing" algorithm in which idle processors, called ***thieves***, "steal" threads from busy processors, called ***victims***. Cilk's scheduler guarantees that the cost of stealing contributes only to critical-path overhead, and not to work overhead. Nevertheless, it is hard to avoid the mutual-exclusion costs incurred by a potential victim, which contribute to work. To minimize work overhead, instead of using locking, Cilk's runtime system uses a Dijkstra-like protocol, which we call the ***THE*** protocol, to manage the runtime deque of ready threads in the work-stealing algorithm. An added advantage of the THE protocol is that it allows an exception to be signaled to a working processor with no additional work overhead, a feature used in Cilk's abort mechanism.

The remainder of this chapter is organized as follows. Section 3.1 justifies the work-first principle. Section 3.2 describes how the two-clone strategy is implemented, and Section 3.3 presents the THE protocol. Section 3.4 gives empirical evidence that the Cilk-5 scheduler is efficient. Finally, Section 3.5 presents related work and offers some conclusions.

3.1 The work-first principle

This section justifies the work-first principle by showing that it follows from three assumptions. First, we assume that Cilk's scheduler operates in practice according to the theoretical model presented in Section 2.8. Second, we assume that in the common case, ample "parallel slackness" [99] exists, that is, the average parallelism of a Cilk program exceeds the number of processors on which we run it by a sufficient margin. Third, we assume (as is indeed the case) that every Cilk program has a C elision against which its one-processor performance can be measured.

As shown in Section 2.8, Cilk's randomized work-stealing scheduler executes a Cilk computation on P processors in expected time $T_P = T_1/P + O(T_\infty)$. Importantly, all communication costs due to Cilk's scheduler are borne by the critical-path term [11, 16], as are most of the other scheduling costs. To make these overheads explicit, we define the ***critical-path overhead*** to be the smallest constant c_∞ such that the following equation holds for all programs:

$$T_P \leq T_1/P + c_\infty T_\infty \ . \tag{3.1}$$

The second assumption needed to justify the work-first principle focuses on the "common-case" regime in which a parallel program operates. Define the ***average parallelism*** as $\overline{P} = T_1/T_\infty$, which corresponds to the maximum possible speedup that the application can obtain. Define also the ***parallel slackness*** [99] to be the ratio $\overline{P}/P$. The ***assumption of parallel slackness*** is that $\overline{P}/P \gg c_\infty$, which means that the number P of processors is much smaller than the average parallelism $\overline{P}$. Under this assumption, it follows that $T_1/P \gg c_\infty T_\infty$, and hence from Inequality (3.1) that $T_P \approx T_1/P$, and we obtain linear speedup. The critical-path overhead c_∞ has little effect on performance when sufficient slackness exists, although it does determine how much slackness must exist to ensure linear speedup.

Whether substantial slackness exists in common applications is a matter of opinion and empiricism, but we suggest that slackness is the common case. The expressiveness of Cilk makes it easy to code applications with large amounts of parallelism. For modest-sized problems, many applications exhibit an average parallelism of over 200, yielding substantial slackness on contemporary SMP's. Even on Sandia National Laboratory's Intel Paragon, which contains 1824 nodes, the $\star$Socrates chess program (coded in Cilk-1) ran in its linear-speedup regime during the 1995 ICCA World Computer Chess Championship (where it placed second in a field of 24). Section 3.4 describes a dozen other diverse applications which were run on an 8-processor SMP with considerable parallel slackness. The parallelism of these applications increases with problem size, thereby ensuring they will run well on large machines.

The third assumption behind the work-first principle is that every Cilk program has a C elision against which its one-processor performance can be measured. Let us denote by T_{S} the running time of the C elision. Then, we define the ***work overhead*** by $c_1 = T_1/T_{\mathrm{S}}$. Incorporating critical-path and work overheads into Inequality (3.1) yields

$$\begin{aligned} T_P &\leq c_1 T_{\mathrm{S}}/P + c_\infty T_\infty \qquad (3.2) \\ &\approx c_1 T_{\mathrm{S}}/P \ , \end{aligned}$$

since we assume parallel slackness.

We can now restate the work-first principle precisely. *Minimize c_1, even at the expense of a larger c_∞*, because c_1 has a more direct impact on performance. Adopting the work-first principle may adversely affect the ability of an application to scale up, however, if the critical-path overhead c_∞ is too large. But, as we shall see in Section 3.4, critical-path overhead is reasonably small in Cilk-5, and many applications can be coded with large amounts of parallelism.

The work-first principle pervades the Cilk-5 implementation. The work-stealing scheduler guarantees that with high probability, only $O(PT_\infty)$ steal (migration) attempts occur (that is, $O(T_\infty)$ on average per processor), all costs for which are borne on the critical path. Consequently, the scheduler for Cilk-5 postpones as much of the scheduling cost as possible to when work is being stolen, thereby removing it as a contributor to work overhead. This strategy of amortizing costs against steal attempts permeates virtually every decision made in the design of the scheduler.

3.2 Cilk's compilation strategy

This section describes how our `cilk2c` compiler generates C postsource from a Cilk program. As dictated by the work-first principle, our compiler and scheduler are designed to reduce the work overhead as much as possible. Our strategy is to generate two clones of each procedure—a ***fast*** clone and a ***slow*** clone. The fast clone operates much as does the C elision and has little support for parallelism. The slow clone has full support for parallelism, along with its concomitant overhead. We first describe the Cilk scheduling algorithm. Then, we describe how the compiler translates the Cilk language constructs into code for the fast and slow clones of each procedure. Lastly, we describe how the runtime system links together the actions of the fast and slow clones to produce a complete Cilk implementation.

As in lazy task creation [76], in Cilk-5 each processor, called a ***worker***, maintains a ***ready deque*** (doubly-ended queue) of ready procedures (technically, procedure instances). Each deque has two ends, a ***head*** and a ***tail***, from which procedures can be added or removed. A worker operates locally on the tail of its own deque, treating it much as C treats its call stack, pushing and popping activation frames. When a worker runs out of work, it becomes a ***thief*** and attempts to steal a procedure instance from another worker, called its ***victim***. The thief steals the procedure from the head of the victim's deque, the opposite end from which the victim is working.

When a procedure is spawned, the fast clone runs. Whenever a thief steals a procedure, however, the procedure is converted to a slow clone.[3] The Cilk scheduler guarantees that the number of steals is small when sufficient slackness exists, and so we expect the fast clones to be executed most of the time. Thus, the work-first principle reduces to minimizing costs in the fast clone, which contribute more heavily to work overhead. Minimizing costs in the slow clone, although a desirable goal, is less important, since these costs contribute less heavily to work overhead and more to critical-path overhead.

---

[3] Cilk procedures can be stolen when they are suspended at a spawn statement, so the slow clone must be able to start executing a Cilk procedure at any spawn or sync point.

```
int fib (int n)
{
    fib_frame *f;                   frame pointer
    f = alloc(sizeof(*f));          allocate frame
    f->sig = fib_sig;               initialize frame
    if (n<2) {
        free(f, sizeof(*f));        free frame
        return n;
    }
    else {
        int x, y;
        f->entry = 1;               save PC
        f->n = n;                   save live vars
        push(f);                    push frame
        x = fib (n-1);              do C call
        if (pop(x) == FAILURE)      pop frame
            return 0;               frame stolen
        ...                         second spawn
        ;                           sync is free!
        free(f, sizeof(*f));        free frame
        return (x+y);
    }
}
```

Figure 3-2: The fast clone generated by `cilk2c` for the `fib` procedure from Figure 2-1. The code for the second spawn is omitted. The functions `alloc` and `free` are inlined calls to the runtime system's fast memory allocator. The signature `fib_sig` contains a description of the `fib` procedure, including a pointer to the slow clone. The `push` and `pop` calls are operations on the scheduling deque and are described in detail in Section 3.3.

We minimize the costs of the fast clone by exploiting the structure of the Cilk scheduler. Because we convert a procedure instance to its slow clone when it is stolen, we maintain the invariant that a fast clone has never been stolen. Furthermore, none of the descendants of a fast clone have been stolen either, since the strategy of stealing from the heads of ready deques guarantees that parents are stolen before their children. As we shall see, this simple fact allows many optimizations to be performed in the fast clone.

We now describe how our `cilk2c` compiler generates postsource C code for the `fib` procedure from Figure 2-1. An example of the postsource for the fast clone of `fib` is given in Figure 3-2. The generated C code has the same general structure as the C elision, with a few additional statements. In lines 4–5, an ***activation frame*** is allocated for `fib` and initialized. The Cilk runtime system uses activation frames to represent procedure instances. Using techniques similar to [49, 50], our inlined allocator typically takes only a few cycles. The frame is initialized in line 5 by storing a pointer to a static structure, called a signature, describing `fib`.

The first spawn in `fib` is translated into lines 12–17. In lines 12–13, the state of the `fib` procedure is saved into the activation frame. The saved state includes the program counter, encoded as an entry number, and all live, dirty variables. Then, the frame is pushed on the runtime deque in line 14. Next, we call the `fib` routine as we would in C. Because the `spawn` statement itself compiles directly to its C elision, the postsource can exploit the optimization capabilities of the C compiler, including its ability to pass arguments and receive return values in registers rather than in memory.

After `fib` returns, lines 16–17 check to see whether the parent procedure instance has been stolen. If it has, then the scheduling deque is empty, and we return to the runtime system by returning immediately with a dummy return value. Since all of the ancestors have been stolen as well, the C stack quickly unwinds and control is returned to the runtime system.[4] The protocol to check whether the parent procedure has been stolen is quite subtle—we postpone discussion of its implementation to Section 3.3. If the parent procedure has not been stolen, however, it continues to execute at line 18, performing the second spawn, which is not shown.

In the fast clone, all `sync` statements compile to no-ops. Because a fast clone never has any children when it is executing, we know at compile time that all previously spawned procedures have completed. Thus, no operations are required for a `sync` statement, as it always succeeds. For example, line 19 in Figure 3-2, the translation of the `sync` statement is just the empty statement. Finally, in lines 20–21, `fib` deallocates the activation frame and returns the computed result to its parent

[4]The `setjmp/longjmp` facility of C could have been used as well, but our unwinding strategy is simpler.

procedure.

The slow clone is similar to the fast clone except that it provides support for parallel execution. When a procedure is stolen, control has been suspended between two of the procedure's threads, that is, at a spawn or sync point. When the slow clone is resumed, it uses a `goto` statement to restore the program counter, and then it restores local variable state from the activation frame. A `spawn` statement is translated in the slow clone just as in the fast clone. For a `sync` statement, `cilk2c` inserts a call to the runtime system, which checks to see whether the procedure has any spawned children that have not returned. Although the parallel bookkeeping in a slow clone is substantial, it contributes little to work overhead, since slow clones are rarely executed.

The separation between fast clones and slow clones also allows us to compile inlets and abort statements efficiently in the fast clone. An inlet call compiles as efficiently as an ordinary spawn. For example, the code for the inlet call from Figure 2-2 compiles similarly to the following Cilk code:

```
tmp = spawn fib(n-1);
summer(tmp);
```

Implicit inlet calls, such as `x += spawn fib(n-1)`, compile directly to their C elisions. An `abort` statement compiles to a no-op just as a `sync` statement does, because while it is executing, a fast clone has no children to abort.

The runtime system provides the glue between the fast and slow clones that makes the whole system work. It includes protocols for stealing procedures, returning values between processors, executing inlets, aborting computation subtrees, and the like. All of the costs of these protocols can be amortized against the critical path, so their overhead does not significantly affect the running time when sufficient parallel slackness exists. The portion of the stealing protocol executed by the worker contributes to work overhead, however, thereby warranting a careful implementation. We discuss this protocol in detail in Section 3.3.

The work overhead of a `spawn` in Cilk-5 is only a few reads and writes in the fast clone—3 reads and 5 writes for the `fib` example. We will experimentally quantify the work overhead in Section 3.4. Some work overheads still remain in our implementation, however, including the allocation and freeing of activation frames, saving state before a spawn, pushing and popping of the frame on the deque, and checking if a procedure has been stolen. A portion of this work overhead is due to the fact that Cilk-5 is duplicating the work the C compiler performs, but as Section 3.4 shows, this overhead is small. Although a production Cilk compiler might be able eliminate this unnecessary work, it would likely compromise portability.

In Cilk-4, the precursor to Cilk-5, we took the work-first principle to the extreme. Cilk-4 performed stack-based allocation of activation frames, since the work overhead of stack allocation is smaller than the overhead of heap allocation. Because of the cactus stack semantics of the Cilk stack (see Section 2.7), however, Cilk-4 had to manage the virtual-memory map on each processor explicitly, as was done in [94]. The work overhead in Cilk-4 for frame allocation was little more than that of incrementing the stack pointer, but whenever the stack pointer overflowed a page, an expensive user-level interrupt ensued, during which Cilk-4 would modify the memory map. Unfortunately, the operating-system mechanisms supporting these operations were too slow and unpredictable, and the possibility of a page fault in critical sections led to complicated protocols. Even though these overheads could be charged to the critical-path term, in practice, they became so large that the critical-path term contributed significantly to the running time, thereby violating the assumption of parallel slackness. A one-processor execution of a program was indeed fast, but insufficient slackness sometimes resulted in poor parallel performance.

In Cilk-5, we simplified the allocation of activation frames by simply using a heap. In the common case, a frame is allocated by removing it from a free list. Deallocation is performed by inserting the frame into the free list. No user-level management of virtual memory is required, except for the initial setup of shared memory. Heap allocation contributes only slightly more than stack allocation to the work overhead, but it saves substantially on the critical path term. On the downside, heap allocation can potentially waste more memory than stack allocation due to fragmentation. For a careful analysis of the relative merits of stack and heap based allocation that supports heap allocation, see the paper by Appel and Shao [3]. For an equally careful analysis that supports stack allocation, see [73].

Thus, although the work-first principle gives a general understanding of where overheads should be borne, our experience with Cilk-4 showed that large enough critical-path overheads can tip the scales to the point where the assumptions underlying the principle no longer hold. We believe that Cilk-5 work overhead is nearly as low as possible, given our goal of generating portable C output from our compiler. Other researchers have been able to reduce overheads even more, however, at the expense of portability. For example, lazy threads [46] obtains efficiency at the expense of implementing its own calling conventions, stack layouts, etc. Although we could in principle incorporate such machine-dependent techniques into our compiler, we feel that Cilk-5 strikes a good balance between performance and portability. We also feel that the current overheads are sufficiently low that other problems, notably minimizing overheads for data synchronization, deserve more attention.

3.3 Implemention of work-stealing

In this section, we describe Cilk-5's work-stealing mechanism, which is based on a Dijkstra-like [29], shared-memory, mutual-exclusion protocol called the "THE" protocol. In accordance with the work-first principle, this protocol has been designed to minimize work overhead. For example, on a 167-megahertz UltraSPARC I, the `fib` program with the THE protocol runs about 25% faster than with hardware locking primitives. We first present a simplified version of the protocol. Then, we discuss the actual implementation, which allows exceptions to be signaled with no additional overhead.

Several straightforward mechanisms might be considered to implement a work-stealing protocol. For example, a thief might interrupt a worker and demand attention from this victim. This strategy presents problems for two reasons. First, the mechanisms for signaling interrupts are slow, and although an interrupt would be borne on the critical path, its large cost could threaten the assumption of parallel slackness. Second, the worker would necessarily incur some overhead on the work term to ensure that it could be safely interrupted in a critical section. As an alternative to sending interrupts, thieves could post steal requests, and workers could periodically poll for them. Once again, however, a cost accrues to the work overhead, this time for polling. Techniques are known that can limit the overhead of polling [36], but they require the support of a sophisticated compiler.

The work-first principle suggests that it is reasonable to put substantial effort into minimizing work overhead in the work-stealing protocol. Since Cilk-5 is designed for shared-memory machines, we chose to implement work-stealing through shared-memory, rather than with message-passing, as might otherwise be appropriate for a distributed-memory implementation. In our implementation, both victim and thief operate directly through shared memory on the victim's ready deque. The crucial issue is how to resolve the race condition that arises when a thief tries to steal the same frame that its victim is attempting to pop. One simple solution is to add a lock to the deque using relatively heavyweight hardware primitives like Compare-And-Swap or Test-And-Set. Whenever a thief or worker wishes to remove a frame from the deque, it first grabs the lock. This solution has the same fundamental problem as the interrupt and polling mechanisms just described, however. Whenever a worker pops a frame, it pays the heavy price to grab a lock, which contributes to work overhead.

Consequently, we adopted a solution that employs Dijkstra's protocol for mutual exclusion [29], which assumes only that reads and writes are atomic. Because our protocol uses three atomic shared variables `T`, `H`, and `E`, we call it the ***THE*** protocol. The key idea is that actions by the worker on the tail of the queue contribute to work overhead, while actions by thieves on the head of the queue contribute only to critical-path overhead. Therefore, in accordance with the work-first principle, we attempt to move costs from the worker to the thief. To arbitrate among different thieves attempting to steal from the same victim, we use a hardware lock, since this overhead can be amortized against the critical path. To resolve conflicts between a worker and the sole thief holding the lock, however, we use a lightweight Dijkstra-like protocol which contributes minimally to work overhead. A worker resorts to a heavyweight hardware lock only when it encounters an actual conflict with a thief, in which case we can charge the overhead that the victim incurs to the critical path.

In the rest of this section, we describe the THE protocol in detail. We first present a simplified protocol that uses only two shared variables `T` and `H` designating the tail and the head of the deque, respectively. Later, we extend the protocol with a third variable `E` that allows exceptions to be signaled to a worker. The exception mechanism is used to implement Cilk's `abort` statement. Interestingly, this extension does not introduce any additional work overhead.

The pseudocode of the simplified THE protocol is shown in Figure 3-3. Assume that shared memory is sequentially consistent [63].[5] The code assumes that the ready deque is implemented as an array of frames. The head and tail of the deque are determined by two indices `T` and `H`, which are stored in shared memory and are visible to all processors. The index `T` points to the first unused element in the array, and `H` points to the first frame on the deque. Indices grow from the head towards the tail so that under normal conditions, we have $\mathtt{T} \geq \mathtt{H}$. Moreover, each deque has a lock `L` implemented with atomic hardware primitives or with OS calls.

The worker uses the deque as a stack. (See Section 3.2.) Before a `spawn`, it pushes a frame onto the tail of the deque. After a `spawn`, it pops the frame, unless the frame has been stolen. A thief attempts to steal the frame at the head of the deque. Only one thief at the time may steal from the deque, since a thief grabs `L` as its first action. As can be seen from the code, the worker alters `T` but not `H`, whereas the thief only increments `H` and does not alter `T`.

The only possible interaction between a thief and its victim occurs when the thief is incrementing `H` while the victim is decrementing `T`. Consequently, it is always safe for a worker to append a new frame at the end of the deque (`push`) without worrying about the actions of the thief. For a `pop` operations, there are three cases, which are shown in Figure 3-4. In case (a), the thief and the victim can both get a frame from

[5] If the shared memory is not sequentially consistent, a memory fence must be inserted between lines 5 and 6 of the worker/victim code and between lines 3 and 4 of the thief code to ensure that these instructions are executed in the proper order.

Worker/Victim

```
push(frame *f) {
  deque[T] = f;
  T++;
}

pop() {
  T--;
  if (H > T) {
    T++;
    lock(L);
    T--;
    if (H > T) {
      T++;
      unlock(L);
      return FAILURE;
    }
    unlock(L);
  }
  return SUCCESS;
}
```

Thief

```
steal() {
  lock(L);
  H++;
  if (H > T) {
    H--;
    unlock(L);
    return FAILURE;
  }
  unlock(L);
  return SUCCESS;
}
```

Figure 3-3: Pseudocode of a simplified version of the THE protocol. The left part of the figure shows the actions performed by the victim, and the right part shows the actions of the thief. None of the actions besides reads and writes are assumed to be atomic. For example, `T--;` can be implemented as `tmp = T; tmp = tmp - 1; T = tmp;`.

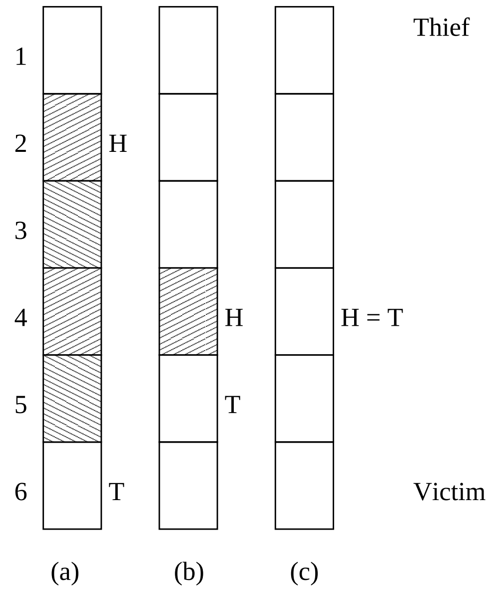


Figure 3-4: The three cases of the ready deque in the simplified THE protocol. A shaded entry indicates the presence of a frame at a certain position in the deque. The head and the tail are marked by `T` and `H`.

the deque. In case (b), the deque contains only one frame. If the victim decrements `T` without interference from thieves, it gets the frame. Similarly, a thief can steal the frame as long as its victim is not trying to obtain it. If both the thief and the victim try to grab the frame, however, the protocol guarantees that at least one of them discovers that $\mathtt{H} > \mathtt{T}$. If the thief discovers that $\mathtt{H} > \mathtt{T}$, it restores `H` to its original value and retreats. If the victim discovers that $\mathtt{H} > \mathtt{T}$, it restores `T` to its original value and restarts the protocol after having acquired `L`. With `L` acquired, no thief can steal from this deque so the victim can pop the frame without interference (if the frame is still there). Finally, in case (c) the deque is empty. If a thief tries to steal, it will always fail. If the victim tries to pop, the attempt fails and control returns to the Cilk runtime system. The protocol cannot deadlock, because each process holds only one lock at a time.

We now argue that the THE protocol contributes little to the work overhead. Pushing a frame involves no overhead beyond updating `T`. In the common case where a worker can succesfully pop a frame, the pop protocol performs only 6 operations—2 memory loads, 1 memory store, 1 decrement, 1 comparison, and 1 (predictable) conditional branch. Moreover, in the common case where no thief operates on the deque, both `H` and `T` can be cached exclusively by the worker. The expensive operation of a worker grabbing the lock `L` occurs only when a thief is simultaneously trying to steal the frame being popped. Since the number of steal attempts depends on T_∞, not on T_1, the relatively heavy cost of a victim grabbing `L` can be considered as part of the critical-path overhead c_∞ and does not influence the work overhead c_1.

We ran some experiments to determine the relative performance of the THE protocol versus the straightforward protocol in which `pop` just locks the deque before accessing it. On a 167-megahertz UltraSPARC I, the THE protocol is about 25% faster than the simple locking protocol. This machine's memory model requires that a memory fence instruction (`membar`) be inserted between lines 6 and 7 of the `pop` pseudocode. We tried to quantify the performance impact of the `membar` instruction, but in all our experiments the execution times of the code with and without `membar` are about the same. On a 200-megahertz Pentium Pro running Linux and `gcc` 2.7.1, the THE protocol is only about 5% faster than the locking protocol. On this processor, the THE protocol spends about half of its time in the memory fence.

Because it replaces locks with memory synchronization, the THE protocol is more "nonblocking" than a straightforward locking protocol. Consequently, the THE protocol is less prone to problems that arise when spin locks are used extensively. For example, even if a worker is suspended by the operating system during the execution of `pop`, the infrequency of locking in the THE protocol means that a thief can usually complete a steal operation on the worker's deque. Recent work by Arora et al. [4] has shown that a completely nonblocking work-stealing scheduler can be implemented. Using these ideas, Lisiecki and Medina [68] have modified the Cilk-5 scheduler to make it completely nonblocking. Their experience is that the THE protocol greatly simplifies a nonblocking implementation.

The simplified THE protocol can be extended to support the signaling of exceptions to a worker. In Figure 3-3, the index `H` plays two roles: it marks the head of the deque, and it marks the point that the worker cannot cross when it pops. These places in the deque need not be the same. In the full THE protocol, we separate the two functions of H into two variables: H, which now only marks the head of the deque, and E, which marks the point that the victim cannot cross. Whenever $E > T$, some exceptional condition has occurred, which includes the frame being stolen, but it can also be used for other exceptions. For example, setting $E = \infty$ causes the worker to discover the exception at its next pop. In the new protocol, E replaces H in line 7 of the worker/victim. Moreover, lines 8–16 of the worker/victim are replaced by a call to an ***exception handler*** to determine the type of exception (stolen frame or otherwise) and the proper action to perform. The thief code is also modified. Before trying to steal, the thief increments E. If there is nothing to steal, the thief restores E to the original value. Otherwise, the thief steals frame H and increments H. From the point of view of a worker, the common case is the same as in the simplified protocol: it compares two pointers (E and T rather than H and T).

The exception mechanism is used to implement `abort`. When a Cilk procedure executes an `abort` instruction, the runtime system serially walks the tree of outstanding descendants of that procedure. It marks the descendants as aborted and signals an abort exception on any processor working on a descendant. At its next `pop`, a processor working on an aborted computation will discover the exception, notice that it has been aborted, and cause all of its procedure instances to return immediately. It is conceivable that a processor could run for a long time without executing a `pop` and discovering that it has been aborted. We made the design decision to accept the possibility of this unlikely scenario, figuring that more cycles were likely to be lost in work overhead if we abandoned the THE protocol for a mechanism that solves this minor problem.

3.4 Benchmarks

In this section, we evaluate the performance of Cilk-5. We show that on 12 applications, the work overhead c_1 is close to 1, which indicates that the Cilk-5 implementation exploits the work-first principle effectively. We then present a breakdown of Cilk's work overhead c_1 on four machines. Finally, we present experiments showing

Program	*Size*	T_1	T_∞	$\overline{P}$	c_1	T_8	T_1/T_8	T_S/T_8
`fib`	35	12.77	0.0005	25540	3.63	1.60	8.0	2.2
`blockedmul`	1024	29.9	0.0044	6730	1.05	4.3	7.0	6.6
`notempmul`	1024	29.7	0.015	1970	1.05	3.9	7.6	7.2
`strassen`	1024	20.2	0.58	35	1.01	3.54	5.7	5.6
*`cilksort`	$4,100,000$	5.4	0.0049	1108	1.21	0.90	6.0	5.0
†`queens`	22	150.	0.0015	96898	0.99	18.8	8.0	8.0
†`knapsack`	30	75.8	0.0014	54143	1.03	9.5	8.0	7.7
`lu`	2048	155.8	0.42	370	1.02	20.3	7.7	7.5
*`cholesky`	BCSSTK32	1427.	3.4	420	1.25	208.	6.9	5.5
`heat`	4096×512	62.3	0.16	384	1.08	9.4	6.6	6.1
`fft`	2^{20}	4.3	0.0020	2145	0.93	0.77	5.6	6.0
`barnes-hut`	2^{16}	108.	0.15	720	1.02	14.8	7.2	7.1

Figure 3-5: The performance of some example Cilk programs. Times are in seconds and are accurate to within about 10%. The serial programs are C elisions of the Cilk programs, except for those programs that are starred (*), where the parallel program implements a different algorithm than the serial program. Programs labeled by a dagger (†) are nondeterministic, and thus, the running time on one processor is not the same as the work performed by the computation. For these programs, the value for T_1 indicates the actual work of the computation on 8 processors, and not the running time on one processor.

that the critical-path overhead c_∞ is reasonably small as well.

Figure 3-5 shows a table of performance measurements taken for 12 Cilk programs on a Sun Enterprise 5000 SMP with 8 167-megahertz UltraSPARC processors, each with 512 kilobytes of L2 cache, 16 kilobytes each of L1 data and instruction caches, running Solaris 2.5. We compiled our programs with `gcc` 2.7.2 at optimization level `-O3`. For a full description of these programs, see the Cilk 5.1 manual [24]. The table shows the work of each Cilk program T_1, the critical path T_∞, and the two derived quantities $\overline{P}$ and c_1. The table also lists the running time T_8 on 8 processors, and the speedup T_1/T_8 relative to the one-processor execution time, and speedup T_S/T_8 relative to the serial execution time.

For the 12 programs, the average parallelism $\overline{P}$ is in most cases quite large relative to the number of processors on a typical SMP. These measurements validate our assumption of parallel slackness, which implies that the work term dominates in Inequality (3.3). For instance, on 1024×1024 matrices, `notempmul` runs with an average parallelism of 1970—yielding adequate parallel slackness for up to several hundred processors. For even larger machines, one normally would not run such a small problem. For `notempmul`, as well as the other 11 applications, the average parallelism grows with problem size, and thus sufficient parallel slackness is likely to exist even for much larger machines, as long as the problem sizes are scaled appropriately.[6]

The work overhead c_1 is only a few percent larger than 1 for most programs, which shows that our design of Cilk-5 faithfully implements the work-first principle. The two cases where the work overhead is larger (`cilksort` and `cholesky`) are due to the fact that we had to change the serial algorithm to obtain a parallel algorithm, and thus the comparison is not against the C elision. For example, the serial C algorithm for sorting is an in-place quicksort, but the parallel algorithm `cilksort` requires an additional temporary array which adds overhead beyond the overhead of Cilk itself. Similarly, our parallel Cholesky factorization (see Section 4.2 for details of this algorithm) uses a quadtree representation of the sparse matrix, which induces more work than the linked-list representation used in the serial C algorithm. Finally, the work overhead for `fib` is large, because `fib` does essentially no work besides spawning procedures. Thus, the overhead $c_1 = 3.63$ for `fib` gives a good estimate of the cost of a Cilk `spawn` versus a traditional C function call. With such a small overhead for spawning, one can understand why for most of the other applications, which perform significant work for each spawn, the overhead of Cilk-5's scheduling is barely noticeable compared to the 10% "noise" in our measurements.

We now present a breakdown of Cilk's serial overhead c_1 into its components. Because scheduling overheads are small for most programs, we perform our analysis with the `fib` program from Figure 2-1. This program is unusually sensitive to scheduling overheads, because it contains little actual computation. We give a breakdown of the serial overhead into three components: the overhead of saving state before spawning, the overhead of allocating activation frames, and the overhead of the THE protocol.

Figure 3-6 shows the breakdown of Cilk's serial overhead for `fib` on four machines.

[6]Our analysis of average parallelism is somewhat suspect because it assumes an ideal memory system. In the real world, the work, critical path, and average parallelism of an application can change as the costs of the application's memory operations vary with the number of processors. Nevertheless, our performance metrics give a rough estimate of the scalability of applications on machines with adequate memory bandwidth.

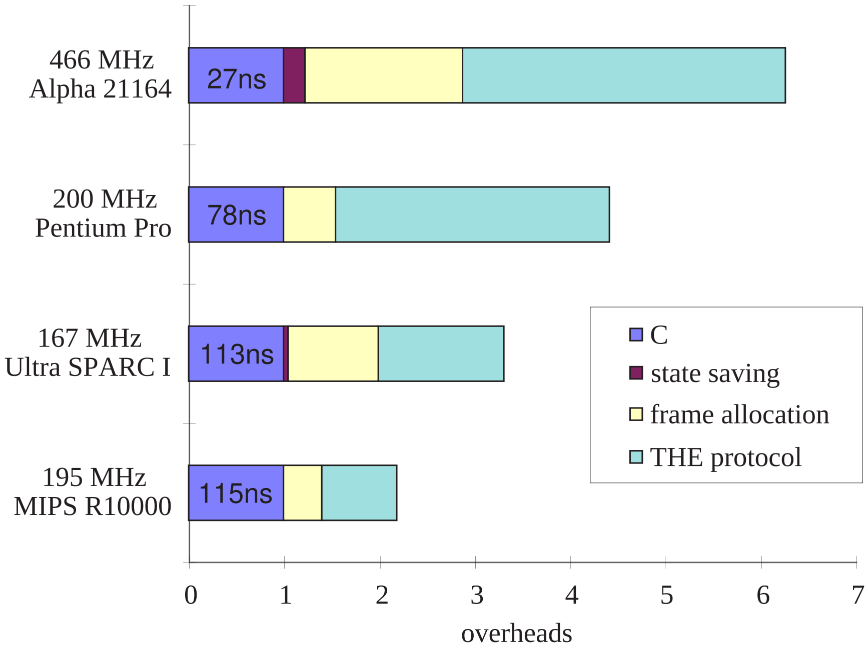


Figure 3-6: Breakdown of overheads for `fib` running on one processor on various architectures. The overheads are normalized to the running time of the serial C elision. The three overheads are for saving the state of a procedure before a spawn, the allocation of activation frames for procedures, and the THE protocol. Absolute times are given for the per-spawn running time of the C elision.

Our methodology for obtaining these numbers is as follows. First, we take the serial C `fib` program and time its execution. Then, we individually add in the code that generates each of the overheads and time the execution of the resulting program. We attribute the additional time required by the modified program to the scheduling code we added. In order to verify our numbers, we timed the `fib` code with all of the Cilk overheads added (the code shown in Figure 3-2), and compared the resulting time to the sum of the individual overheads. In all cases, the two times differed by less than 10%.

Overheads vary across architectures, but the overhead of Cilk is typically only a few times the C running time on this spawn-intensive program. Overheads on the Alpha machine are particularly large, because its native C function calls are fast compared to the other architectures. The state-saving costs are small for `fib`, because all four architectures have write buffers that can hide the latency of the writes required.

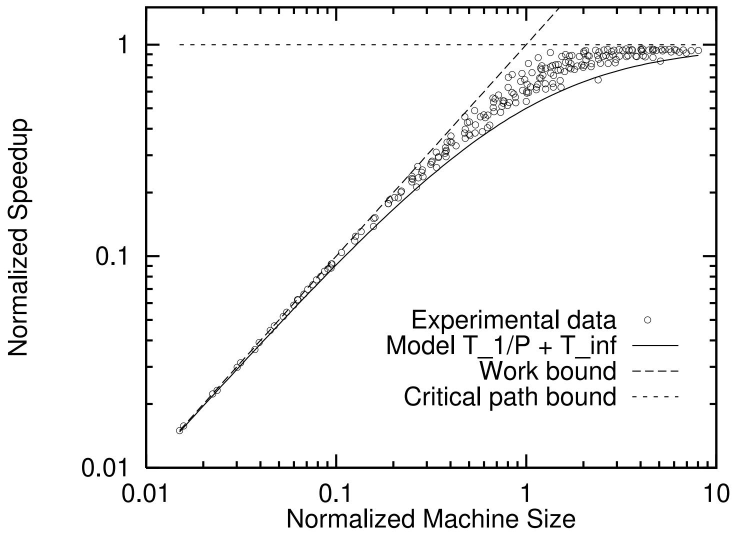


Figure 3-7: Normalized speedup curve for the knary benchmark in Cilk-5. The horizontal axis is the number P of processors and the vertical axis is the speedup T_1/T_P, but each data point has been normalized by dividing by T_1/T_∞. The graph also shows the speedup predicted by the formula $T_P = T_1/P + T_\infty$.

We also attempted to measure the critical-path overhead c_∞. We used the synthetic `knary` benchmark [14] to synthesize computations artificially with a wide range of work and critical-path lengths. Figure 3-7 shows the outcome from many such experiments. The figure plots the measured speedup T_1/T_P for each run against the machine size P for that run. In order to plot different computations on the same graph, we normalized the machine size and the speedup by dividing these values by the average parallelism $\overline{P} = T_1/T_\infty$, as was done in [14]. For each run, the horizontal position of the plotted datum is the inverse of the slackness $P/\overline{P}$, and thus, the normalized machine size is 1.0 when the number of processors is equal to the average parallelism. The vertical position of the plotted datum is $(T_1/T_P)/\overline{P} = T_\infty/T_P$, which measures the fraction of maximum obtainable speedup. As can be seen in the chart, for almost all runs of this benchmark, we observed $T_P \leq T_1/P + 1.0T_\infty$. (One exceptional data point satisfies $T_P \approx T_1/P + 1.05T_\infty$.) Thus, although the work-first principle caused us to move overheads to the critical path, the ability of Cilk applications to scale up was not significantly compromised.

3.5 Conclusion

We conclude this chapter by examining some related work.

Mohr *et al.* [76] introduced lazy task creation in their implementation of the Mul-T language. Lazy task creation is similar in many ways to our lazy scheduling techniques. Mohr *et al.* report a work overhead of around 2 when comparing with serial T, the Scheme dialect on which Mul-T is based. Our research confirms the intuition behind their methods and shows that work overheads of close to 1 are achievable.

The Cid language [82] is like Cilk in that it uses C as a base language and has a simple preprocessing compiler to convert parallel Cid constructs to C. Cid is designed to work in a distributed memory environment, and so it employs latency-hiding mechanisms which Cilk-5 could avoid. Both Cilk and Cid recognize the attractiveness of basing a parallel language on C so as to leverage C compiler technology for high-performance codes. Cilk is a faithful extension of C, however, supporting the simplifying notion of a C elision and allowing Cilk to exploit the C compiler technology more readily.

TAM [28] and Lazy Threads [46] also analyze many of the same overhead issues in a more general, "nonstrict" language setting, where the individual performances of a whole host of mechanisms are required for applications to obtain good overall performance. In contrast, Cilk's multithreaded language provides an execution model based on work and critical-path length that allows us to focus our implementation efforts by using the work-first principle. Using this principle as a guide, we have concentrated our optimizing effort on the common-case protocol code to develop an efficient and portable implementation of the Cilk language.

Chapter 4

Applications

In this chapter, we describe some parallel applications we have written in Cilk. In the context of Cilk, this chapter gives some anecdotal evidence that realistic applications are easy to program and perform well. In addition, some of these applications have independent interest apart from Cilk as they are implemented with some novel algorithms. The applications include some dense matrix algorithms including matrix multiplication and LU factorization, a sparse Cholesky factorization algorithm, a Barnes-Hut N-body simulator, and the world's fastest Rubik's cube solver.

4.1 Dense matrix algorithms

In this section, we describe the four dense matrix algorithms shown in Figure 3-5, `blockedmul`, `notempmul`, `strassen`, and `lu`. The first three algorithms are variants of matrix multiplication, and the last is a code for LU factorization. This section shows how the model of work and critical path can be applied to analyze algorithms written in Cilk and explore tradeoffs between work, average parallelism, and space utilization.

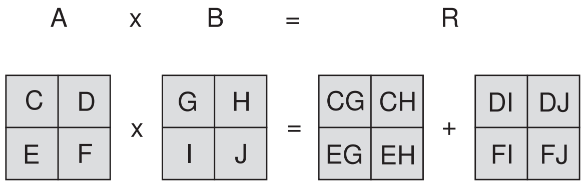


Figure 4-1: Recursive decomposition of matrix multiplication. The multiplication of $n \times n$ matrices requires eight multiplications of $n/2 \times n/2$ matrices, followed by one addition of $n \times n$ matrices.

4.1.1 Matrix multiplication

One way to program matrix multiplication is to use the recursive divide-and-conquer algorithm shown in Figure 4-1. To multiply one $n \times n$ matrix by another, we divide each matrix into four $n/2 \times n/2$ submatrices, recursively compute some products of these submatrices, and then add the results together. This algorithm lends itself to a parallel implementation, because each of the eight recursive multiplications is independent and can be executed in parallel.

Figure 4-2 shows Cilk code for a "blocked" implementation of recursive matrix multiplication in which the (square) input matrices `A` and `B` and the output matrix `R` are stored as a collection of 16×16 submatrices, called ***blocks***. The Cilk procedure `matrixmul` takes as arguments pointers to the first block in each matrix as well as a variable `nb` denoting the number of blocks in any row or column of the matrices. From the pointer to the first block of a matrix and the value of `nb`, the location of any other block in the matrix can be computed quickly. As `matrixmul` executes, values are stored into `R`, as well as into a temporary matrix `tmp`.

The procedure `blockedmul` operates as follows. Lines 3–4 check to see if the matrices to be multiplied consist of a single block, in which case a call is made to a serial routine `multiply_block` (not shown) to perform the multiplication. Otherwise, line 8 allocates some temporary storage in shared memory for the results, lines 9–10 compute pointers to the 8 submatrices of `A` and `B`, and lines 11–12 compute pointers to the 8 submatrices of `R` and the temporary matrix `tmp`. At this point, the divide step of the divide-and-conquer paradigm is complete, and we begin on the conquer step. Lines 13-20 recursively compute the 8 required submatrix multiplications in parallel,

```
1 cilk void blockedmul(long nb, block *A, block *B, block *R)
2   {
3     if (nb == 1)
4       multiply_block(A, B, R);
5     else {
6       block *C,*D,*E,*F,*G,*H,*I,*J;
7       block *CG,*CH,*EG,*EH,*DI,*DJ,*FI,*FJ;
8       block tmp[nb*nb];

        /* get pointers to input submatrices */
9       partition(nb, A, &C, &D, &E, &F);
10      partition(nb, B, &G, &H, &I, &J);

        /* get pointers to result submatrices */
11      partition(nb, R, &CG, &CH, &EG, &EH);
12      partition(nb, tmp, &DI, &DJ, &FI, &FJ);

        /* solve subproblems recursively */
13      spawn blockedmul(nb/2, C, G, CG);
14      spawn blockedmul(nb/2, C, H, CH);
15      spawn blockedmul(nb/2, E, H, EH);
16      spawn blockedmul(nb/2, E, G, EG);

17      spawn blockedmul(nb/2, D, I, DI);
18      spawn blockedmul(nb/2, D, J, DJ);
19      spawn blockedmul(nb/2, F, J, FJ);
20      spawn blockedmul(nb/2, F, I, FI);
21      sync;

        /* add results together into R */
22      spawn matrixadd(nb, tmp, R);
23      sync;
24    }
25    return;
26  }
```

Figure 4-2: Cilk code for recursive blocked matrix multiplication.

storing the results in the 8 disjoint submatrices of `R` and `tmp`. The `sync` statement in line 21 causes the procedure to suspend until all the procedures it spawned have finished. Then, line 22 spawns a parallel addition in which the matrix `tmp` is added into `R`. (The procedure `matrixadd` is itself implemented in a recursive, parallel, divide-and-conquer fashion, and the code is not shown.) The `sync` in line 23 ensures that the addition completes before `blockedmul` returns.

The work and critical-path length for `blockedmul` can 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, $T_1(n) = \Theta(n^3)$. 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 multiplica-

```
1 cilk void notempmul(long nb, block *A, block *B, block *R)
2   {
3     if (nb == 1)
4       multiplyadd_block(A, B, R);
5     else {
6       block *C,*D,*E,*F,*G,*H,*I,*J;
7       block *CGDI,*CHDJ,*EGFI,*EHFJ;

        /* get pointers to input submatrices */
8       partition(nb, A, &C, &D, &E, &F);
9       partition(nb, B, &G, &H, &I, &J);

        /* get pointers to result submatrices */
10      partition(nb, R, &CGDI, &CHDJ, &EGFI, &EHFJ);

        /* solve subproblems recursively */
11      spawn notempmul(nb/2, C, G, CGDI);
12      spawn notempmul(nb/2, C, H, CHDJ);
13      spawn notempmul(nb/2, E, H, EHFJ);
14      spawn notempmul(nb/2, E, G, EGFI);
15      sync;

16      spawn notempmul(nb/2, D, I, CGDI);
17      spawn notempmul(nb/2, D, J, CHDJ);
18      spawn notempmul(nb/2, F, J, EHFJ);
19      spawn notempmul(nb/2, F, I, EGFI);
20      sync;
21    }
22    return;
23  }
```

Figure 4-3: Cilk code for a no-temporary version of recursive blocked matrix multiplication.

tions 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 $T_\infty(n) = \Theta(\lg^2 n)$.

One drawback of the `blockedmul` algorithm from Figure 4-2 is that it requires temporary storage. We developed an in-place version of the same algorithm, called `notempmul`, which trades off a longer critical path for less storage. The code for `notempmul` is shown in Figure 4-3. The code is very similar to `blockedmul` except that there is an extra `sync` in line 15. By adding this extra synchronization, we are able to add in the second four recursive multiplications directly to the result matrix instead of storing them in a temporary matrix and adding them in later. The extra synchronization, however, makes the critical path longer. The critical-path length recurrence becomes $T_\infty(n) = 2T_\infty(n/2) + \Theta(\lg n)$, whose solution is $T_\infty(n) = \Theta(n)$.

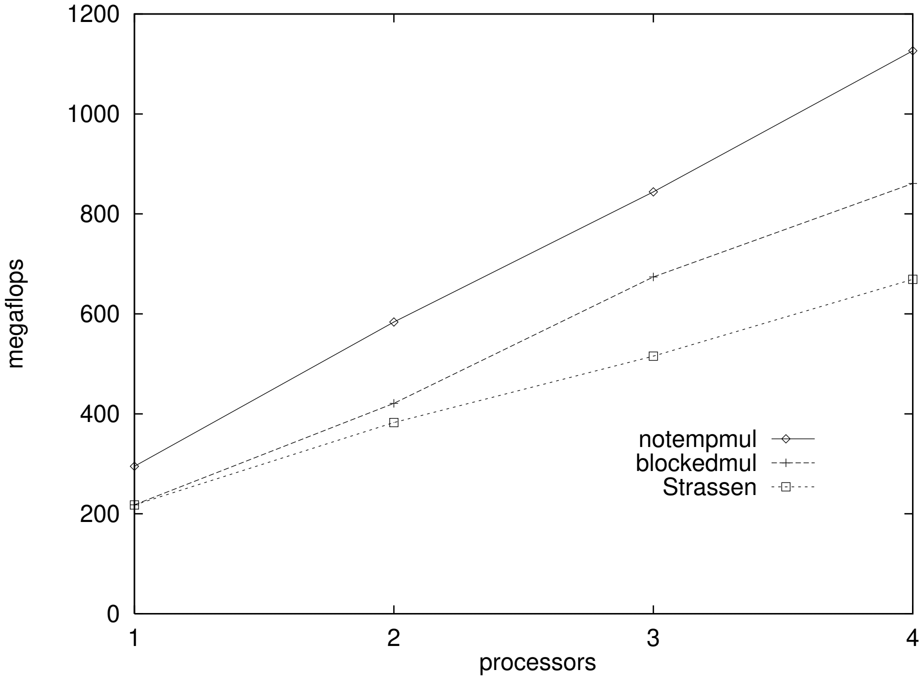


Figure 4-4: Total megaflops rate of Cilk matrix multiplication algorithms for 1024×1024 matrices (for the purpose of this graph, all algorithms are assumed to perform $2n^3$ flops, even though Strassen performs asymptotically less flops). These experiments were run on an Alpha 4100 SMP with 4 466MHz processors.

By lengthening the critical path, and hence reducing the average parallelism, we are able to use less storage. The average parallelism is still quite large, however, as can be seen from Figure 3-5, where a 1024×1024 `notempmul` multiply has a parallelism of 1970. Thus, the use of `notempmul` is generally a win except on very large machines.

We also experimented with Strassen's algorithm [96], a subcubic algorithm for matrix multiplication. The algorithm is significantly more complicated than that shown in Figure 4-1, but still required only an evening to code in Cilk. On one processor, Strassen is competitive with our other matrix multiplication codes for large matrices. Because its average parallelism is small, however, the Strassen code is less suitable for large parallel machines.

Figure 4-4 gives an absolute comparison of the three matrix multiplication algorithms presented.

4.1.2 LU decomposition

A divide-and-conquer algorithm for LU decomposition can be constructed in a similar recursive fashion to the matrix multiplication algorithms. LU decomposition is the process of factoring a matrix A into the product of a lower triangular matrix and an upper triangular matrix. To see how our algorithm works, divide the matrix A and its factors L and U into four parts so that $A = L \cdot U$ 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 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 $L_{11} \cdot U_{11}$.

To understand the performance of this LU-decomposition algorithm, we must first understand how the back- and forward-substitution algorithms work. To solve these problems on an $n \times n$ matrix, we can also use a parallel divide-and-conquer strategy. For back substitution (forward substitution is symmetric), we wish to solve the matrix equation $A = LX$ for the unknown matrix X, where L is a lower triangular matrix. Subdividing the three matrices as we did for LU-decomposition, we solve the equation as follows. First, solve $A_{00} = L_{00}X_{00}$ for X_{00} recursively, and in parallel solve $A_{01} = L_{00}X_{01}$ for X_{01}. Then, compute $A'_{10} = A_{10} - L_{10}X_{00}$ and $A'_{11} = A_{11} - L_{10}X_{01}$ using one of the matrix muliplication routines from Section 4.1.1. Finally, solve $A'_{10} = L_{11}X_{10}$ for X_{10} recursively, and in parallel solve $A'_{11} = L_{11}X_{11}$ for X_{11}.

To analyze back substitution, let us assume that we are implementing an in-place algorithm, so that we can use the multiplication algorithm `notempmul` that requires no auxiliary space, but which has a critical path of length $\Theta(n)$. The computational work for back substitution satisfies $T_1(n) = 4T_1(n/2) + \Theta(n^3)$, since matrix multiplication has computational work $\Theta(n^3)$, which has solution $T_1(n) = \Theta(n^3)$. The critical-path length for back substitution is $T_\infty(n) = 2T_\infty(n/2) + \Theta(n)$, since the first two recursive subproblems together have a critical path of $T_\infty(n/2)$, as do the second two subproblems, which must wait until the first two are done. The solution to this recurrence is $T_\infty(n) = \Theta(n \lg n)$. The results for forward substitution are identical.

We can now analyze the LU-decomposition algorithm. First, observe that if `notempmul` is used to form the Schur complement as well as in the back and forward substitution, the entire algorithm can be performed in place with no extra storage. For the computational work of the algorithm, we obtain the recurrence $T_1(n) = 2T_1(n/2) + \Theta(n^3)$, since we have two recursive calls to the algorithm and $\Theta(n^3)$ computational work is required for the back substitution, the forward substitution, and the matrix multiplication to compute the Schur complement. This recurrence gives us a solution of $T_1(n) = \Theta(n^3)$ for the computational work. The critical-path length has recurrence $T_\infty(n) = 2T_\infty(n/2) + \Theta(n \lg n)$, since the back and forward substitutions have $\Theta(n \lg n)$ critical-path length. The solution to this recurrence is $T_\infty(n) = \Theta(n \lg^2 n)$.

If we replace all of the matrix multiplication routines in the LU decomposition algorithm with calls to `blockedmul` algorithm, we can improve the critical path to $\Theta(n \lg n)$ at the expense of using extra temporary space. We are able to predict using our computational model, however, that the in-place algorithm performs better on our 8-processor SMP for large matrices. For instance, the average parallelism of the in-place algorithm is 370 for 2048×2048 matrices, easily satisfying the assumption of parallel slackness. Hence, the longer critical path length of the in-place algorithm will not have much effect on the overall running time, and allocation costs and cache effects from using more space dominate the performance tradeoff.

As a final note, we observe that our LU decomposition algorithm does not perform any pivoting, and therefore can be numerically unstable. We are investigating techniques to adapt this algorithm to perform some form of pivoting.

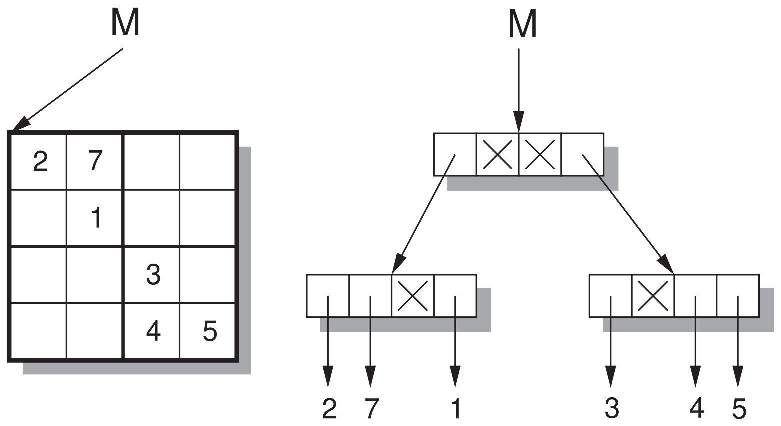


Figure 4-5: Example of a quadtree representation of matrix M. The standard dense representation is shown on the left, and the sparse quadtree representation is shown on the right. The dense matrix is represented by a pointer to the first element of the matrix. The sparse matrix is represented by a pointer to a tree-like structure representing the nonzero elements of M. Each level of the quadtree representation contains pointers to the four submatrices (top-left quadrant, top-right quadrant, bottom-left quadrant, bottom-right quadrant) of the matrix. In our program, the recursion does not go down to individual matrix elements as is shown in the figure, but instead switches to a dense representation for a 4×4 matrix.

4.2 Sparse Cholesky factorization

This section describes the implementation of a sparse Cholesky factorization algorithm developed by Charles Leiserson, Aske Plaat, and myself. We investigate the quadtree representation used to store the matrix and present the factorization algorithm that exploits the quadtree representation to obtain parallelism.

Traditionally, sparse matrices are stored in linked list form: each row (and possibly column) is represented as a linked list of the nonzero elements in that row. This representation is compact and efficient for a serial program, but presents some problems for a parallel program. In particular, it is difficult to expose high-level parallelism because it is difficult to represent efficiently submatrices of a matrix.

For our parallel sparse Cholesky factorization algorithm, we chose to change the matrix representation to allow the problem to be parallelized more efficiently. We represent a matrix as a ***quadtree*** [100], which is a recursive data structure shown in Figure 4-5. A matrix M is represented by pointers to the four submatrices that make up the quadrants of M. Sparsity is encoded by representing submatrices with all zero elements as null pointers. For efficiency reasons, the recursion of the data structure bottoms out at matrices of size 4×4, where we store the 16 entries of the 4×4 matrix in dense form.

Cholesky factorization is the process of computing, for a symmetric positive definite matrix M, a lower-triangular matrix L such that $M = LL^T$. Our parallel algorithm for sparse Cholesky factorization is a divide-and-conquer algorithm which follows the recursive structure of the quadtree. The algorithm for factoring a matrix $M = \begin{bmatrix} M_{00} & M_{10}^T \\ M_{10} & M_{11} \end{bmatrix}$ into a matrix $L = \begin{bmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{bmatrix}$ is similar to the LU decomposition algorithm and proceeds as follows:

1. Recursively Cholesky factor the upper-left quadrant of M, i.e. compute L_{00} such that $M_{00} = L_{00}L_{00}^T$.

2. Solve for L_{10} in the equation $M_{10} = L_{10}L_{00}^T$ using parallel back substitution.

3. Compute $M_{11} - L_{10}L_{10}^T$ using parallel matrix multiplication and subtraction.

4. Compute L_{11} as the Cholesky factorization of $M_{11} - L_{10}L_{10}^T$.

When the recursion reaches a 4×4 matrix at the leaves of the quadtree, we do a direct Cholesky factorization of the dense 4×4 matrix. The code for the parallel back substitution and matrix multiplication is similar to the dense algorithms for these problems given in Section 4.1. The only fundamental difference in the sparse algorithms is that opportunities for optimization can occur if one or more submatrix of the quadtree is all zero. Our algorithms for back substitution and matrix multiplication use these opportunities to take advantage of the sparsity. Unfortunately, these optimizations depend on the structure of the sparse matrix, and hence analyzing the work and critical path of sparse Cholesky is difficult. Fortunately, the Cilk runtime system can measure the work and critical path of a Cilk computation, so we can experimentally determine T_1 and T_∞ for the Cholesky algorithm run on specific input matrices. Figure 3-5 shows the results of an experiment for the matrix `bcsstk32` obtained from Matrix Market [17]. We can see that with an average parallelism of 420, the Cholesky algorithm run on this particular problem instance should scale well.

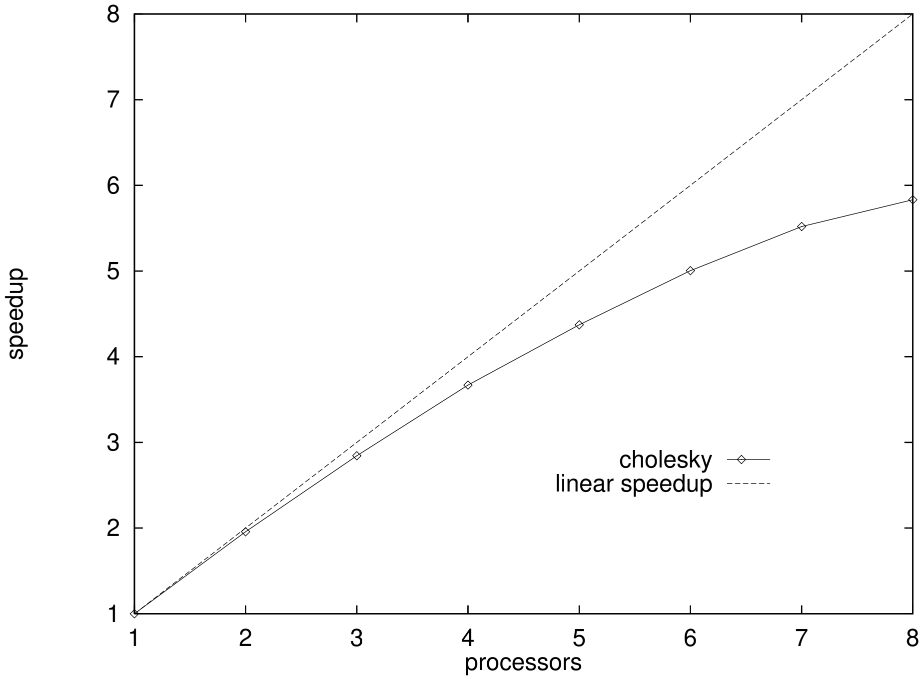


Figure 4-6: Speedup of our sparse Cholesky algorithm on the matrix `bcsstk29`.

Figure 4-6 shows, for a different problem instance, that sparse Cholesky does scale well. These experiments were run with the sparse matrix `bcsstk29` obtained from Matrix Market [17], reordered using MATLAB's column minimum degree permutation. The matrix has $13,992$ rows with $316,740$ nonzero elements. Despite the irregular nature of our parallel sparse matrix computation, the Cilk runtime system is able to schedule the computation efficiently on our 8 processor UltraSPARC SMP. Furthermore, the nature of the algorithm lends itself very easily to its expression in Cilk. The entire factorization program, including I/O, timing, and testing code, is expressed in less than 1000 lines of Cilk. The combination of simplicity and performance makes Cilk a good fit for this application.

We did have to reorganize the representation of the sparse matrix in order to obtain parallelism, however. The overhead of the quadtree organization adds about 20% overhead to the running time of the Cilk code on one processor. This work overhead is unfortunate, but we see no simple way of parallelizing Cholesky factorization using the more efficient linked-list representation. It is an open question whether linked-list parallelizations can achieve similar speedups to those shown in Figure 4-6.

Finally, we note that as with our LU decomposition algorithm, our factorization algorithm does no pivoting. We do not know how our algorithm might be adapted to do partial or complete pivoting. Thus, our algorithm is only recommended for matrices which are well-conditioned.

4.3 Barnes-Hut

Barnes-Hut is an algorithm for simulating the motion of particles under the influence of gravity. Barnes-Hut is part of the SPLASH-2 benchmark suite from Stanford [101], a set of standard parallel benchmark applications. We currently have a version of the Barnes-Hut algorithm coded in Cilk, derived from a C version of the algorithm obtained from Barnes's home page [5]. This C code was also the basis for the SPLASH-2 benchmark program.

The Barnes-Hut code is organized into four phases, a tree-build phase which builds an octtree describing the hierarchical decomposition of space, a force calculation phase which calculates gravitational forces on each particle, and two particle-push phases which move particles along their calculated trajectories. Our parallel Barnes-Hut code parallelizes all four phases. The most computation-intensive portion of the code, the force calculation phase, is easily parallelizable because there are no dependencies between particles in this phase. The two particle-push phases are also easily parallelizable. The only difficult phase to parallelize is the tree-building phase, because there are complicated dependencies between particles. The tree building phase is parallelized by taking advantage of the write-once nature of the tree. Although the parallel tree building may result in nondeterministic intermediate states, the final tree that results is deterministic. The critical path of the parallel algorithm is only $O(\lg n)$, assuming no lock conflicts and a distribution of particles that leads to a balanced tree.

A comparison of our Cilk Barnes-Hut code is made with the SPLASH-2 Barnes-Hut code in Section 1.2. That comparison shows that the effort required to parallelize the original C code using Cilk is significantly less than the effort required to parallelize

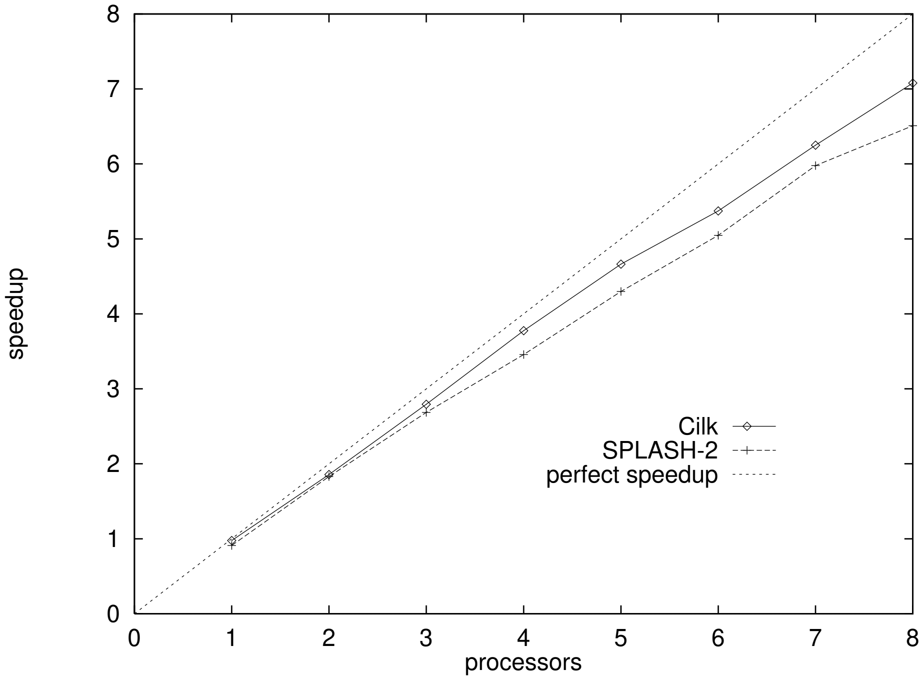


Figure 4-7: End-to-end speedup (T_S/T_P) of Cilk and SPLASH-2 versions of the Barnes-Hut algorithm. These experiments were run on a Sun Enterprise 5000 with 8 167MHz UltraSPARC processors.

the original C code using the SPLASH-2 thread library. For a performance comparison, Figure 1-2 compares the performance of the Cilk and SPLASH-2 parallelizations. The Cilk version of Barnes-Hut runs only 2.4% slower than the C version, showing that the overhead of spawn and sync is quite low, even for this relatively fine-grained application (each tree insertion, force calculation, and particle push is its own thread in the Cilk version). In contrast, the work overhead of the SPLASH-2 code is 9.9% due to the explicit load balancing and partitioning code that is required. In this section, we additionally show speedup curves for both parallelizations in Figure 4-7. This figure indicates that no scalability is lost when writing a program in Cilk versus writing a program in a thread library like SPLASH-2. Although both parallelizations achieve approximately the same speedup T_1/T_8, when we look at end-to-end speedup T_S/T_8 the Cilk parallelization does better because of its lower work overhead. From these numbers, we conclude that Cilk provides both a programming environment which enables parallel programming at a higher level of abstraction, and performance competitive with implementations that use no such abstractions.

4.4 Rubik's cube

Rubik's cube has fascinated mathematicians and laypeople alike since its introduction by Ernö Rubik [93, 85]. Despite the fact that a large amount of research has been done on Rubik's cube, some important questions remain unanswered. This section will focus on one particular unanswered question — what is the hardest position to solve? Although we will not answer this question definitively, we present some strong evidence that the hardest cube to solve is the one pictured in Figure 4-8, called "superflip" because it is the solved cube with each edge piece flipped in place.

In order to find the hardest position, we need to define what it means for a position to be hard to solve. We examine the problem of solving Rubik's cube using the ***quarter-turn metric***. In this metric, each quarter-turn of a face of the cube counts as one move. Thus, there are twelve possible moves, turning each of the six faces either clockwise or counterclockwise.[1] We denote the six clockwise moves as F,B,U,D,L,R for the front, back, up, down, left, and right faces of the cube, and the counterclockwise moves are denoted with a prime ('). With this definition of a move, a cube is hard to solve if the minimum number of moves to solve that position, called its ***depth***, is large. The depth of the superflip cube is known to be 24, and it has been conjectured that it may be the hardest Rubik's cube position. Although the depth of superflip is known, little else is known about other deep cubes. This work is the first to produce an approximate histogram of cube depths.

Determining the depth of a position is a difficult task. There are $43,252,003,274,489,856,000$ possible positions of Rubik's cube, which is too many to exhaustively search with today's computers. Don Dailey and I have developed a program, however, that can find the minimum depth of most cubes in at most a few hours. We describe how this program works and how we have used it to provide evidence that the superflip cube is in fact a very hard cube.

[1] Other metrics include the ***half-turn metric***, in which half turns of the faces count as one move, and the ***half-slice metric***, in which quarter and half turns of the center slices also count as one move.

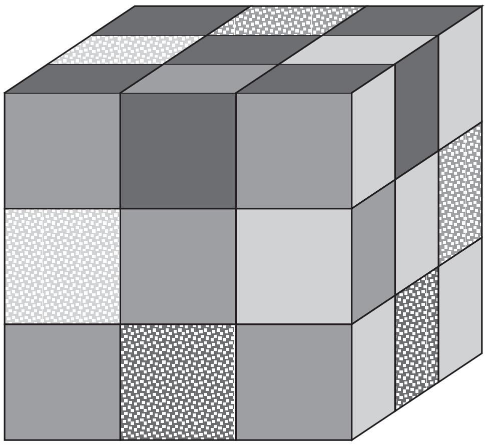


Figure 4-8: The superflip cube, conjectured to be the hardest Rubik's cube position to solve. It is formed by flipping in place each of the edge pieces of the solved cube.

4.4.1 Search

In this section, we describe the method we use to find minimal solution sequences (and hence depths) of cube positions. Our method is basically a brute-force search algorithm with very agressive pruning heuristics. Our program can solve an average cube in about 20 minutes, and most in a few hours, on a 4-processor 466MHz Alpha machine using about 1.2 gigabytes of memory.

Our search algorithm is simple. From a cube position, we try all possible move sequences of a certain length and see if any reach the solved position. We use a technique called iterative deepening in which we try successively longer move sequences until we find a solution. Thus, the first move sequence found gives a minimal solution sequence. Of course, some move sequences are redundant. For instance, the move sequence F F and the move sequence F' F' both generate the same cube. In order to reduce the branching factor of our search, we try to eliminate as many of these redundant move sequences as possible. We do this pruning using four rules:

1. Do not undo the last move.

2. Do not play three of the same move in a row.

3. Do not play two counterclockwise moves in a row.

4. Play "parallel" moves in increasing order. Parallel moves are moves of opposite

faces. For instance, the move sequences F B and B F both generate the same position. We do not play the lexicographically larger of these two sequences.

These rules reduce the branching factor from 12 to an asymptotic value of approximately 9.374. Why only these four rules? Interestingly, there are no more simple rules like the ones above. The above rules can all be enforced by disallowing certain triples of moves. If we disallow quadrouples or quintuples of moves, we get no further improvement in the branching factor. In other words, all sequences of 5 moves are distinct when filtered with the above four rules. Additionally, less than 1% of all 6 move sequences are redundant. Thus, these four rules provide all the branching factor pruning we can reasonably expect.

4.4.2 Pruning heuristics

Once we have removed most redundant sequences, we apply pruning heuristics that eliminate certain positions from possible consideration for a minimal move sequence. Our rules generate assertions of the form "this position cannot be solved in less than x moves." When searching a particular position to a particular depth, we apply all of our rules, and if one of them asserts that the position in question cannot be solved in the given number of moves, we prune the search. We have considered many different pruning rules, but only two rules are currently used by our solver.

Our first rule is that a position cannot be solved in less moves than the corners of the cube can be solved. The "corners cube", consisting of Rubik's cube minus the edge pieces, can be solved by brute force because it has only $88,179,840$ possible positions. We use a table of the depths of these positions, stored at 4 bits per entry ($\approx$ 42MB), to determine the number of moves required to solve the corner cube.[2]

Our second rule uses a hash table to determine a bound on the depth of a cube. Each hash table entry contains a lower bound on the minimum depth of all cubes that hash to that entry. To fill the table, we use an exhaustive enumeration of all cubes

[2]We would like to use an edge cube rule as well, but the edge cube has $490,497,638,400$ positions, slightly too large for the memory of our machines.

near the solved cube and compute the minimum depth cube that maps to each table entry. When searching, we hash our unknown cube and determine the minimum possible depth of that cube using the hash table. If we are looking for a shorter solution than allowed by the hash table information, we can prune the search. We currently use a 1 GB hash table, stored with 2 bits per entry. An entry can encode one of four states: no cube ≤ 12 in depth, no cube ≤ 11 in depth, no cube ≤ 10 in depth, and no lower bound.

4.4.3 Symmetry

We also take advantage of the symmetries of the cube. A ***symmetry*** of a cube position is obtained by recoloring the facets of the cube in a way that preserves oppositeness of colors. There are $6 \times 4 \times 2 = 48$ possible recolorings, and hence symmetries. Alternatively, the 48 symmetries can be viewed as the rigid transformations of space that take the cube to itself. For instance, there is a symmetry obtained by rotating the cube 90 degrees around a face, 120 degrees around a corner, etc. We can apply these symmetries to any cube position to obtain another "symmetric" position. The set of cube positions obtained by applying the 48 symmetry operators to a particular position gives the ***equivalence class*** of that position. These equivalence classes partition the set of cube positions.

An important property of positions in the same equivalence class is that they have the same depth. To prove this fact, consider a possible solution sequence $M = m_1, m_2, \ldots, m_{k-1}, m_k$ of a position P. Then, viewing the moves in the solution sequence as operators on the cube state, we have $SOLVED = m_k m_{k-1} \cdots m_2 m_1 P$. Multiplying on the left by a symmetry s, and adding some identity symmetries $s^{-1}s$ following each move, we get the following equation:

$$sSOLVED = (sm_k s^{-1})(sm_{k-1} s^{-1}) \cdots (sm_2 s^{-1})(sm_1 s^{-1}) sP \ .$$

Because any symmetry applied to $SOLVED$ gives the same position, $sSOLVED$ is equal to $SOLVED$. Thus, if M is a solution sequence for P, then

$$M' = sm_1s^{-1}, sm_2s^{-1}, \ldots, sm_{k-1}s^{-1}, sm_ks^{-1}$$

is a solution sequence for sP.[3] Therefore, two symmetric positions have solution sequences of the same length and therefore those positions have the same depth.

We take advantage of the fact that all positions in the same equivalence class have the same depth by storing only a canonical representative of each equivalence class in the hash table. Because most equivalence classes are of size 48, we save almost a factor of 48 in hash table space.

Calculating the canonical representative of an equivalence class can be expensive. In the worst case, we have to enumerate all positions in the equivalence class by applying all 48 symmetry operators to a position in order to determine which position is the canonical representative. We use a trick, however, which lets us compute the canonical representative quickly in most cases. The trick uses a crafty definition of the canonical representative. For an equivalence class of cubes, the canonical representative is the one with the lexicographically smallest representation, where the representation is ordered so that the state of the corner cubies is in the high order bits and the state of the edge cubies is in the low order bits. Thus, the canonical representative of a position's equivalence class is completely determined by the state of the corners of that position, as long as no two members of the equivalence class have the same corner state. It turns out that less than 0.26% of positions belong to an equivalence class that has two or more positions with the same corner state.

In order to calculate canonical representatives quickly, then, we again use the corner cube and an auxiliary 84 MB table to figure out which symmetry generates the canonical representative for a particular position. For each corner state, we store the symmetry operator that, when applied to the given position, generates the canonical representation. Therefore, we only need to apply one symmetry operator, instead of 48, to determine the canonical representative of a position's equivalence class. In

[3]It is easy to verify that the conjugates sm_is^{-1} are operators representing legal moves.

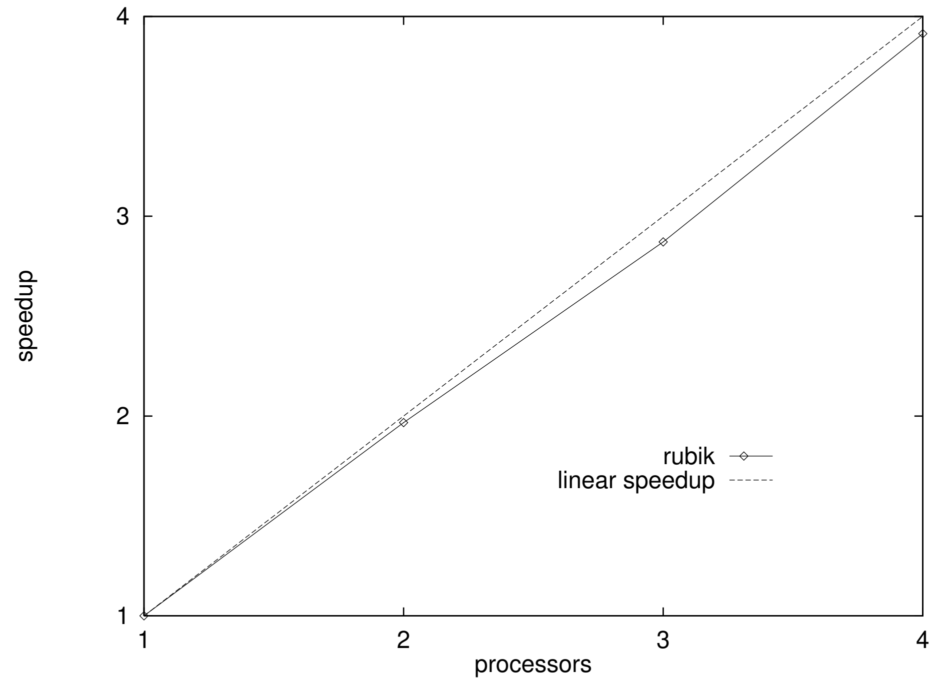


Figure 4-9: Speedup (T_1/T_P) for our Rubik's cube solver. The work overhead c_1 for the solver is less than 1.12.

the rare case when this operator is not completely determined by the corner state, a dummy value is stored in the table, and if this dummy value is detected, all 48 symmetry operators are applied to the position and the minimum lexicographic position is found. Since only 0.26% of positions require this additional work, however, we usually only need to compute one symmetry.

4.4.4 Experiments

Using our search and heuristic techniques, we can solve a random cube in about 20 minutes on a four-processor 466MHz Alpha machine using about 1.2 GB of memory. We use Cilk to exploit all four processors of the machine. This application is very easy to program using Cilk because of its recursive tree structure. The work overhead of Cilk is only 12% ($c_1 = 1.12$) for this application. This overhead is somewhat higher than most Cilk applications because the average thread length, $2.9\mu s$, is very short. This overhead, however, is well worth the dynamic load balancing and ease of expression provided by Cilk. Rubik's cube requires dynamic load balancing because

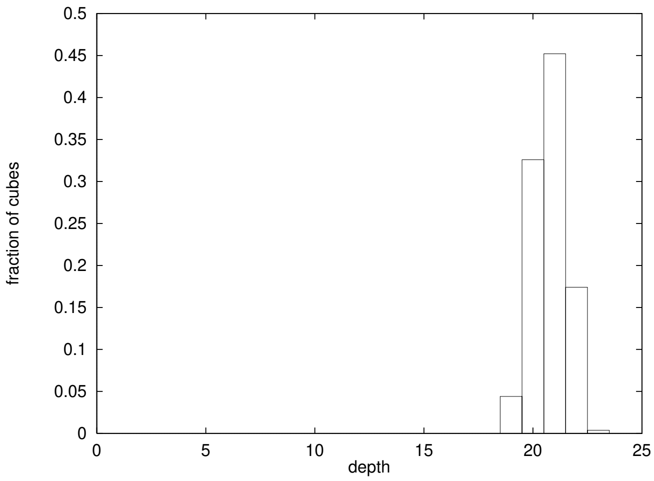


Figure 4-10: Sampled histogram of depths of random cubes. This histogram represents data from over 400 random cubes.

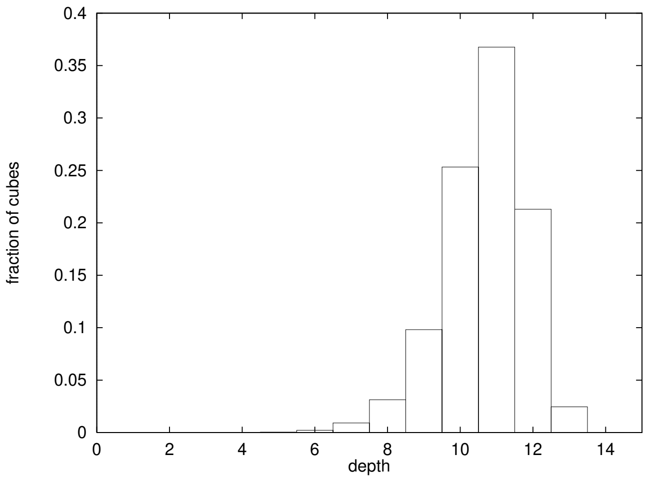


Figure 4-11: Exact histogram of depths of the $2 \times 2 \times 2$ "corner" cube. The maximum depth of a corner cube is 14, and only a 0.000075 fraction of corner cubes have this depth.

the depth of search subtrees can vary widely depending on when the heuristic cutoffs kick in. Thus, Cilk provides a natural language in which to express this parallel program. For searches with deterministic amounts of work (those which find no solution and hence have to search the entire tree), we get near-perfect linear speedup for this application (see Figure 4-9).

We now return to the central question. What is the hardest position to solve? We have used our program to generate Figure 4-10, a randomly sampled histogram of cube depths. After solving over 400 random cubes, we have found only a few cubes with depth 23, and no cubes of depth 24, showing statistically that few cubes are depth 23 or greater. In fact, this research has found the first depth 23 cubes which are not near superflip. The only previously known depth 23 equivalence classes are the one adjacent to superflip, superflip + F U F, and superflip + F F B. If we compare this sampled histogram with the exact histogram for the $2 \times 2 \times 2$ cube in Figure 4-11, we see that they are quite similar. Since the maximum-depth corner cube, which is of depth 14, is only 3 greater than the depth of the corner cubes at the peak of the distribution, we can expect the maximum-depth full cube to be only a few depths away from the peak of its distribution as well. Since the depth of the superflip cube is known to be 24, 3 greater than the peak of the distribution, we conjecture that superflip is likely to be the hardest position of Rubik's cube.

Chapter 5

The Nondeterminator-2

Recall from Chapter 1 that a ***data race*** occurs when two parallel threads holding no locks in common access the same memory location and at least one of the threads modifies the location. This chapter describes the algorithms and strategies used by the Nondeterminator-2 debugging tool to find data races in Cilk programs.[1] Like its predecessor, the Nondeterminator (which checks for simple "determinacy" races), the Nondeterminator-2 is a debugging tool, not a verifier, since it checks for data races only in the computation generated by a serial execution of the program on a given input.

We give an algorithm, All-Sets, which determines whether the computation generated by a serial execution of a Cilk program on a given input contains a race. For a program that runs serially in time T, accesses V shared memory locations, uses a total of n locks, and holds at most $k \ll n$ locks simultaneously, All-Sets runs in $O(n^k T\,\alpha(V,V))$ time and $O(n^k V)$ space, where α is Tarjan's functional inverse of Ackermann's function.

Since All-Sets may be too inefficient in the worst case, we propose a much more efficient algorithm which can be used to detect races in programs that obey the "umbrella" locking discipline, a programming methodology that is more flexible than similar disciplines proposed in the literature. We present an algorithm, Brelly,

[1] The contents of this chapter are joint work with Guang-Ien Cheng, Mingdong Feng, Charles Leiserson, and Andrew Stark and will appear at SPAA'98 [23].

which detects violations of the umbrella discipline in $O(kT\,\alpha(V,V))$ time using $O(kV)$ space.

We also prove that any "abelian" Cilk program, one whose critical sections commute, produces a determinate final state if it is deadlock free and if it generates any computation which is data-race free. Thus, the Nondeterminator-2's two algorithms can verify the determinacy of a deadlock-free abelian program running on a given input.

5.1 Data races

In a parallel multithreaded computation, a ***data race*** exists if logically parallel threads access the same location, the two threads hold no locks in common, and at least one of the threads writes to the location. A data race is usually a bug, because depending on how the threads are scheduled, the program may exhibit unexpected, nondeterministic behavior. If the two threads hold a lock in common, however, the nondeterminism is usually not a bug. By introducing locks, the programmer presumably intends to allow the locked critical sections to be scheduled in either order, as long as they are not interleaved.

Figure 5-1 illustrates a data race in a Cilk program. The procedures `foo1`, `foo2`, and `foo3` run in parallel, resulting in parallel accesses to the shared variable `x`. The accesses by `foo1` and `foo2` are protected by lock `A` and hence do not form a data race. Likewise, the accesses by `foo1` and `foo3` are protected by lock `B`. The accesses by `foo2` and `foo3` are not protected by a common lock, however, and therefore form a data race. If all accesses had been protected by the same lock, only the value 3 would be printed, no matter how the computation is scheduled. Because of the data race, however, the value of `x` printed by `main` might be 2, 3, or 6, depending on scheduling, since the statements in `foo2` and `foo3` are composed of multiple machine instructions which may interleave, possibly resulting in a lost update to `x`.

Since a data race is usually a bug, automatic data-race detection has been studied extensively. Static race detectors [78] can sometimes determine whether a program

```
int x;                          cilk void foo3() {
Cilk_lockvar A, B;                Cilk_lock(&B);
                                  x++;
cilk void foo1() {                Cilk_unlock(&B);
  Cilk_lock(&A);                }
  Cilk_lock(&B);
  x += 5;                       cilk int main() {
  Cilk_unlock(&B);                Cilk_lock_init(&A);
  Cilk_unlock(&A);                Cilk_lock_init(&B);
}                                 x = 0;
                                  spawn foo1();
cilk void foo2() {                spawn foo2();
  Cilk_lock(&A);                  spawn foo3();
  x -= 3;                         sync;
  Cilk_unlock(&A);                printf("%d", x);
}                               }
```

Figure 5-1: A Cilk program with a data race. The data race is between the accesses to `x` in `foo2` and `foo3`.

will ever produce a data race when run on all possible inputs. Since static debuggers cannot fully understand the semantics of programs, however, most race detectors are dynamic tools in which potential races are detected at runtime by executing the program on a given input. Some dynamic race detectors perform a post-mortem analysis based on program execution traces [34, 53, 72, 79], while others perform an "on-the-fly" analysis during program execution. On-the-fly debuggers directly instrument memory accesses via the compiler [30, 31, 37, 38, 71, 83], by binary rewriting [89], or by augmenting the machine's cache coherence protocol [75, 84].

The race-detection algorithms in this chapter are based on the Nondeterminator [37], which finds "determinacy races" in Cilk programs that do not use locks. The Nondeterminator executes a Cilk program serially on a given input, maintaining an efficient "SP-bags" data structure to keep track of the logical series/parallel relationships between threads. For a Cilk program that runs serially in time T and accesses V shared-memory locations, the Nondeterminator runs in $O(T\,\alpha(V,V))$ time and $O(V)$ space, where α is Tarjan's functional inverse of Ackermann's function, which for all practical purposes is at most 4.

The Nondeterminator-2, which is currently under development, finds data races in Cilk programs that use locks. This race detector contains two algorithms, both of which use the same efficient SP-bags data structure from the original Nondeterminator. The first of these algorithms, ALL-SETS, is an on-the-fly algorithm which, like most other race-detection algorithms, assumes that no locks are held across parallel control statements, such as `spawn` and `sync`, and thus all critical sections are sequential code. The second algorithm, BRELLY, is a faster on-the-fly algorithm, but in addition to reporting data races as bugs, it also reports as bugs some complex (but race-free) locking protocols.

The ALL-SETS algorithm executes a Cilk program serially on a given input and either detects a data race in the computation or guarantees that none exist. For a Cilk program that runs serially in time T, accesses V shared-memory locations, uses a total of n locks, and holds at most $k \ll n$ locks simultaneously, ALL-SETS runs in $O(n^k T\,\alpha(V,V))$ time and $O(n^k V)$ space. Tighter, more complicated bounds on ALL-SETS will be given in Section 5.2.

In previous work, Dinning and Schonberg's "lock-covers" algorithm [31] also detects all data races in a computation. The ALL-SETS algorithm improves the lock-covers algorithm by generalizing the data structures and techniques from the original Nondeterminator to produce better time and space bounds. Perkovic and Keleher [84] offer an on-the-fly race-detection algorithm that "piggybacks" on a cache-coherence protocol for lazy release consistency. Their approach is fast (about twice the serial work, and the tool runs in parallel), but it only catches races that actually occur during a parallel execution, not those that are logically present in the computation.

Although the asymptotic performance bounds of ALL-SETS are the best to date, they are a factor of n^k larger in the worst case than those for the original Nondeterminator. The BRELLY algorithm is asymptotically faster than ALL-SETS, and its performance bounds are only a factor of k larger than those for the original Nondeterminator. For a Cilk program that runs serially in time T, accesses V shared-memory locations, and holds at most k locks simultaneously, the serial BRELLY algorithm runs in $O(kT\,\alpha(V,V))$ time and $O(kV)$ space. Since most programs do not hold many locks simultaneously, this algorithm runs in nearly linear time and space. The improved performance bounds come at a cost, however. Rather than detecting data races directly, Brelly only detects violations of a "locking discipline" that precludes data races.

A ***locking discipline*** is a programming methodology that dictates a restriction on the use of locks. For example, many programs adopt the discipline of acquiring locks in a fixed order so as to avoid deadlock [57]. Similarly, the "umbrella" locking discipline precludes data races. It requires that each location be protected by the same lock within every parallel subcomputation of the computation. Threads that are in series may use different locks for the same location (or possibly even none, if no parallel accesses occur), but if two threads in series are both in parallel with a third and all access the same location, then all three threads must agree on a single lock for that location. If a program obeys the umbrella discipline, a data race cannot occur, because parallel accesses are always protected by the same lock. The Brelly algorithm detects violations of the umbrella locking discipline.

Savage et al. [89] originally suggested that efficient debugging tools can be developed by requiring programs to obey a locking discipline. Their Eraser tool enforces a simple discipline in which any shared variable is protected by a single lock throughout the course of the program execution. Whenever a thread accesses a shared variable, it must acquire the designated lock. This discipline precludes data races from occurring, and Eraser finds violations of the discipline in $O(kT)$ time and $O(kV)$ space. (These bounds are for the serial work; Eraser actually runs in parallel.) Eraser only works in a parallel environment containing several linear threads, however, with no nested parallelism or thread joining as is permitted in Cilk. In addition, since Eraser does not understand the series/parallel relationship of threads, it does not fully understand at what times a variable is actually shared. Specifically, it heuristically guesses when the "initialization phase" of a variable ends and the "sharing phase" begins, and thus it may miss some data races.

In comparison, our Brelly algorithm performs nearly as efficiently, is guaranteed to find all violations, and importantly, supports a more flexible discipline. In particular, the umbrella discipline allows separate program modules to be composed in series

```
int x;
Cilk_lockvar A, B, C;

cilk void bar1() {
  Cilk_lock(&A);
  Cilk_lock(&B);
  x += 1;
  Cilk_unlock(&B);
  Cilk_unlock(&A);
}

cilk void bar2() {
  Cilk_lock(&A);
  Cilk_lock(&C);
  x += 2;
  Cilk_unlock(&C);
  Cilk_unlock(&A);
}

cilk void bar3() {
  Cilk_lock(&B);
  Cilk_lock(&C);
  x += 3;
  Cilk_unlock(&C);
  Cilk_unlock(&B);
}

cilk int main() {
  Cilk_lock_init(&A);
  Cilk_lock_init(&B);
  Cilk_lock_init(&C);
  x = 0;
  spawn bar1();
  spawn bar2();
  spawn bar3();
  sync;
}
```

Figure 5-2: A Cilk program with no data race which violates the umbrella methodology. Accesses to the variable `x` are each guarded by two of the three locks `A`, `B`, and `C`, and thus do not race with each other. The three parallel accesses to `x` do not agree on a single lock to protect `x`, however, so this program violates the umbrella methodology.

without agreement on a global lock for each location. For example, an application may have three phases—an initialization phase, a work phase, and a clean-up phase—which can be developed independently without agreeing globally on the locks used to protect locations. If a fourth module runs in parallel with all of these phases and accesses the same memory locations, however, the umbrella discipline does require that all phases agree on the lock for each shared location. Thus, although the umbrella discipline is more flexible than Eraser's discipline, it is more restrictive than what a general data-race detection algorithm, such as ALL-SETS, permits. For example, the data-race free program in Figure 5-2 can be verified by ALL-SETS to be data-race free, but BRELLY will detect an umbrella discipline violation.

Most dynamic race detectors, like ALL-SETS and BRELLY, attempt to find, in the terminology of Netzer and Miller [81], ***apparent*** data races—those that appear to occur in a computation according to the parallel control constructs—rather than ***feasible*** data races—those that can actually occur during program execution. The distinction arises, because operations in critical sections may affect program control depending on the way threads are scheduled. Thus, an apparent data race between two threads in a given computation may not actually be feasible, because the computation itself may change if the threads were scheduled in a different order. Since the problem of exactly finding feasible data races is computationally difficult,[2] attention has naturally focused on the easier (but still difficult) problem of finding apparent data races.

For some classes of programs, however, a feasible data race on a given input exists if and only if an apparent data race exists in every computation for that input. To check for a feasible data race in such a program, it suffices to check a single computation for an apparent data race. One class of programs having this property are "abelian" programs in which critical sections protected by the same lock "commute": intuitively, they produce the same effect regardless of scheduling. For a computation generated by a deadlock-free abelian program running on a given input, we prove that if no data races exist in that computation, then the program is ***determinate***: all schedulings produce the same final result. For abelian programs, therefore, ALL-SETS and BRELLY can verify the determinacy of the program on a given input. Our results on abelian programs formalize and generalize the claims of Dinning and Schonberg [31, 32], who argue that for "internally deterministic" programs, checking a single computation suffices to detect all races in the program.

The remainder of this chapter is organized as follows. Section 5.2 presents the ALL-SETS algorithm, and Section 5.3 presents the BRELLY algorithm. Section 5.4 gives some empirical results obtained by using the Nondeterminator-2 in its ALL-SETS and BRELLY modes. Section 5.5 defines the notion of abelian programs and proves that data-race free abelian programs produce determinate results. Section 5.6 offers some concluding remarks.

[2]Even in simple models, finding feasible data races is NP-hard [80].

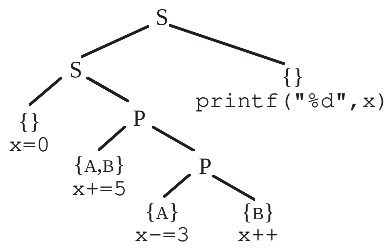


Figure 5-3: The series-parallel parse tree for the Cilk program in Figure 5-1, abbreviated to show only the accesses to shared location `x`. Each leaf is labeled with a code fragment that accesses `x`, with the lock set for that access shown above the code fragment.

5.2 The All-Sets algorithm

In this section, we present the ALL-SETS algorithm, which detects data races in Cilk computations that use locks. We first give some background on Cilk and the series-parallel control structure of its computations. We then discuss locking in Cilk. Finally, we present the ALL-SETS algorithm itself, show that it is correct, and analyze its performance.

As described in Section 2.8, the computation of a Cilk program on a given input can be viewed as a directed acyclic graph (dag) in which vertices are instructions and edges denote ordering constraints imposed by control statements. The computation dag generated by a Cilk program can itself be represented as a binary ***series-parallel parse tree***, as illustrated in Figure 5-3 for the program in Figure 5-1. In the parse tree of a Cilk computation, leaf nodes represent threads. Each internal node is either an ***S-node*** if the computation represented by its left subtree logically precedes the computation represented by its right subtree, or a ***P-node*** if its two subtrees' computations are logically in parallel. (We use the term "logically" to mean with respect to the series-parallel control, not with respect to any additional synchronization through shared variables.)

A parse tree allows the series/parallel relation between two threads e_1 and e_2 to be determined by examining their least common ancestor, which we denote by $\text{LCA}(e_1, e_2)$. If $\text{LCA}(e_1, e_2)$ is a P-node, the two threads are logically in parallel, which we denote by $e_1 \parallel e_2$. If $\text{LCA}(e_1, e_2)$ is an S-node, the two threads are logically in series, which we denote by $e_1 \prec e_2$, assuming that e_1 precedes e_2 in a left-to-right depth-first treewalk of the parse tree. The series relation $\prec$ is transitive.

Cilk provides the user with mutual-exclusion locks as described in Section 2.5. We assume in this chapter, as does the general literature, that any lock/unlock pair is contained in a single thread, and thus holding a lock across a parallel control construct is forbidden.[3] The ***lock set*** of an access is the set of locks held by the thread when the access occurs. The ***lock set*** of several accesses is the intersection of their respective lock sets.

If the lock set of two parallel accesses to the same location is empty, and at least one of the accesses is a WRITE, then a data race exists. To simplify the description and analysis of the race detection algorithm, we shall use a small trick to avoid the extra condition for a race that "at least one of the accesses is a WRITE." The idea is to introduce a ***fake lock*** for read accesses called the R-LOCK, which is implicitly acquired immediately before a READ and released immediately afterwards. The fake lock behaves from the race detector's point of view just like a normal lock, but during an actual computation, it is never actually acquired and released (since it does not actually exist). The use of R-LOCK simplifies the description and analysis of ALL-SETS, because it allows us to state the condition for a data race more succinctly: *if the lock set of two parallel accesses to the same location is empty, then a data race exists.* By this condition, a data race (correctly) does not exist for two read accesses, since their lock set contains the R-LOCK.

The ALL-SETS algorithm is based on the efficient SP-BAGS algorithm used by the original Nondeterminator to detect determinacy races in Cilk programs that do not use locks. The SP-BAGS algorithm executes a Cilk program on a given input in serial, depth-first order. This execution order mirrors that of normal C programs: every subcomputation that is spawned executes completely before the procedure that spawned it continues. While executing the program, SP-BAGS maintains an SP-bags data structure based on Tarjan's nearly linear-time least-common-ancestors algorithm [98].

[3]The Nondeterminator-2 can still be used with programs for which this assumption does not hold, but the race detector prints a warning, and some races may be missed. We are developing extensions of the Nondeterminator-2's detection algorithms that work properly for programs that hold locks across parallel control constructs.

The SP-bags data structure allows SP-BAGS to determine the series/parallel relation between the currently executing thread and any previously executed thread in $O(\alpha(V,V))$ amortized time, where V is the size of shared memory. In addition, SP-BAGS maintains a "shadow space" where information about previous accesses to each location is kept. This information is used to determine previous threads that have accessed the same location as the current thread. For a Cilk program that runs in T time serially and references V shared memory locations, the SP-BAGS algorithm runs in $O(T\,\alpha(V,V))$ time and uses $O(V)$ space.

The ALL-SETS algorithm also uses the SP-bags data structure to determine the series/parallel relationship between threads. Its shadow space *lockers* is more complex than the shadow space of SP-BAGS, however, because it keeps track of which locks were held by previous accesses to the various locations. The entry *lockers*$[l]$ stores a list of ***lockers***: threads that access location l, each paired with the lock set that was held during the access. If $\langle e, H\rangle \in$ *lockers*$[l]$, then location l is accessed by thread e while it holds the lock set H.

As an example of what the shadow space *lockers* may contain, consider a thread e that performs the following:

```
Cilk_lock(&A); Cilk_lock(&B);
READ(l)
Cilk_unlock(&B); Cilk_unlock(&A);
Cilk_lock(&B); Cilk_lock(&C);
WRITE(l)
Cilk_unlock(&C); Cilk_unlock(&B);
```

For this example, the list *lockers*$[l]$ contains two lockers—$\langle e, \{\mathtt{A}, \mathtt{B}, \text{R-LOCK}\}\rangle$ and $\langle e, \{\mathtt{B}, \mathtt{C}\}\rangle$.

The ALL-SETS algorithm is shown in Figure 5-4. Intuitively, this algorithm records all lockers, but it is careful to prune redundant lockers, keeping at most one locker per distinct lock set. Lines 1–3 check to see if a data race has occurred and report any violations. Lines 5–11 then add the current locker to the *lockers* shadow space and prune redundant lockers. A locker $\langle e, H\rangle$ is redundant if there exists a

```
ACCESS(l) in thread e with lock set H
 1  for each ⟨e', H'⟩ ∈ lockers[l]
 2       do if e' ∥ e and H' ∩ H = ∅
 3             then declare a data race
 4  redundant ← FALSE
 5  for each ⟨e', H'⟩ ∈ lockers[l]
 6       do if e' ≺ e and H' ⊇ H
 7             then lockers[l] ← lockers[l] − {⟨e', H'⟩}
 8          if e' ∥ e and H' ⊆ H
 9             then redundant ← TRUE
10  if redundant = FALSE
11     then lockers[l] ← lockers[l] ∪ {⟨e, H⟩}
```

Figure 5-4: The ALL-SETS algorithm. The operations for the `spawn`, `sync`, and `return` actions are unchanged from the SP-BAGS algorithm on which ALL-SETS is based. Additionally, the `Cilk_lock()` and `Cilk_unlock()` functions must be instrumented to add and remove locks from the lock set H appropriately.

stronger locker in *lockers*$[l]$, one which races with a future access whenever $\langle e, H\rangle$ races with that future access. We remove the redundant locker $\langle e', H'\rangle$ in line 7 because the locker $\langle e, H\rangle$ is stronger than $\langle e', H'\rangle$. Similarly, we do not add the locker $\langle e, H\rangle$ to *lockers*$[l]$ if we record in line 9 that another stronger locker $\langle e', H'\rangle$ is already in *lockers*$[l]$.

Before proving the correctness of ALL-SETS, we restate two important lemmas from [37].

Lemma 1 *Suppose that three threads e_1, e_2, and e_3 execute in order in a serial, depth-first execution of a Cilk program, and suppose that $e_1 \prec e_2$ and $e_1 \parallel e_3$. Then, we have $e_2 \parallel e_3$.* ■

Lemma 2 (Pseudotransitivity of $\parallel$) *Suppose that three threads e_1, e_2, and e_3 execute in order in a serial, depth-first execution of a Cilk program, and suppose that $e_1 \parallel e_2$ and $e_2 \parallel e_3$. Then, we have $e_1 \parallel e_3$.* ■

We now prove that the ALL-SETS algorithm is correct.

Theorem 3 *The* ALL-SETS *algorithm detects a data race in a computation of a Cilk program running on a given input if and only if a data race exists in the computation.*

Proof: ($\Rightarrow$) To prove that any race reported by the ALL-SETS algorithm really exists in the computation, observe that every locker added to $lockers[l]$ in line 11 consists of a thread and the lock set held by that thread when it accesses l. The algorithm declares a race when it detects in line 2 that the lock set of two parallel accesses (by the current thread e and one from $lockers[l]$) is empty, which is exactly the condition required for a data race.

($\Leftarrow$) Assuming a data race exists in a computation, we shall show that a data race is reported. If a data race exists, then we can choose two threads e_1 and e_2 such that e_1 is the last thread before e_2 in the serial execution which has a data race with e_2. If we let H_1 and H_2 be the lock sets held by e_1 and e_2, respectively, then we have $e_1 \parallel e_2$ and $H_1 \cap H_2 = \emptyset$.

We first show that immediately after e_1 executes, $lockers[l]$ contains some thread e_3 that races with e_2. If $\langle e_1, H_1 \rangle$ is added to $lockers[l]$ in line 11, then e_1 is such an e_3. Otherwise, the *redundant* flag must have been set in line 9, so there must exist a locker $\langle e_3, H_3 \rangle \in lockers[l]$ with $e_3 \parallel e_1$ and $H_3 \subseteq H_1$. Thus, by pseudotransitivity (Lemma 2), we have $e_3 \parallel e_2$. Moreover, since $H_3 \subseteq H_1$ and $H_1 \cap H_2 = \emptyset$, we have $H_3 \cap H_2 = \emptyset$, and therefore e_3, which belongs to $lockers[l]$, races with e_2.

To complete the proof, we now show that the locker $\langle e_3, H_3 \rangle$ is not removed from $lockers[l]$ between the times that e_1 and e_2 are executed. Suppose to the contrary that $\langle e_4, H_4 \rangle$ is a locker that causes $\langle e_3, H_3 \rangle$ to be removed from $lockers[l]$ in line 7. Then, we must have $e_3 \prec e_4$ and $H_3 \supseteq H_4$, and by Lemma 1, we have $e_4 \parallel e_2$. Moreover, since $H_3 \supseteq H_4$ and $H_3 \cap H_2 = \emptyset$, we have $H_4 \cap H_2 = \emptyset$, contradicting the choice of e_1 as the last thread before e_2 to race with e_2.

Therefore, thread e_3, which races with e_2, still belongs to $lockers[l]$ when e_2 executes, and so lines 1–3 report a race. ■

In Section 5.1, we claimed that for a Cilk program that executes in time T on one processor, references V shared memory locations, uses a total of n locks, and holds at most $k \ll n$ locks simultaneously, the ALL-SETS algorithm can check this computation for data races in $O(n^k T\,\alpha(V,V))$ time and using $O(n^k V)$ space. These bounds, which are correct but weak, are improved by the next theorem.

Theorem 4 *Consider a Cilk program that executes in time T on one processor, references V shared memory locations, uses a total of n locks, and holds at most k locks simultaneously. The* ALL-SETS *algorithm checks this computation for data races in $O(TL(k+\alpha(V,V)))$ time and $O(kLV)$ space, where L is the maximum of the number of distinct lock sets used to access any particular location.*

Proof: First, observe that no two lockers in *lockers* have the same lock set, because the logic in lines 5–11 ensure that if $H = H'$, then locker $\langle e, H\rangle$ either replaces $\langle e', H'\rangle$ (line 7) or is considered redundant (line 9). Thus, there are at most L lockers in the list *lockers*$[l]$. Each lock set takes at most $O(k)$ space, so the space needed for *lockers* is $O(kLV)$. The length of the list *lockers*$[l]$ determines the number of series/parallel relations that are tested. In the worst case, we need to perform $2L$ such tests (lines 2 and 6) and $2L$ set operations (lines 2, 6, and 8) per access. Each series/parallel test takes amortized $O(\alpha(V,V))$ time, and each set operation takes $O(k)$ time. Therefore, the ALL-SETS algorithm runs in $O(TL(k+\alpha(V,V)))$ time. ∎

The looser bounds claimed in Section 5.1 of $O(n^k T\,\alpha(V,V))$ time and $O(n^k V)$ space for $k \ll n$ follow because $L \le \sum_{i=0}^{k}\binom{n}{i} = O(n^k/k!)$. As we shall see in Section 5.4, however, we rarely see the worst-case behavior given by the bounds in Theorem 4.

5.3 The Brelly algorithm

The umbrella locking discipline requires all accesses to any particular location within a given parallel subcomputation to be protected by a single lock. Subcomputations in series may each use a different lock, or even none, if no parallel accesses to the location occur within the subcomputation. In this section, we formally define the umbrella discipline and present the BRELLY algorithm for detecting violations of this discipline.

We prove that the Brelly algorithm is correct and analyze its performance, which we show to be asymptotically better than that of All-Sets.

The umbrella discipline can be defined precisely in terms of the parse tree of a given Cilk computation. An ***umbrella*** of accesses to a location l is a subtree rooted at a P-node containing accesses to l in both its left and right subtrees, as is illustrated in Figure 5-5. An umbrella of accesses to l is ***protected*** if its accesses have a nonempty lock set and ***unprotected*** otherwise. A program obeys the ***umbrella locking discipline*** if it contains no unprotected umbrellas. In other words, within each umbrella of accesses to a location l, all threads must agree on at least one lock to protect their accesses to l.

The next theorem shows that adherence to the umbrella discipline precludes data races from occuring.

Theorem 5 *A Cilk computation with a data race violates the umbrella discipline.*

Proof: Any two threads involved in a data race must have a P-node as their least common ancestor in the parse tree, because they operate in parallel. This P-node roots an unprotected umbrella, since both threads access the same location and the lock sets of the two threads are disjoint. ■

The umbrella discipline can also be violated by unusual, but data-race free, locking protocols. For instance, suppose that a location is protected by three locks and that every thread always acquires two of the three locks before accessing the location. No single lock protects the location, but every pair of such accesses is mutually exclusive. The All-Sets algorithm properly certifies this bizarre example as race-free, whereas Brelly detects a discipline violation. In return for disallowing these unusual locking protocols (which in any event are of dubious value), Brelly checks programs asymptotically much faster than All-Sets.

Like All-Sets, the Brelly algorithm extends the SP-bags algorithm used in the original Nondeterminator and uses the r-lock fake lock for read accesses (see Section 5.2). Figure 5-6 gives pseudocode for Brelly. Like the SP-bags algorithm,

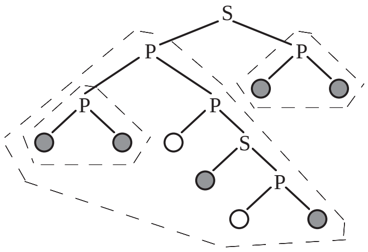


Figure 5-5: Three umbrellas of accesses to a location l. In this parse tree, each shaded leaf represents a thread that accesses l. Each umbrella of accesses to l is enclosed by a dashed line.

Brelly executes the program on a given input in serial depth-first order, maintaining the SP-bags data structure so that the series/parallel relationship between the currently executing thread and any previously executed thread can be determined quickly. Like the All-Sets algorithm, Brelly also maintains a set H of currently held locks. In addition, Brelly maintains two shadow spaces of shared memory: *accessor*, which stores for each location the thread that performed the last "serial access" to that location; and *locks*, which stores the lock set of that access. Each entry in the *accessor* space is initialized to the initial thread (which logically precedes all threads in the computation), and each entry in the *locks* space is initialized to the empty set.

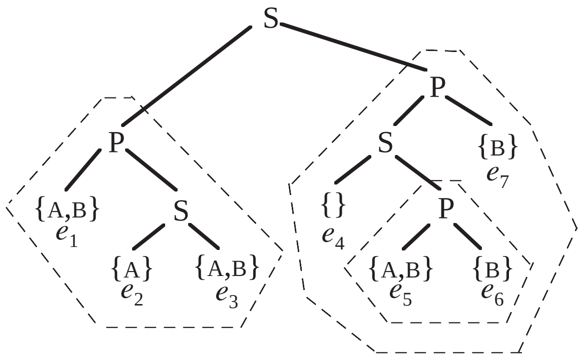
Unlike the All-Sets algorithm, Brelly keeps only a single lock set, rather than a list of lock sets, for each shared-memory location. For a location l, each lock in $locks[l]$ potentially belongs to the lock set of the largest umbrella of accesses to l that includes the current thread. The Brelly algorithm tags each lock $h \in locks[l]$ with two pieces of information: a thread $nonlocker[h]$ and a flag $alive[h]$. The thread $nonlocker[h]$ is a thread that accesses l without holding h. The flag $alive[h]$ indicates whether h should still be considered to potentially belong to the lock set of the umbrella. To allow reports of violations to be more precise, the algorithm "kills" a lock h by setting $alive[h] \leftarrow$ FALSE when it determines that h does not belong to the lock set of the umbrella, rather than simply removing it from $locks[l]$.

Whenever Brelly encounters an access by a thread e to a location l, it checks for a violation with previous accesses to l, updating the shadow spaces appropriately

```
ACCESS(l) in thread e with lock set H
 1  if accessor[l] ≺ e
 2     then ▷ serial access
             locks[l] ← H, leaving nonlocker[h] with its old
               nonlocker if it was already in locks[l] but
               setting nonlocker[h] ← accessor[l] otherwise
 3           for each lock h ∈ locks[l]
 4               do alive[h] ← TRUE
 5           accessor[l] ← e
 6     else ▷ parallel access
 7           for each lock h ∈ locks[l] − H
 8               do if alive[h] = TRUE
 9                     then alive[h] ← FALSE
10                          nonlocker[h] ← e
11           for each lock h ∈ locks[l] ∩ H
12               do if alive[h] = TRUE and nonlocker[h] ∥ e
13                     then alive[h] ← FALSE
14           if no locks in locks[l] are alive (or locks[l] = ∅)
15             then report violation on l involving
                         e and accessor[l]
16                  for each lock h ∈ H ∩ locks[l]
17                      do report access to l without h
                              by nonlocker[h]
```

Figure 5-6: The BRELLY algorithm. While executing a Cilk program in serial depth-first order, at each access to a shared-memory location l, the code shown is executed. Not shown are the updates to H, the set of currently held set of locks, which occur whenever locks are acquired or released. To determine whether the currently executing thread is in series or parallel with previously executed threads, BRELLY uses the SP-bags data structure from [37].

for future reference. If $accessor[l] \prec e$, we say the access is a ***serial access***, and the algorithm performs lines 2–5, setting $locks[l] \leftarrow H$ and $accessor[l] \leftarrow e$, as well as updating $nonlocker[h]$ and $alive[h]$ appropriately for each $h \in H$. If $accessor[l] \parallel e$, we say the access is a ***parallel access***, and the algorithm performs lines 6–17, killing the locks in $locks[l]$ that do not belong to the current lock set H (lines 7–10) or whose nonlockers are in parallel with the current thread (lines 11–13). If BRELLY discovers in line 14 that there are no locks left alive in $locks[l]$ after a parallel access, it has discovered an unprotected umbrella, and it reports a discipline violation in



thread	access type	*accessor*[l]	*locks*[l]	A	*nonlocker*[A]	B	*nonlocker*[B]
initial		e_0	$\{\}$				
e_1	serial	e_1	$\{A, B\}$	alive	e_0	alive	e_0
e_2	parallel	e_1	$\{A, B\}$	alive	e_0	killed	e_2
e_3	parallel	e_1	$\{A, B\}$	alive	e_0	killed	e_2
e_4	serial	e_4	$\{\}$				
e_5	serial	e_5	$\{A, B\}$	alive	e_4	alive	e_4
e_6	parallel	e_5	$\{A, B\}$	killed	e_6	alive	e_4
e_7	parallel	e_5	$\{A, B\}$	killed	e_6	killed	e_4

Figure 5-7: A sample execution of the BRELLY algorithm. We restrict our attention to the algorithm's operation on a single location l. In the parse tree, each leaf represents an access to l and is labeled with the thread that performs the access (e.g., e_1) and the lock set of that access (e.g., $\{A, B\}$). Umbrellas are enclosed by dashed lines. The table displays the values of *accessor*[l] and *locks*[l] after each thread's access. The state of each lock and its nonlocker are listed after *locks*[l]. The "access type" column indicates whether the access is a parallel or serial access.

lines 15–17.

When reporting a violation, BRELLY specifies the location l, the current thread e, and the thread *accessor*[l]. It may be that e and *accessor*[l] hold locks in common, in which case the algorithm uses the nonlocker information in lines 16–17 to report threads which accessed l without each of these locks.

Figure 5-7 illustrates how BRELLY works. The umbrella containing threads e_1, e_2, and e_3 is protected by lock A but not by lock B, which is reflected in *locks*[l] after thread e_3 executes. The umbrella containing e_5 and e_6 is protected by B but not by A, which is reflected in *locks*[l] after thread e_6 executes. During the execution of thread e_6, A is killed and *nonlocker*[A] is set to e_6, according to the logic in lines 7–10. When

e_7 executes, B remains as the only lock alive in *locks*$[l]$ and *nonlocker*[B] is e_4 (due to line 2 during e_5's execution). Since $e_4 \parallel e_7$, lines 11–13 kill B, leaving no locks alive in *locks*$[l]$, properly reflecting the fact that no lock protects the umbrella containing threads e_4 through e_7. Consequently, the test in line 14 causes BRELLY to declare a violation at this point.

The following two lemmas, which will be helpful in proving the correctness of BRELLY, are stated without proof.

Lemma 6 *Suppose a thread e performs a serial access to location l during an execution of* BRELLY. *Then all previously executed accesses to l logically precede e in the computation.* ∎

Lemma 7 *The* BRELLY *algorithm maintains the invariant that for any location l and lock $h \in$ locks$[l]$, the thread nonlocker$[h]$ is either the initial thread or a thread that accessed l without holding h.* ∎

Theorem 8 *The* BRELLY *algorithm detects a violation of the umbrella discipline in a computation of a Cilk program running on a given input if and only if a violation exists.*

Proof: We first show that BRELLY only detects actual violations of the discipline, and then we argue that no violations are missed. In this proof, we denote by *locks*$^*[l]$ the set of locks in *locks*$[l]$ that have TRUE *alive* flags.

($\Rightarrow$) Suppose that BRELLY detects a violation caused by a thread e, and let $e_0 = accessor[l]$ when e executes. Since we have $e_0 \parallel e$, it follows that $p = \text{LCA}(e_0, e)$ roots an umbrella of accesses to l, because p is a P-node and it has an access to l in both subtrees. We shall argue that the lock set U of the umbrella rooted at p is empty. Since BRELLY only reports violations when *locks*$^*[l] = \emptyset$, it suffices to show that $U \subseteq$ *locks*$^*[l]$ at all times after e_0 executes.

Since e_0 is a serial access, lines 2–5 cause *locks*$^*[l]$ to be the lock set of e_0. At this point, we know that $U \subseteq$ *locks*$^*[l]$, because U can only contain locks held by every access in p's subtree. Suppose that a lock h is killed (and thus removed from $locks^*[l]$), either in line 9 or line 13, when some thread e' executes a parallel access between the times that e_0 and e execute. We shall show that in both cases $h \notin U$, and so $U \subseteq locks^*[l]$ is maintained.

In the first case, if thread e' kills h in line 9, it does not hold h, and thus $h \notin U$.

In the second case, we shall show that w, the thread stored in $nonlocker[h]$ when h is killed, is a descendant of p, which implies that $h \notin U$, because by Lemma 7, w accesses l without the lock h. Assume for the purpose of contradiction that w is not a descendant of p. Then, we have $\text{LCA}(w, e_0) = \text{LCA}(w, e')$, which implies that $w \parallel e_0$, because $w \parallel e'$. Now, consider whether $nonlocker[h]$ was set to w in line 10 or in line 2 (not counting when $nonlocker[h]$ is left with its old value in line 2). If line 10 sets $nonlocker[h] \leftarrow w$, then w must execute before e_0, since otherwise, w would be a parallel access, and lock h would have been killed in line 9 by w before e' executes. By Lemma 6, we therefore have the contradiction that $w \prec e_0$. If line 2 sets $nonlocker[h] \leftarrow w$, then w performs a serial access, which must be prior to the most recent serial access by e_0. By Lemma 6, we once again obtain the contradiction that $w \prec e_0$.

($\Leftarrow$) We now show that if a violation of the umbrella discipline exists, then BRELLY detects a violation. If a violation exists, then there must be an unprotected umbrella of accesses to a location l. Of these unprotected umbrellas, let T be a maximal one in the sense that T is not a subtree of another umbrella of accesses to l, and let p be the P-node that roots T. The proof focuses on the values of $accessor[l]$ and $locks[l]$ just after p's left subtree executes.

We first show that at this point, $accessor[l]$ is a left-descendant of p. Assume for the purpose of contradiction that $accessor[l]$ is not a left-descendant of p (and is therefore not a descendant of p at all), and let $p' = \text{LCA}(accessor[l], p)$. We know that p' must be a P-node, since otherwise $accessor[l]$ would have been overwritten in line 5 by the first access in p's left subtree. But then p' roots an umbrella which is a proper superset of T, contradicting the maximality of T.

Since $accessor[l]$ belongs to p's left subtree, no access in p's right subtree overwrites

$locks[l]$, as they are all logically in parallel with $accessor[l]$. Therefore, the accesses in p's right subtree may only kill locks in $locks[l]$. It suffices to show that by the time all accesses in p's right subtree execute, all locks in $locks[l]$ (if any) have been killed, thus causing a race to be declared. Let h be some lock in $locks^*[l]$ just after the left subtree of p completes.

Since T is unprotected, an access to l unprotected by h must exist in at least one of p's two subtrees. If some access to l is not protected by h in p's right subtree, then h is killed in line 9. Otherwise, let e_{left} be the most-recently executed thread in p's left subtree that performs an access to l not protected by h. Let e' be the thread in $accessor[l]$ just after e_{left} executes, and let e_{right} be the first access to l in the right subtree of p. We now show that in each of the following cases, we have $nonlocker[h] \parallel e_{right}$ when e_{right} executes, and thus h is killed in line 13.

Case 1: Thread e_{left} is a serial access. Just after e_{left} executes, we have $h \notin locks[l]$ (by the choice of e_{left}) and $accessor[l] = e_{left}$. Therefore, when h is later placed in $locks[l]$ in line 2, $nonlocker[h]$ is set to e_{left}. Thus, we have $nonlocker[h] = e_{left} \parallel e_{right}$.

Case 2: Thread e_{left} is a parallel access and $h \in locks[l]$ just before e_{left} executes. Just after e' executes, we have $h \in locks[l]$ and $alive[h] = \text{TRUE}$, since $h \in locks[l]$ when e_{left} executes and all accesses to l between e' and e_{left} are parallel and do not place locks into $locks[l]$. By pseudotransitivity (Lemma 2), $e' \parallel e_{left}$ and $e_{left} \parallel e_{right}$ implies $e' \parallel e_{right}$. Note that e' must be a descendant of p, since if it were not, T would be not be a maximal umbrella of accesses to l. Let e'' be the most recently executed thread before or equal to e_{left} that kills h. In doing so, e'' sets $nonlocker[h] \leftarrow e''$ in line 10. Now, since both e' and e_{left} belong to p's left subtree and e'' follows e' in the execution order and comes before or is equal to e_{left}, it must be that e'' also belongs to p's left subtree. Consequently, we have $nonlocker[h] = e'' \parallel e_{right}$.

Case 3: Thread e_{left} is a parallel access and $h \notin locks[l]$ just before e_{left} executes. When h is later added to $locks[l]$, its $nonlocker[h]$ is set to e'. As above, by pseudotransitivity, $e' \parallel e_{left}$ and $e_{left} \parallel e_{right}$ implies $nonlocker[h] = e' \parallel e_{right}$.

In each of these cases, $nonlocker[h] \parallel e_{right}$ still holds when e_{right} executes, since

e_{left}, by assumption, is the most recent thread to access l without h in p's left subtree. Thus, h is killed in line 13 when e_{right} executes. ∎

Theorem 9 *On a Cilk program which on a given input executes serially in time T, uses V shared-memory locations, and holds at most k locks simultaneously, the* Brelly *algorithm runs in $O(kT\ \alpha(V, V))$ time and $O(kV)$ space.*

Proof: The total space is dominated by the *locks* shadow space. For any location l, the Brelly algorithm stores at most k locks in *locks*$[l]$ at any time, since locks are placed in *locks*$[l]$ only in line 2 and $|H| \leq k$. Hence, the total space is $O(kV)$.

Each loop in Figure 5-6 takes $O(k)$ time if lock sets are kept in sorted order, excluding the checking of *nonlocker*$[h] \parallel e$ in line 12, which dominates the asymptotic running time of the algorithm. The total number of times *nonlocker*$[h] \parallel e$ is checked over the course of the program is at most kT, requiring $O(kT\,\alpha(V, V))$ time. ∎

5.4 Experimental results

We are in the process of implementing both the All-Sets and Brelly algorithms as part of the Nondeterminator-2 debugging tool. Our experiences are therefore highly preliminary. In this section, we describe our initial results from running these two algorithms on four Cilk programs that use locks. Our implementations of All-Sets and Brelly have not yet been optimized, and so better performance than what we report here is likely to be possible.

According to Theorem 4, the factor by which All-Sets slows down a program is roughly $\Theta(Lk)$ in the worst case, where L is the maximum number of distinct lock sets used by the program when accessing any particular location, and k is the maximum number of locks held by a thread at one time. According to Theorem 9, the worst-case slowdown factor for Brelly is about $\Theta(k)$. In order to compare our experimental results with the theoretical bounds, we characterize our four test programs in terms of the parameters k and L:[4]

`maxflow`: A maximum-flow code based on Goldberg's push-relabel method [45]. Each vertex in the graph contains a lock. Parallel threads perform simple operations asynchronously on graph edges and vertices. To operate on a vertex u, a thread acquires u's lock, and to operate on an edge (u, v), the thread acquires both u's lock and v's lock (making sure not to introduce a deadlock). Thus, for this application, the maximum number of locks held by a thread is $k = 2$, and L is at most the maximum degree of any vertex.

`barnes-hut`: The n-body gravity simulation code from Section 4.3. In the tree-building phase, parallel threads race to build various parts of the octtree.[5] Each part is protected by an associated lock, and the first thread to acquire that lock builds that part of the structure. As the program never holds more than one lock at a time, we have $k = L = 1$.

`bucket`: A bucket sort [26, Section 9.4]. Parallel threads acquire the lock associated with a bucket before adding elements to it. This algorithm is analogous to the typical way a hash table is accessed in parallel. For this program, we have $k = L = 1$.

`rad`: A 3-dimensional radiosity renderer running on a "maze" scene. The original 75-source-file C code was developed in Belgium by Bekaert et. al. [6]. We used Cilk to parallelize its scene geometry calculations. Each surface in the scene has its own lock, as does each "patch" of the surface. In order to lock a patch, the surface lock must also be acquired, so that $k = 2$, and L is the maximum number of patches per surface, which increases at each iteration as the rendering is refined.

Figure 5-8 shows the preliminary results of our experiments on the test codes. These results indicate that the performance of ALL-SETS is indeed dependent on the parameter L. Essentially no performance difference exists between ALL-SETS and BRELLY when $L = 1$, but ALL-SETS gets progressively worse as L increases. On all

[4]These characterizations do not count the implicit "fake" R-LOCK used by the detection algorithms.

[5]The tree-building algorithm in our version of Barnes-Hut is different from the tree-building algorithm in the SPLASH-2 code.

	Parameters			Time (sec.)			Slowdown	
program	input	k	L	orig.	ALL.	BR.	ALL.	BR.
`maxflow`	sp. 1K	2	32	0.05	30	3	590	66
	sp. 4K	2	64	0.2	484	14	2421	68
	d. 256	2	256	0.2	263	15	1315	78
	d. 512	2	512	2.0	7578	136	3789	68
`barnes-hut`	1K	1	1	0.6	47	47	79	78
	2K	1	1	1.6	122	119	76	74
`bucket`	100K	1	1	0.3	22	22	74	73
`rad`	iter. 1	2	65	1.2	109	45	91	37
	iter. 2	2	94	1.0	179	45	179	45
	iter. 5	2	168	2.8	773	94	276	33
	iter. 13	2	528	9.1	13123	559	1442	61

Figure 5-8: Timings of our implementations on a variety of programs and inputs. (The input parameters are given as sparse/dense and number of vertices for `maxflow`, number of bodies for `barnes-hut`, number of elements for `bucket`, and iteration number for `rad`.) The parameter L is the maximum number of distinct lock sets used while accessing any particular location, and k is the maximum number of locks held simultaneously. Running times for the original optimized code, for ALL-SETS, and for BRELLY are given, as well as the slowdowns of ALL-SETS and BRELLY as compared to the original running time.

of our test programs, BRELLY runs fast enough to be useful as a debugging tool. In some cases, ALL-SETS is as fast, but in other cases, the overhead of ALL-SETS is too extreme (iteration 13 of `rad` takes over 3.5 hours) to allow interactive debugging.

5.5 Abelian programs

By checking a single computation for the absence of determinacy races, the original Nondeterminator can guarantee that a Cilk program without locking is determinate: it always produces the same answer (when run on the same input). To date, no similar claim has been made by any data-race detector for any class of programs with locks. We cannot make a general claim either, but in this section, we introduce a class of nondeterministic programs for which a determinacy claim can be made. We prove that the absence of data races in a single computation of a deadlock-free “abelian” program implies that the program (when run on the same input) is determinate. As a consequence, ALL-SETS and BRELLY can verify the determinacy of abelian programs

```
int x, y;
Cilk_lockvar A;

cilk int main() {
  Cilk_lock_init(&A);
  x = 0;
  spawn bar1();
  spawn bar2();
  sync;
  printf("%d", y);
}

cilk void bar1() {
  Cilk_lock(&A);
  x++;
  if (x == 1)
     y = 3;
  Cilk_unlock(&A);
}

cilk void bar2() {
  Cilk_lock(&A);
  x++;
  Cilk_unlock(&A);
  y = 4;
}
```

Figure 5-9: A Cilk program that generates a computation with an infeasible data race on the variable `y`.

from examining a single computation. We do not claim that abelian programs form an important class in any practical sense. Rather, we find it remarkable that a guarantee of determinacy can be made for any nontrivial class of nondeterministic programs.

Locking introduces nondeterminism intentionally, allowing many different computations to arise from the same program, some of which may have data races and some of which may not. Since All-Sets and Brelly examine only one computation, they cannot detect data races that appear in other computations. More subtlely, the data races that these algorithms do detect might actually be infeasible, never occurring in an actual program execution.

Figure 5-9 shows a program that exhibits an infeasible data race. In the computation generated when `bar1` obtains lock `A` before `bar2`, a data race exists between the two updates to `y`. In the scheduling where `bar2` obtains lock `A` first, however, `bar1`'s update to `y` never occurs. In other words, no scheduling exists in which the two updates to `y` happen simultaneously, and in fact, the final value of `y` is always 4. Thus, the computation generated by the serial depth-first scheduling, which is the one examined by All-Sets and Brelly, contains an infeasible data race.

Deducing from a single computation that the program in Figure 5-9 is determinate appears difficult. But not all programs are so hard to understand. For example, the program from Figure 5-1 exhibits a race no matter how it is scheduled, and therefore, ALL-SETS and BRELLY can always find a race. Moreover, if all accesses to `x` in the program were protected by the same lock, no data races would exist in any computation. For such a program, checking a single computation for the absence of races suffices to guarantee that the program is determinate. The reason we can verify the determinacy of this program from a single computation is because it has "commuting" critical sections.

The critical sections in the program in Figure 5-1 obey the following strict definition of commutativity: Two critical sections R_1 and R_2 ***commute*** if, beginning with any (reachable) program state S, the execution of R_1 followed by R_2 yields the same state S' as the execution of R_2 followed by R_1; and furthermore, in both execution orders, each critical section must execute the identical sequence of instructions on the identical memory locations.[6] Thus, not only must the program state remain the same, the same accesses to shared memory must occur, although the values returned by those accesses may differ. The program in Figure 5-1 also exhibits "properly nested locking." Locks are ***properly nested*** if any thread which acquires a lock A and then a lock B releases B before releasing A. We say that a program is ***abelian*** if any pair of parallel critical sections that are protected by the same lock commute, and all locks in the program are properly nested. The programs in Figures 5-1 and 2-3 are examples of abelian programs.

The idea that critical sections should commute is natural. A programmer presumably locks two critical sections with the same lock not only because he intends them to be atomic, but because he intends them to "do the same thing" no matter in what order they are executed. The programmer's notion of commutativity is usually less restrictive, however, than what our definition allows. First, both execution orders of two critical sections may produce distinct program states that the programmer

[6]It may be the case that even though R_1 and R_2 are in parallel, they cannot appear adjacent in any execution because a lock is acquired preceeding R_1 and released after R_1 which is also acquired by R_2 (or vice versa). Therefore, we require the additional technical condition that the execution of R_1 followed by any prefix R_2' of R_2 generates for R_2' the same instructions operating on the same locations as executing R_2' alone.

nevertheless views as equivalent. Our definition insists that the program states be identical. Second, even if they leave identical program states, the two execution orders may cause different memory locations to be accessed. Our definition demands that the same memory locations be accessed.

In practice, therefore, most programs are not abelian, but abelian programs nevertheless form a nontrivial class of nondeterministic programs that can be checked for determinacy. For example, all programs that use locking to accumulate values atomically, such as the histogram program in Figure 2-3, fall into this class. Although abelian programs form an arguably small class in practice, the guarantees of determinacy that All-Sets and Brelly provide for them are not provided by any other existing race-detectors for *any* class of lock-employing programs. It is an open question whether a more general class of nondeterministic programs exists for which an efficient race-detector can offer a provable guarantee of determinacy.

In order to study the determinacy of abelian programs, we first give a formal multithreaded machine model that more precisely describes an actual execution of a Cilk program. We view the abstract execution machine for Cilk as a (sequentially consistent [63]) shared memory together with a collection of ***interpreters***, each with some private state. (See [15, 28, 51] for examples of multithreaded implementations similar to this model.) Interpreters are dynamically created during execution by each `spawn` statement. The ith such child of an interpreter is given a unique ***interpreter name*** by appending i to its parent's name.

When an instruction is ***executed*** by an interpreter, it maps the current state of the multithreaded machine to a new state. An interpreter whose next instruction cannot be executed is said to be ***blocked***. If all interpreters are blocked, the machine is ***deadlocked***.

Although a multithreaded execution may proceed in parallel, we consider a serialization of the execution in which only one interpreter executes at a time, but the instructions of the different interpreters may be interleaved.[7] The initial state

[7]The fact that any parallel execution can be simulated in this fashion is a consequence of our choice of sequential consistency as the memory model.

of the machine consists of a single interpreter whose program counter points to the first instruction of the program. At each step, a nondeterministic choice among the current nonblocked interpreters is made, and the instruction pointed to by its program counter is executed. The resulting sequence of instructions is referred to as an ***execution*** of the program.

When an instruction executes in a run of a program, it affects the state of the machine in a particular way. To formalize the effect of an instruction execution, we define an ***instantiation*** of an instruction to be a 3-tuple consisting of an instruction I, the shared memory location l on which I operates (if any), and the name of the interpreter that executes I. We assume that the instantiation of an instruction is a deterministic function of the machine state.

We define a ***region*** to be either a single instantiation other than a LOCK or UNLOCK instruction, or a sequence of instantiations that comprise a critical section (including the LOCK and UNLOCK instantiations themselves).[8] Every instantiation belongs to at least one region and may belong to many. Since a region is a sequence of instantiations, it is determined by a particular execution of the program and not by the program code alone. We define the ***nesting count*** of a region R to be the maximum number of locks that are acquired in R and held simultaneously at some point in R.

The execution of a program can alternatively be viewed as sequence of instantiations, rather than instructions, and an instantiation sequence can always be generated from an instruction sequence. We formally define a ***computation*** as a dag in which the vertices are instantiations and the edges denote synchronization. Edges go from each instantiation to the next instantiation executed by the same interpreter, from each spawn instantiation to the first instantiation executed by the spawned interpreter, and from the last instantiation of each interpreter to the next sync instantiation executed by its parent interpreter.

We can now give a more precise definition of a data race. A ***data race*** exists in a

[8] The instantiations within a critical section must be serially related in the dag, as we disallow parallel control constructs while locks are held.

computation if two logically parallel instantiations access the same memory location without holding the same lock, and at least one of the accesses is a WRITE. Since a memory location is a component of each instantiation, it is unambiguous what it means for two instantiations to access the same memory location. In contrast, if the computation were constructed so that the nodes were instructions, it would not be apparent from the dag alone whether two nodes reference the same memory location.

A ***scheduling*** of a computation G is a sequence of instantiations forming a permutation of the vertex set of G. This sequence must satisfy the ordering constraints of the dag, as well as have the property that any two LOCK instantiations that acquire the same lock are separated by an UNLOCK of that lock in between. A ***partial scheduling*** of G is a scheduling of a prefix of G, and if any partial scheduling of G can be extended to a scheduling of G, we say that G is ***deadlock free***. Otherwise, G has at least one ***deadlock scheduling***, which is a partial scheduling that cannot be extended.

Not every scheduling of G corresponds to some actual execution of the program. If a scheduling or partial scheduling does correspond to an actual execution as defined by the machine model, we call that scheduling a ***true scheduling*** of G; otherwise it is a ***false scheduling***. Since we are only concerned with the final memory states of true schedulings, we define two schedulings (or partial schedulings) of G to be ***equivalent*** if both are false, or both are true and have the same final memory state. An alternate definition of commutativity, then, is that two regions R_1 and R_2 commute if, beginning with any reachable machine state S, the instantiation sequences R_1R_2 and R_2R_1 are equivalent.

Our study of the determinacy of abelian programs will proceed as follows. Starting with a data-race free, deadlock-free computation G resulting from the execution of an abelian program, we first prove that adjacent regions in a scheduling of G can be commuted. Second, we show that regions which are spread out in a scheduling of G can be grouped together. Third, we prove that all schedulings of G are true and yield the same final memory state. Finally, we prove that all executions of the abelian program generate the same computation and hence the same final memory state.

Lemma 10 (Reordering) *Let G be a data-race free, deadlock-free computation resulting from the execution of an abelian program. Let X be some scheduling of G. If regions R_1 and R_2 appear adjacent in X, i.e., $X = X_1 R_1 R_2 X_2$, and $R_1 \parallel R_2$, then the two schedulings $X_1 R_1 R_2 X_2$ and $X_1 R_2 R_1 X_2$ are equivalent.*

Proof: We prove the lemma by double induction on the nesting count of the regions. Our inductive hypothesis is the theorem as stated for regions R_1 of nesting count i and regions R_2 of nesting count j.

Base case: $i = 0$. Then R_1 is a single instantiation. Since R_1 and R_2 are adjacent in X and are parallel, no instantiation of R_2 can be guarded by a lock that guards R_1, because any lock held at R_1 is not released until after R_2. Therefore, since G is data-race free, either R_1 and R_2 access different memory locations or R_1 is a READ and R_2 does not write to the location read by R_1. In either case, the instantiations of each of R_1 and R_2 do not affect the behavior of the other, so they can be executed in either order without affecting the final memory state.

Base case: $j = 0$. Symmetric with above.

Inductive step: In general, R_1 of count $i \geq 1$ has the form LOCK(A) $\cdots$ UNLOCK(A), and R_2 of count $j \geq 1$ has the form LOCK(B) $\cdots$ UNLOCK(B). If A = B, then R_1 and R_2 commute by the definition of abelian. Otherwise, there are three possible cases.

Case 1: Lock A appears in R_2, and lock B appears in R_1. This situation cannot occur, because it implies that G is not deadlock free, a contradiction. To construct a deadlock scheduling, we schedule X_1 followed by the instantiations of R_1 up to (but not including) the first LOCK(B). Then, we schedule the instantiations of R_2 until a deadlock is reached, which must occur, since R_2 contains a LOCK(A) (although the deadlock may occur before this instantiation is reached).

Case 2: Lock A does not appear in R_2. We start with the sequence $X_1 R_1 R_2 X_2$ and commute pieces of R_1 one at a time with R_2: first, the instantiation UNLOCK(A), then the (immediate) subregions of R_1, and finally the instantiation LOCK(A). The instantiations LOCK(A) and UNLOCK(A) commute with R_2, because A does not appear anywhere in R_2. Each subregion of R_1 commutes with R_2 by the inductive hypothesis, because each subregion has lower nesting count than R_1. After commuting all of R_1

past R_2, we have an equivalent execution $X_1 R_2 R_1 X_2$.

Case 3: Lock B does not appear in R_1. Symmetric to Case 2. ■

Lemma 11 (Region grouping) *Let G be a data-race free, deadlock-free computation resulting from the execution of an abelian program. Let X be some scheduling of G. Then, there exists an equivalent scheduling X' of G in which the instantiations of every region are contiguous.*

Proof: We shall create X' by grouping the regions in X one at a time. Each grouping operation will not destroy the grouping of already grouped regions, so eventually all regions will be grouped.

Let R be a noncontiguous region in X that completely overlaps no other noncontiguous regions in X. Since region R is noncontiguous, other regions parallel with R must overlap R in X. We first remove all overlapping regions which have exactly one endpoint (an endpoint is the bounding LOCK or UNLOCK of a region) in R, where by "in" R, we mean appearing in X between the endpoints of R. We shall show how to remove regions which have only their UNLOCK in R. The technique for removing regions with only their LOCK in R is symmetric.

Consider the partially overlapping region S with the leftmost UNLOCK in R. Then all subregions of S which have any instantiations inside R are completely inside R and are therefore contiguous. We remove S by moving each of its (immediate) subregions in R to just left of R using commuting operations. Let S_1 be the leftmost subregion of S which is also in R. We can commute S_1 with every instruction I to its left until it is just past the start of R. There are three cases for the type of instruction I. If I is not a LOCK or UNLOCK, it commutes with S_1 by Lemma 10 because it is a region in parallel with S_1. If $I = \text{LOCK}(\text{B})$ for some lock B, then S_1 commutes with I, because S_1 cannot contain LOCK(B) or UNLOCK(B). If $I = \text{UNLOCK}(\text{B})$, then there must exist a matching LOCK(B) inside R, because S is chosen to be the region with the leftmost UNLOCK without a matching LOCK. Since there is a matching LOCK in R, the region defined by the LOCK/UNLOCK pair must be contiguous by the choice of R. Therefore, we can commute S_1 with this whole region at once using Lemma 10.

We can continue to commute S_1 to the left until it is just before the start of R. Repeat for all other subregions of S, left to right. Finally, the UNLOCK at the end of S can be moved to just before R, because no other LOCK or UNLOCK of that same lock appears in R up to that UNLOCK.

Repeat this process for each region overlapping R that has only an UNLOCK in R. Then, remove all regions which have only their LOCK in R by pushing them to just after R using similar techniques. Finally, when there are no more unmatched LOCK or UNLOCK instantiations in R, we can remove any remaining overlapping regions by pushing them in either direction to just before or just after R. The region R is now contiguous.

Repeating for each region, we obtain an execution X' equivalent to X in which each region is contiguous. ∎

Lemma 12 *Let G be a data-race free, deadlock-free computation resulting from the execution of an abelian program. Then every scheduling of G is true and yields the same final memory state.*

Proof: Let X be the execution that generates G. Then X is a true scheduling of G. We wish to show that any scheduling Y of G is true. We shall construct a set of equivalent schedulings of G that contain the schedulings X and Y, thus proving the lemma.

We construct this set using Lemma 11. Let X' and Y' be the schedulings of G with contiguous regions which are obtained by applying Lemma 11 to X and Y, respectively. From X' and Y', we can commute whole regions using Lemma 10 to put their threads in the serial depth-first order specified by G, obtaining schedulings X'' and Y''. We have $X'' = Y''$, because a computation has only one serial depth-first scheduling. Thus, all schedulings X, X', $X'' = Y''$, Y', and Y are equivalent. Since X is a true scheduling, so is Y, and both have the same final memory state. ∎

Theorem 13 *An abelian Cilk program that produces a deadlock-free computation with no data races is determinate.*

Proof: Let X be an execution of an abelian program that generates a data-race free, deadlock-free computation G. Let Y be an arbitrary execution of the same program. Let H be the computation generated by Y, and let H_i be the prefix of H that is generated by the first i instantiations of Y. If H_i is a prefix of G for all i, then $H = G$, and therefore, by Lemma 12, executions X and Y have the same final memory state. Otherwise, assume for contradiction that i_0 is the largest value of i for which H_i is a prefix of G. Suppose that the (i_0+1)st instantiation of Y is executed by an interpreter with name η. We shall derive a contradiction through the creation of a new scheduling Z of G. We construct Z by starting with the first i_0 instantiations of Y, and next adding the successor of H_{i_0} in G that is executed by interpreter η. We then complete Z by adding, one by one, any nonblocked instantiation from the remaining portion of G. One such instantiation always exists because G is deadlock free. By Lemma 12, the scheduling Z that results is a true scheduling of G. We thus have two true schedulings which are identical in the first i_0 instantiations but which differ in the $(i_0 + 1)$st instantiation. In both schedulings the $(i_0 + 1)$st instantiation is executed by interpreter η. But, the state of the machine is the same in both Y and Z after the first i_0 instantiations, which means that the $(i_0 + 1)$st instantiation must be the same for both, which is a contradiction. ∎

We state one more lemma which allows us to show that ALL-SETS and BRELLY can give a guarantee of determinacy for deadlock-free abelian programs. We leave the proof of this lemma to Appendix A because of its technical nature.

Lemma 14 *Let G be a computation generated by a deadlock-free abelian program. If G is data-race free, then it is deadlock free.*

Corollary 15 *If the* ALL-SETS *algorithm detects no data races in an execution of a deadlock-free abelian Cilk program, then the program running on the same input is determinate.*

Proof: Combine Theorems 3 and 13 and Lemma 14. ∎

Corollary 16 *If the* BRELLY *algorithm detects no violations of the umbrella discipline in an execution of a deadlock-free abelian Cilk program, then the program run on the same input is determinate.*

Proof: Combine Theorems 5, 8, and 13 and Lemma 14. ■

5.6 Conclusion

Although ALL-SETS and BRELLY are fast race-detection algorithms, many practical questions remain as to how to use these algorithms to debug real programs. In this section, we discuss our early experiences in using the Nondeterminator-2, which currently provides both algorithms as options, to debug Cilk programs.

A key decision by Cilk programmers is whether to adopt the umbrella locking discipline. A programmer might first debug with ALL-SETS, but unless he has adopted the umbrella discipline, he will be unable to fall back on BRELLY if ALL-SETS seems too slow. We recommend that programmers use the umbrella discipline initially, which is good programming practice in any event, and only use ALL-SETS if they are forced to drop the discipline.

The Nondeterminator-2 reports any apparent data race as a bug. As we have seen, however, some data races are infeasible. We have experimented with ways that the user can inform the Nondeterminator-2 that certain races are infeasible, so that the debugger can avoid reporting them. One approach we have tried is to allow the user to "turn off" the Nondeterminator-2 in certain pieces of code using compiler pragmas and other linguistic mechanisms. Unfortunately, turning off the Nondeterminator-2 requires the user to check for data races manually between the ignored accesses and all other accesses in the program. A better strategy has been to give the user fake locks—locks that are acquired and released only in debugging mode, as in the implicit R-LOCK fake lock. The user can then protect accesses involved in apparent but infeasible races using a common fake lock. Fake locks reduce the number of false reports made by the Nondeterminator-2, and they require the user to manually check for data races only between critical sections locked by the same fake lock.

Another cause of false reports is "publishing." One thread allocates a heap object, initializes it, and then "publishes" it by atomically making a field in a global data structure point to the new object so that the object is now available to other threads. If a logically parallel thread now accesses the object in parallel through the global data structure, an apparent data race occurs between the initialization of the object and the access after it was published. Fake locks do not seem to help much, because it is hard for the initializer to know all the other threads that may later access the object, and we do not wish to suppress data races among those later accesses. We do not yet have a good solution for this problem.

With the BRELLY algorithm, some programs may generate many violations of the umbrella discipline that are not caused by actual data races. We have implemented several heuristics in the Nondeterminator-2's BRELLY mode to report straightforward data races and hide violations that are not real data races whenever possible.

False reports are not a problem when the program being debugged is abelian, but programmers would like to know whether an ostensibly abelian program is actually abelian. Dinning and Schonberg give a conservative compile-time algorithm to check if a program is "internally deterministic" [31], and we have given thought to how the abelian property might likewise be conservatively checked. The parallelizing compiler techniques of Rinard and Diniz [87] may be applicable.

We are currently investigating versions of ALL-SETS and BRELLY that correctly detect races even when parallelism is allowed within critical sections. A more ambitious goal is to detect potential deadlocks by dynamically detecting the user's accordance with a flexible locking discipline that precludes deadlocks.

Chapter 6

Dag consistency

This chapter defines dag consistency, a weak memory model for multithreaded computing, and presents the BACKER algorithm for maintaining dag consistency.[1] We argue that dag consistency is a natural consistency model for Cilk programs, and we give both theoretical and empirical evidence that the BACKER algorithm is efficient. We prove that the number of page faults (cache misses) $F_P(C)$ incurred by BACKER running on P processors, each with a shared-memory cache of C pages, is at most $F_1(C)+2Cs$, where where s is the number of steals executed by Cilk's scheduler. The $F_1(C)$ term represents the page faults incurred by the serial execution, and the $2Cs$ term represents additional faults due to "warming up" the processors' caches on each steal. We present empirical evidence that this warm-up overhead is actually much smaller in practice than the theoretical bound.

6.1 Introduction

Why do we care about weak memory consistency models? Architects of shared memory for parallel computers have attempted to support Lamport's strong model of sequential consistency [63]: *The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations*

[1] The contents of this chapter are joint work with Robert Blumofe, Matteo Frigo, Christopher Joerg, and Charles Leiserson and appeared at IPPS'96 [12].

of each individual processor appear in this sequence in the order specified by its program. Unfortunately, they have generally found that Lamport's model is difficult to implement efficiently, and hence relaxed models of shared-memory consistency have been developed [33, 43, 44] that compromise on semantics for a faster implementation. By and large, all of these consistency models have had one thing in common: they are "processor centric" in the sense that they define consistency in terms of actions by physical processors. In contrast, dag consistency is defined on the abstract computation dag of a Cilk program, and hence is "computation centric".

To define a computation-centric memory model like dag consistency, it suffices to define what values are allowed to be returned by a read. Intuitively, a read can "see" a write in the dag-consistency model only if there is some serial execution order consistent with the dag in which the read sees the write. Unlike sequential consistency, but similar to certain processor-centric models [43, 47], dag consistency allows different reads to return values that are based on different serial orders, but the values returned must respect the dependencies in the dag.

The mechanisms to support dag-consistent distributed shared memory on the Connection Machine CM5 are implemented in software. Nevertheless, codes such as matrix multiplication run efficiently, as can be seen in Figure 6-1. The dag-consistent shared memory performs at 5 megaflops per processor as long as the work per processor is sufficiently large. This performance compares fairly well with other matrix multiplication codes on the CM5 (that do not use the CM5's vector units). For example, an implementation coded in Split-C [27] attains just over 6 megaflops per processor on 64 processors using a static data layout, a static thread schedule, and an optimized assembly-language inner loop. In contrast, Cilk's dag-consistent shared memory is mapped across the processors dynamically, and the Cilk threads performing the computation are scheduled dynamically at runtime. We believe that the overhead in our CM5 implementation can be reduced, but that in any case, this overhead is a reasonable price to pay for the ease of programming and dynamic load balancing provided by Cilk.

The primary motivation for any weak consistency model, including dag consis-

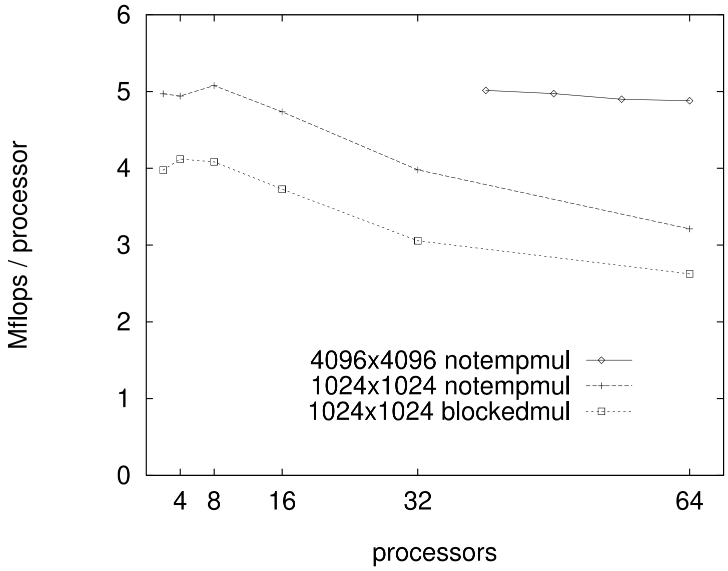


Figure 6-1: Megaflops per processor versus the number of processors for several matrix multiplication runs on the Connection Machine CM5. The shared-memory cache on each processor is set to 2MB. The lower curve is for the `blockedmul` code in Figure 4-2 and the upper two curves are for the `notempmul` code in Figure 4-3.

tency, is performance. In addition, however, a memory model must be understandable by a programmer. We argue that dag consistency is a reasonable memory model for a programmer to use. If the programmer wishes to ensure that a read sees a write, he must ensure that there is a path in the computation dag from the write to the read. The programmer ensures that such a path exists by placing a `sync` statement between the write and read in his program. In fact, our experience shows that most of the Cilk programs from Chapter 4 already have this property. Thus, they work without modification under dag consistency. All Cilk applications that do not require locks, including all of the matrix algorithms from Section 4.1 and Section 4.2, a version of the Barnes-Hut algorithm from Section 4.3 that does not parallelize the tree build, and the Rubik's cube solver from Section 4.4 require only dag consistency.

Irregular applications like Barnes-Hut and Strassen's algorithm provide a good test of Cilk's ability to schedule computations dynamically. We achieve a speedup of 9 on an 8192-particle Barnes-Hut simulation using 32 processors, which is competitive with other software implementations of distributed shared memory [59] on the CM5. Strassen's algorithm runs as fast as regular matrix multiplication on a small number of processors for 2048×2048 matrices.

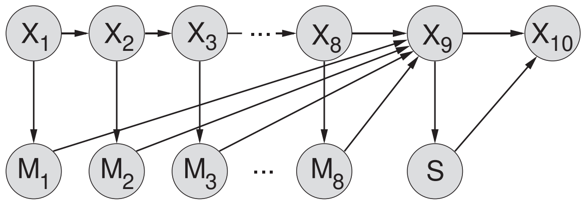


Figure 6-2: Dag of the blocked matrix multiplication algorithm `blockedmul`.

The remainder of this chapter is organized as follows. Section 6.2 gives a motivating example of dag consistency using the `blockedmul` matrix multiplication algorithm. Section 6.3 gives a formal definition of dag consistency and describes the abstract BACKER coherence algorithm for maintaining dag consistency. Section 6.4 describes an implementation of the BACKER algorithm on the Connection Machine CM5. Section 6.5 analyzes the number of faults taken by Cilk programs, both theoretically and empirically. Section 6.6 investigates the running time of dag-consistent shared memory programs and presents a model for their performance. Section 5.6 compares dag-consistency with some related consistency models and offers some ideas for future work.

6.2 Example: matrix multiplication

To illustrate the concepts behind dag consistency, consider once again the parallel matrix multiplication algorithm from Figure 4-1. Like any Cilk computation, the execution of `blockedmul` can be viewed as a dag of threads. Figure 6-2 illustrates the structure of the dag for `blockedmul`. The `spawn` and `sync` statements of the procedure `blockedmul` break it up into ten threads $X_1, \ldots, X_{10}$, where thread X_1 corresponds to the partitioning of the matrices and the spawning of subproblem M_1 in lines 1–13, threads X_2 through X_8 correspond to the spawning of subproblems M_2 through M_8 in lines 14–20, thread X_9 corresponds to the spawning of the addition S in line 22, and thread X_{10} corresponds to the `return` in line 25.

Dag-consistent shared memory is a natural consistency model to support a shared-memory program such as `blockedmul`. Certainly, sequential consistency can guarantee the correctness of the program, but a closer look at the precedence relation given by the dag reveals that a much weaker consistency model suffices. Specifically, the 8 recursively spawned children $M_1, M_2, \ldots, M_8$ need not have the same view of shared memory, because the portion of shared memory that each writes is neither read nor written by the others. On the other hand, the parallel addition of `tmp` into `R` by the computation S requires S to have a view in which all of the writes to shared memory by $M_1, M_2, \ldots, M_8$ have completed.

The intuition behind dag consistency is that each thread sees values that are consistent with some serial execution order of the dag, but two different threads may see different serial orders. Thus, the writes performed by a thread are seen by its successors, but threads that are incomparable in the dag may or may not see each other's writes. In `blockedmul`, the computation S sees the writes of $M_1, M_2, \ldots, M_8$, because all the threads of S are successors of $M_1, M_2, \ldots, M_8$, but since the M_i are incomparable, they cannot depend on seeing each others writes. We define dag consistency precisely in Section 6.3.

6.3 The BACKER coherence algorithm

This section describes our coherence algorithm, which we call BACKER, for maintaining dag consistency. We first give a formal definition of dag-consistent shared memory and explain how it relates to the intuition of dag consistency that we have gained thus far. We then describe the cache and "backing store" used by BACKER to store shared-memory objects, and we give three fundamental operations for moving shared-memory objects between cache and backing store. Finally, we give the BACKER algorithm and describe how it ensures dag consistency.

Shared memory consists of a set of objects that threads can read and write. To track which thread 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 thread performing the write. We assume without loss of generality that each thread performs at most one read or write. In addition, we make the technical assumption that an initial sequence of instructions writes a value to every object. We now define dag consistency in terms of the computation. A computation is represented by its graph $G = (V, E)$, where V is a set of vertices representing threads of the computation, and E is a set of edges representing ordering constraints on the threads. For two threads u and v, we say $u \prec v$ if $u \neq v$ and there is a directed path in G from u to v.

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

1. *For all instructions $u \in V$, the instruction $f_x(u)$ writes to x.*

2. *If an instruction u writes to x, then we have $f_x(u) = u$.*

3. *If an instruction u reads x, it receives a value tagged with $f_x(u)$.*

4. *For all instructions $u \in V$, we have $u \not\prec f_x(u)$.*

5. *For each triple u, v, and w of instructions such that $u \prec v \prec w$, if $f_x(v) \neq f_x(u)$ holds, then we have $f_x(w) \neq f_x(u)$.*

Informally, the observer function $f_x(u)$ represents the viewpoint of instruction u on the contents of object x, that is, the tag of x from u's perspective. Therefore, if an instruction u writes, the tag of x becomes u (part 2 of the definition), and when it reads, it reads something tagged with $f_x(u)$ (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 6-3. When there is a path from u to w through v, then v "masks" u, in the sense that if the value observed by u is no longer current when v executes, then it cannot be current when w executes. Instruction w can still have a different viewpoint on x than v. For instance, instruction w may see a write on x performed by some other instruction (such as s and t in the figure) that is incomparable with v.

For deterministic programs, this definition implies the intuitive notion that a read can "see" a write only if there is some serial execution order of the dag in

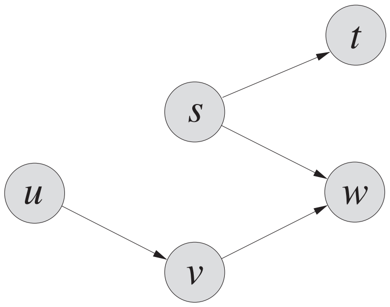


Figure 6-3: Illustration of the definition of dag consistency. When there is a path from u to w through v, then a write by v to an object "masks" u's write to the object, not allowing u's write to be read by w. Instruction w may see writes to the object performed by instructions s and t, however.

which the read sees the write. As it turns out, however, this intuition is ill defined for certain nondeterministic programs. For example, there exist nondeterministic programs whose parallel execution can contain reads that do not occur in any serial execution. Definition 1 implies the intuitive semantics for deterministic programs and is well defined for all programs.

Programs can easily be written that are guaranteed to be deterministic. Nondeterminism arises when there is a "determinacy race", a write to an object that is incomparable with another read or write to the same object. To avoid nondeterminism, it suffices that no write to an object occurs that is incomparable with another read or write to the same object, in which case all writes to the object must lie on a single path in the dag. Moreover, all writes and any one given read must also lie on a single path. Consequently, by Definition 1, every read of an object sees exactly one write to that object, and the execution is deterministic. This determinism guarantee can be verified by the Nondeterminator [37], which checks for determinacy races.[2]

We now describe the BACKER coherence algorithm for maintaining dag-consistent shared memory.[3] In this algorithm, versions of shared-memory objects can reside simultaneously in any of the processors' local caches or the global backing store. Each processor's ***cache*** contains objects recently used by the threads that have executed

[2]The Nondeterminator-2 is not required in this case because locks are not part of the definition of dag consistency.

[3]See [58] for details of a "lazier" coherence algorithm than BACKER based on climbing the spawn tree.

on that processor, and the ***backing store*** provides default global storage for each object. For our Cilk system on the CM5, portions of each processor's main memory are reserved for the processor's cache and for a portion of the distributed backing store, although on some systems, it might be reasonable to implement the backing store on disk. 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.

Three basic operations are used by the BACKER 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. Unlike implementations of other models of consistency, all three operations are bilateral between a processor's cache and the backing store, and other processors' caches are never involved.

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, the BACKER performs additional reconciles and flushes to enforce dag consistency. For each edge $i \to j$ in the computation dag, if threads i and j are scheduled on different processors, say p and q, then BACKER reconciles all of p's cached objects after p executes i but before p enables j, and it reconciles and flushes all of q's cached objects before q executes j. Although flushing all of q's objects seems extreme, we are nevertheless able to achieve good performance with BACKER, because these flushes do not happen very often. In a production implementation of BACKER, however, some means of keeping objects in a cache across multiple synchronizations might be warranted.

The key reason BACKER works is that it is always safe, at any point during the execution, for a processor p to reconcile an object or to flush a clean object. Suppose we arbitrarily insert a reconcile of an object into the computation performed by p. Assuming that there is no other communication involving p, if p later fetches the object that it previously reconciled, it will receive either the value that it wrote earlier or a value written by a thread i' that is incomparable with the thread i performing the read. In the first case, part 5 of Definition 1 is satisfied by the semantics of ordinary serial execution. In the second case, the thread i' that performed the write is incomparable with i, and thus part 5 of the definition holds because there is no path from i' to i. The other four parts of Definition 1 are easy to verify.

The BACKER algorithm uses this safety property to guarantee dag consistency even when there is communication. Suppose that a thread i resides on processor p with an edge to a thread j on processor q. In this case, 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. At this point, the state of q's cache (empty) is the same as p's if j had executed with i on processor p, but a reconcile and flush had occurred between them. Consequently, BACKER ensures dag consistency.[4]

With all the reconciles and flushes being performed by the BACKER algorithm, why should we expect it to be an efficient coherence algorithm? The main reason is that once a processor has fetched an object into its cache, the object never needs to be updated with external values or invalidated, unless communication involving that processor occurs to enforce a dependency in the dag. Consequently, the pro-

[4]For a rigorous proof that BACKER maintains dag consistency, see [69]. In fact, BACKER maintains a stronger memory model, called ***location consistency***, which is the weakest memory model stronger than dag consistency that is ***constructible*** (exactly implementable by an on-line algorithm). For a full discussion of constructibility in memory models and the relation of dag consistency to other memory models, see [40, 42].

cessor can run with the speed of a serial algorithm with no overheads. Moreover, in Cilk, communication to enforce dependencies can be amortized against steals, so such communication does not happen often if the computation has enough parallel slackness.

6.4 Implementation

This section describes our implementation of dag-consistent shared memory for the Cilk-3 runtime system running on the Connection Machine Model CM5 parallel supercomputer [66]. We describe the grouping of shared memory into pages and describe the "diff" mechanism [61] for managing dirty bits. Finally, we discuss minor anomalies in atomicity that can occur when the size of the concrete objects supported by the shared-memory system is different from the abstract objects that the programmer manipulates.

The Cilk-3 system on the CM5 supports concrete shared-memory objects of 32-bit words. All consistency operations are logically performed on a per-word basis. If the runtime system had to operate on every word independently, however, the system would be terribly inefficient. Since extra fetches and reconciles do not adversely affect the BACKER coherence algorithm, we implemented the familiar strategy of grouping objects into ***pages*** [54, Section 8.2], each of which is fetched or reconciled as a unit. Assuming that spatial locality exists when objects are accessed, grouping objects helps amortize the runtime system overhead.

An important issue we faced with the implementation of dag-consistent shared memory was how to keep track of which objects on a page have been written. Current microprocessors do not provide hardware support for maintaining user-level dirty bits at the granularity of words. Rather than using dirty bits explicitly, Cilk uses a ***diff*** mechanism as is used in the Treadmarks system [61]. The diff mechanism computes the dirty bit for an object by comparing that object's value with its value in a ***twin*** copy made at fetch time. Our implementation makes this twin copy only for pages loaded in read/write mode, thereby avoiding the overhead of copying for read-only pages. The diff mechanism imposes extra overhead on each reconcile, but it imposes no extra overhead on each access [102].

Dag consistency can suffer from atomicity anomalies when abstract objects that the programmer is reading and writing are larger than the concrete objects supported by the shared-memory system. For example, suppose the programmer is treating two 4-byte concrete objects as one 8-byte abstract object. If two incomparable threads each write the entire 8-byte object, the programmer might expect an 8-byte read of the structure by a common successor to receive one of the two 8-byte values written. The 8-byte read may nondeterministically receive 4 bytes of one value and 4 bytes of the other value, however, since the 8-byte read is really two 4-byte reads, and the consistency of the two halves is maintained separately. Fortunately, this problem can only occur if the abstract program is nondeterministic, that is, if the program is nondeterministic even when the abstract and concrete objects are the same size. When writing deterministic programs, the programmer need not worry about this atomicity problem.

As with other consistency models, including sequential consistency, atomicity anomalies can also occur when the programmer packs several abstract objects into a single system object. Fortunately, this problem can easily be avoided in the standard way by not packing together abstract objects that might be updated in parallel.

The size of the backing store determines how large a shared-memory application one can run. On the CM5, the backing store is implemented in a distributed fashion by allocating a large fraction of each processor's memory to this function. To determine which processor holds the backing store for a page, a hash function is applied to the page identifier (a pair of the virtual address and the allocating subcomputation). A fetch or reconcile request for a page is made to the backing store of the processor to which the page hashes. This policy ensures that backing store is spread evenly across the processors' memory. In other systems, it might be reasonable to place the backing store on disk *à la* traditional virtual memory.

6.5 An analysis of page faults

In this section, we examine the number $F_P(C)$ of page faults that a Cilk computation incurs when run on P processors using the implementation of the BACKER coherence algorithm with cache size C described in Section 6.4. Although we will prove tighter bounds later in Section 7.2, this analysis is provided to give an intuition for why BACKER performs well when exectuing Cilk programs. We prove that $F_P(C)$ can be related to the number $F_1(C)$ of page faults taken by a 1-processor execution by the formula $F_P(C) \leq F_1(C) + 2Cs$, where C is the size of each processor's cache in pages and s is the total number of steals executed by the scheduler. The $2Cs$ term represents faults due to "warming up" the processors' caches, and we present empirical evidence that this overhead is actually much smaller in practice than the theoretical bound.

We begin with a theorem that bounds the number of page faults of a Cilk application. The proof takes advantage of properties of the least-recently used (LRU) page replacement scheme used by Cilk, as well as the fact that Cilk's scheduler, like C, executes serial code in a depth-first fashion.

Theorem 17 *Let $F_P(C)$ be the number of page faults of a Cilk computation when run on P processors with a cache of C pages on each processor. Then, we have $F_P(C) \leq F_1(C) + 2Cs$, where s is the total number of steals that occur during Cilk's execution of the computation.*

Proof: The proof is by induction on the number s of steals. For the base case, observe that if no steals occur, then the application runs entirely on one processor, and thus it faults $F_1(C)$ times by definition. For the inductive case, consider an execution E of the computation that has s steals. Choose any subcomputation T from which no processor steals during the execution E. Construct a new execution E' of the computation which is identical to E, except that T is never stolen. Since E' has only $s-1$ steals, we know it has at most $F_1(C) + 2C(s-1)$ page faults by the inductive hypothesis.

To relate the number of page faults during execution E to the number during execution E', we examine cache behavior under LRU replacement. Consider two processors that execute simultaneously and in lock step a block of code using two different starting cache states, where each processor's cache has C pages. The main property of LRU we exploit is that the number of page faults in the two executions can differ by at most C page faults. This property follows from the observation that no matter what the starting cache states might be, the states of the two caches must be identical after one of the two executions takes C page faults. Indeed, at the point when one execution has just taken its Cth page fault, each cache contains exactly the last C distinct pages referenced [25].

We can now count the number of page faults during the execution E. The fault behavior of E is the same as the fault behavior of E' except for the subcomputation T and the subcomputation, call it U, from which it stole. Since T is executed in depth-first fashion, the only difference between the two executions is that the starting cache state of T and the starting cache state of U after T are different. Therefore, execution E makes at most $2C$ more page faults than execution E', and thus execution E has at most $F_1(C) + 2C(s-1) + 2C = F_1(C) + 2Cs$ page faults. ∎

Theorem 17 says that the total number of faults on P processors is at most the total number of faults on 1 processor plus an overhead term. The overhead arises whenever a steal occurs, because in the worst case, the caches of both the thieving processor and the victim processor contain no pages in common compared to the situation when the steal did not occur. Thus, they must be "warmed up" until the caches "synchronize" with the cache of a serial execution. If most stolen tasks touch less than C shared-memory pages, however, then the warm-up overhead will not be as large as the worst-case bound in Theorem 17.

To measure the warm-up overhead, we counted the number of page faults taken by several applications—including `blockedmul`, `notempmul`, and Strassen's algorithm—for various choices of cache, processor, and problem size. For each run we measured the ***cache warm-up fraction*** $(F_P(C) - F_1(C))/2Cs$, which represents the fraction of the cache that needs to be warmed up on each steal. We know from Theorem 17

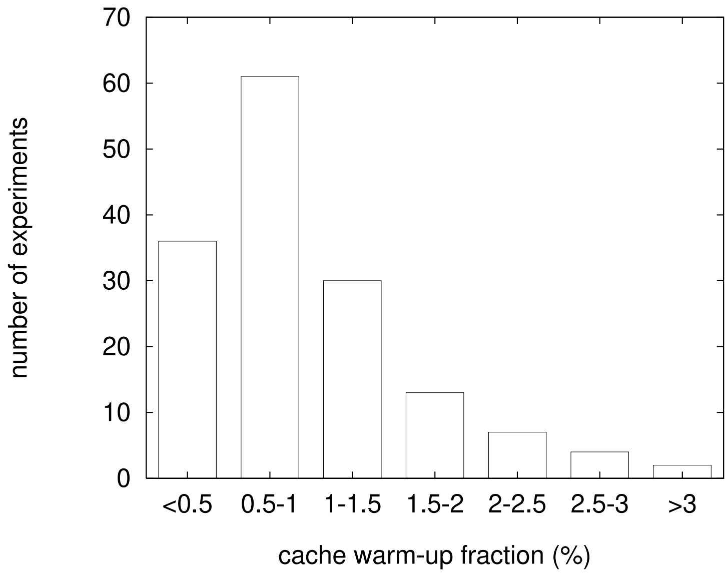


Figure 6-4: Histogram of the cache warm-up fraction $(F_P(C) - F_1(C))/2Cs$ for a variety of applications, cache sizes, processor counts, and problem sizes. The vertical axis shows the number of experiments with a cache warm-up fraction in the shown range.

that the cache warm-up fraction is at most 1. Our experiments indicate that the cache warm-up fraction is, in fact, typically less than 3%, as can be seen from the histogram in Figure 6-4 showing the cache warm-up fraction for 153 experimental runs of the above applications, with processor counts ranging from 2 to 64 and cache sizes from 256KB to 2MB. Thus, we see less than 3% of the extra $2Cs$ faults.

To understand why cache warm-up costs are so low, we performed an experiment that recorded the size of each subproblem stolen. We observed that most of the subproblems stolen during an execution were small. In fact, only 5–10% of the stolen subproblems were "large," where a large subproblem is defined to be one that takes C or more pages to execute. The other 90–95% of the subproblems are small and are stolen when little work is left to do and many of the processors are idle. Therefore, most of the stolen subproblems never perform C page faults before terminating. The bound $F_P(C) \leq F_1(C) + 2Cs$ derived in Theorem 17 thus appears to be rather loose, and our experiments indicate that much better performance can be expected.

6.6 Performance

In this section, we model the performance of Cilk on synthetic benchmark applications similar to `blockedmul`. In order to model performance for Cilk programs 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 computation that results when a given Cilk program 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 observe that the number of instructions with weight $m+1$ is just the number of page faults of the one processor execution, or $F_1(C)$. Thus, we have $T_1(C) = T_1 + mF_1(C)$.

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

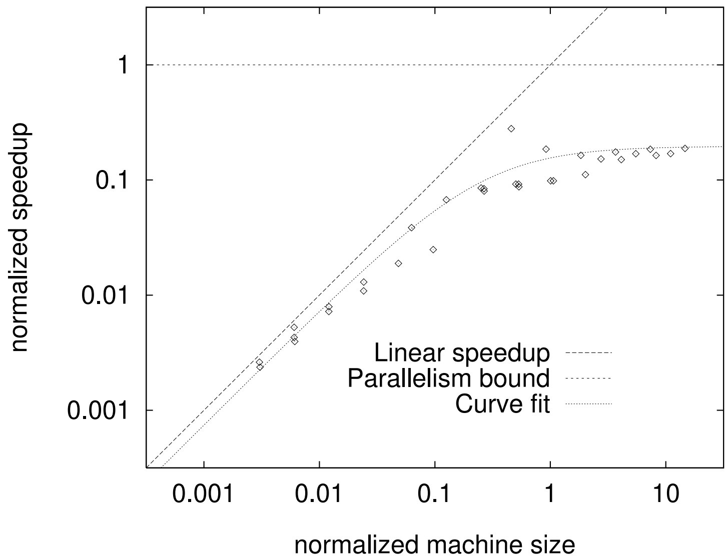


Figure 6-5: Normalized speedup curve for matrix multiplication. The horizontal axis is normalized machine size and the vertical axis is normalized speedup. Experiments consisted of 512×512, 1024×1024, and 2048×2048 problem sizes on 2 to 64 processors, for matrix multiplication algorithms with various critical paths.

computation, the execution time with P processors is much less than $T_1(C)/P$. In this thesis, 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.

The ratio $T_1(C)/T_\infty$ is the ***average parallelism*** of the computation. We found that the running time $T_P(C)$ of the benchmarks on P processors can be estimated as $T_P(C) \approx 1.34(T_1(C)/P) + 5.1(T_\infty)$. Speedup was always at least a third of perfect linear speedup for benchmarks with large average parallelism and running time was always within a factor of 10 of optimal for those without much parallelism.

To analyze Cilk's implementation of the BACKER coherence algorithm, we measured the work and critical-path length for synthetic benchmarks obtained by adding `sync` statements to the matrix multiplication program shown in Figure 4-2. By judiciously placing `sync` statements in the code, we were able to obtain synthetic benchmarks that exhibited a wide range of average parallelism. We ran the benchmarks on various numbers of processors of the CM5, each time recording the number P of processors and the actual runtime $T_P(C)$.[5]

Figure 6-5 shows a normalized speedup curve [15] for the synthetic benchmarks. This curve is obtained by plotting speedup $T_1(C)/T_P(C)$ versus machine size P, but normalizing each of these values by dividing them by the average parallelism $T_1(C)/T_\infty$. We use a normalized speedup curve, because it allows us to plot runs of different benchmarks on the same graph. Also plotted in the figure are the perfect linear-speedup curve $T_P(C) = T_1(C)/P$ (the 45° line) and the limit on performance given by the parallelism bound $T_P(C) \geq T_\infty$ (the horizontal line).

The quantity T_∞ is not necessarily a tight lower bound on $T_P(C)$, because it ignores page faults. Indeed, the structure of `blockedmul` on $n \times n$ matrices causes $\Omega(\lg n)$ faults to be taken along any path through the dag. Although the bound $T_P(C) \geq T_\infty(C)$ is tighter (and makes our numbers look better), it appears difficult to compute. We can estimate using analytical techniques, however, that for our matrix multiplication algorithms, $T_\infty(C)$ is about twice as large as T_∞. Had we used this value for T_∞ in the normalized speedup curve in Figure 6-5, each data point would shift up and right by this factor of 2, giving somewhat tighter results.

The normalized speedup curve in Figure 6-5 shows that dag-consistent shared-memory applications can obtain good speedups. The data was fit to a curve of the form $T_P(C) = c_1 T_1(C)/P + c_\infty T_\infty$. We obtained a fit with $c_1 = 1.34$ and $c_\infty = 5.1$, with an R^2 correlation coefficient of 0.963 and a mean relative error of 13.8%. Thus, the shared memory imposes about a 34% performance penalty on the work of an algorithm, and a factor of 5 performance penalty on the critical path. The factor of 5 on the critical path term is quite good considering all of the scheduling, protocol,

[5]The experiments in this chapter were run using an earlier version of Cilk, Cilk-3, which had explicit shared memory pointers and software page fault checks.

and communication that could potentially contribute to this term.

There are two possible explanations for the additional 34% on the work term. The extra work could represent congestion at the backing store, which causes page faults to cost more than in the one-processor run. Alternatively, it could be because our $T_1(C)$ measure is too conservative. To compute $T_1(C)$, we run the backing store on processors other than the one running the benchmark, while when we run on P processors, we use the same P processors to implement the backing store. We have not yet determined which of these two explanations is correct.

6.7 Conclusion

Many other researchers have investigated distributed shared memory. To conclude, we briefly outline work in this area and offer some ideas for future work.

The notion that independent tasks may have incoherent views of each others' memory is not new to Cilk. The BLAZE [70] language incorporated a memory semantics similar to that of dag consistency into a PASCAL-like language. The Myrias [7] computer was designed to support a relaxed memory semantics similar to dag consistency, with many of the mechanisms implemented in hardware. Loosely-Coherent Memory [64] allows for a range of consistency protocols and uses compiler support to direct their use. Compared with these systems, Cilk provides a multithreaded programming model based on directed acyclic graphs, which leads to a more flexible linguistic expression of operations on shared memory.

Cilk's implementation of dag consistency borrows heavily on the experiences from previous implementations of distributed shared memory. Like Ivy [67] and others [20, 39, 61], Cilk's implementation uses fixed-sized pages to cut down on the overhead of managing shared objects. In contrast, systems that use cache lines [21, 62, 86] require some degree of hardware support [91] to manage shared memory efficiently. As another alternative, systems that use arbitrary-sized objects or regions [22, 59, 88, 90, 97] require either an object-oriented programming model or explicit user management of objects.

The idea of dag-consistent shared memory can be extended to the domain of file I/O to allow multiple threads to read and write the same file in parallel. 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.

Chapter 7

Analysis of dag consistency

In Chapter 6, we proposed dag-consistent distributed shared memory as a shared memory model for multithreaded parallel-programming systems such as Cilk. In this chapter, we analyze the execution time, page faults, and space requirements of Cilk programs where dag consistency is maintained by the BACKER coherence algorithm.[1] We prove that under the assumption that accesses to the backing store are random and independent, Cilk with BACKER executes a program with total work $T_1(C)$ and critical path T_∞ in expected time $O(T_1(C)/P + mCT_\infty)$, where C is the size of the cache in pages and m is the minimum page transfer time. As a corollary to this theorem, we improve upon the bounds in the previous section to prove that the number of page faults $F_P(C)$ is bounded by $F_1(C) + O(CPT_\infty)$.

We also prove bounds on S_P, the space used by a Cilk program on P processors, and $F_1(C)$, the faults of the serial execution, for "regular" divide-and-conquer Cilk programs. We use these bounds to analyze some of the example applications in Chapter 4. For instance, we show that `blockedmul` for $n \times n$ matrices incurs $F_1(C, n) = O(n^3/m^{3/2}\sqrt{C})$ faults and uses $S_P(n) = O(n^2 P^{1/3})$ space.

[1]The contents of this chapter are joint work with Robert Blumofe, Matteo Frigo, Christopher Joerg, and Charles Leiserson and appeared at SPAA'96 [13].

7.1 Introduction

To analyze the performance of Cilk programs which use a shared virtual address space implemented by BACKER, we must take into account all of the protocol actions required by BACKER. The BACKER algorithm implements this virtual space by cacheing physical pages from a backing store which is distributed across the processors. We assume that when a page fault (cache miss) 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) [25] 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.

Recall from Section 2.8 that both T_1/P and T_∞ are lower bounds on the running time of any computation. The randomized work-stealing scheduler used by Cilk achieves performance close to these lower bounds for the case of Cilk programs that do not use shared memory. Specifically, for any such program and any number P of processors, the scheduler executes the program in $T_1/P + O(T_\infty)$ expected time.

To analyze the complete system, however, we must include the overhead costs of BACKER as well. As in Section 6.6, we assume a Cilk program 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. We define the measures $F_1(C)$, $T_1(C)$, $T_P(C)$, and $T_\infty(C)$ as in that section. 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, for any given input problem, the expected execution time $T_P(C)$ is $O(T_1(C)/P + mCT_\infty)$. 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, procedures may concurrently access the same pages by program design. We can artificially solve this problem by insisting, as does the EREW-PRAM model, that the program 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 had assumed, for example, that accesses to the backing store independently suffer Poisson-distributed delays.

In this chapter, we also analyze the number of page faults that occur during program execution. Under the same assumptions, we show that for any Cilk program, the expected number of page faults to execute the program on P processors, each with an LRU cache of size C, is at most $F_1(C) + O(CPT_\infty)$. In addition, for "regular" divide-and-conquer Cilk programs, we derive a good upper bound on $F_1(C)$ in terms of the input size of the problem. For example, we show that the total number of page faults incurred by the divide-and-conquer matrix-multiplication algorithm `blockedmul` 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 chapter, we analyze the space requirements of "simple" Cilk programs that use dag-consistent shared memory. For a given simple Cilk program, let S_1 denote the space required by the standard, depth-first serial execution of the program to solve a given problem. In an analysis of the Cilk scheduler, Blumofe and Leiserson have shown that the space used by a P-processor execution is at most S_1P in the worst case [11, 16]. 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 Cilk programs. For example, we show that the `blockedmul` program on P processors uses only $O(n^2P^{1/3})$ space when multiplying $n \times n$ matrices, which is tighter than the $O(n^2P)$ result obtained by directly applying the S_1P bound.

The remainder of this chapter is organized as follows. Section 7.2 analyzes the execution time of Cilk programs using the BACKER coherence algorithm. Section 7.3 analyzes the number of page faults taken by divide-and-conquer Cilk programs, and Section 7.4 does the same for space requirements. Section 7.5 presents some sample analyses of algorithms that use dag-consistent shared memory. Finally, Section 7.6 offers some comparisons with other consistency models and some ideas for the future.

7.2 Analysis of execution time

In this section, we bound the execution time Cilk programs when dag consistency is maintained by the BACKER algorithm, under the assumption that accesses to the backing store are random and independent. For a given Cilk program, let $T_P(C)$ denote the time taken by the program to solve a given problem on a parallel computer with P processors, each with an LRU cache of C pages, when the execution is scheduled by Cilk in conjunction with the BACKER coherence algorithm. In this section, we show that if accesses to backing store are random and independent, then the expected value of $T_P(C)$ is $O(T_1(C)/P + mCT_\infty)$, where m denotes the minimum time to transfer a page and T_∞ is the critical-path length of the computation. 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 [11, 16].

In the following analysis, we consider the computation that results when a given Cilk program is executed to solve a given input problem. We assume that the computation is executed by Cilk's work-stealing scheduler in conjunction with the BACKER coherence algorithm on a parallel computer with P homogeneous 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. We assume that a minimum of m time steps are required to transfer a page. 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.

The work-stealing scheduler assumed in our analysis is the same work-stealing scheduler used in Chapter 3, but with a small technical modification. Between successful steals, we wish to guarantee that a processor performs at least C page transfers (fetches or reconciles) so that it does not steal too often. Consequently, whenever a processor runs out of work, if it has not performed C page transfers since its last successful steal, the modified work-stealing scheduler 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.

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.

Before embarking on the analysis, however, we first define some helpful 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 18 *Each primary transfer during an execution can be associated with a currently running primary subcomputation such that each primary subcomputation has at most* $3C$ *associated primary transfers. Similarly, each secondary transfer during an execution can be associated with a currently running secondary subcomputation such that each secondary subcomputation has at most* $3C$ *associated secondary transfers.*

Proof: For this proof, we use the fact shown in Section 6.6 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 19 *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 $\epsilon > 0$, with probability at least $1 - \epsilon$, the total number of steal requests and primary transfers is at most $O(CPT_\infty + CP \lg(1/\epsilon))$.*

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 $1 - \epsilon$, an execution contains only $O(T_\infty + \lg(1/\epsilon))$ rounds.

To bound the number of rounds, we shall use a delay-sequence argument. We define a modified dag D' exactly as in [16]. (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 18, 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 [16, 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 want to show that with probability at least $1 - \epsilon$, the total number of rounds is $O(T_\infty + \lg(1/\epsilon))$. Consider a possible delay sequence. Recall from [16] 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

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

which can be made less than ϵ by choosing $R = 14T_\infty + \lg(1/\epsilon)$. Therefore, there are at most $O(T_\infty + \lg(1/\epsilon))$ rounds with probability at least $1-\epsilon$. In each round, there are at most $28CP$ events, so there are at most $O(CPT_\infty + CP\lg(1/\epsilon))$ 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 $X \le \sum_{u \in U}(5mC + m\sum_{ip} x_{uip})$. Rearranging, we have $X \le 10mCT_\infty + m\sum_{uip} x_{uip}$, 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:

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

This probability can be made less than $(1/2)^z$ by choosing $z \ge 20eCT_\infty$. 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 $1 - \epsilon$, there are $O(T_\infty + \lg(1/\epsilon))$ rounds, and hence $O(CPT_\infty + CP\lg(1/\epsilon))$ events, during the execution. ■

We now bound the running time of a computation.

Theorem 20 *Consider any Cilk program executed on P processors, each with an LRU cache of C pages, using Cilk's 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 $\epsilon > 0$, the execution time is $O(T_1(C)/P + mCT_\infty + m\lg P + mC\lg(1/\epsilon))$ with probability at least $1 - \epsilon$. Moreover, the expected execution time is $O(T_1(C)/P + mCT_\infty)$.*

Proof: As in [16], 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 now bound the amount of money in each of the buckets at the end of the computation by using the fact, from Lemma 19, that with probability at least $1 - \epsilon'$, there are $O(CPT_\infty + CP\lg(1/\epsilon'))$ 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 $O(CPT_\infty + CP\lg(1/\epsilon'))$ steal requests, so there are $O(CPT_\infty + CP\lg(1/\epsilon'))$ dollars in the STEAL bucket.

StealWait. We use the analysis of the ***recycling game*** ([16, 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 $O(N+P\lg P+P\lg(1/\epsilon'))$ with probability at least $1-\epsilon'$. Since steal requests obey the assumptions of the recycling game, if there are $O(CPT_\infty+CP\lg(1/\epsilon'))$ steals, then the total time waiting for steal requests is $O(CPT_\infty+P\lg P+CP\lg(1/\epsilon'))$ with probability at least $1-\epsilon'$. We must add to this total an extra $O(mCPT_\infty+mCP\lg(1/\epsilon'))$ 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 $O(CPT_\infty + CP \lg(1/\epsilon'))$ such reconciles. Finally, we must add $O(mCPT_\infty + mCP \lg(1/\epsilon))$ dollars because each steal request might also have up to m idle steps associated with it. Thus, with probability at least $1 - \epsilon'$, we have a total of $O(mCPT_\infty + P \lg P + mCP \lg(1/\epsilon'))$ dollars in the StealWait bucket.

Xfer. We know that there are $O(F_1(C) + CPT_\infty + CP \lg(1/\epsilon'))$ transfers during the execution: a fetch and a reconcile for each intrinsic fault, $O(CPT_\infty + CP \lg(1/\epsilon'))$ primary transfers from Lemma 19, and $O(CPT_\infty + CP \lg(1/\epsilon'))$ 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 procedure in the spawn tree. 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 procedure just after k is spawned. If not, then k is completed before j's procedure 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 $O(CPT_\infty + CP \lg(1/\epsilon'))$ secondary transfers. Since each transfer takes m time, the number of dollars in the Xfer bucket is $O(mF_1(C) + mCPT_\infty + mCP \lg(1/\epsilon'))$.

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

$O(mF_1(C) + mCPT_\infty + mP \lg P + mCP \lg(1/\epsilon'))$ dollars in the XFERWAIT bucket with probability at least $1 - \epsilon'$.

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

We now bound the number of page faults.

Corollary 21 *Consider any Cilk program executed on P processors, each with an LRU cache of C pages, using Cilk's 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 $\epsilon > 0$, the number of page faults is at most $F_1(C)+O(CPT_\infty+CP \lg(1/\epsilon))$ with probability at least $1-\epsilon$. 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 $O(CPT_\infty + CP \lg(1/\epsilon))$ primary and secondary faults. The expected bound follows similarly. ∎

7.3 Analysis of page faults

This section provides upper bounds on the number of page faults for "regular" divide-and-conquer Cilk programs when the parallel execution is scheduled by Cilk and dag consistency is maintained by the BACKER algorithm. In a ***regular divide-and-conquer*** Cilk program, each procedure, when spawned to solve a problem of size n, operates as follows. If n is larger than some given constant, the procedure divides the problem into a subproblems, each of size n/b for some constants $a \geq 1$ and $b > 1$, and then it recursively spawns child procedures to solve each subproblem. When all a of the children have completed, the procedure merges their results, and then returns. In the base case, when n is smaller than the specified constant, the procedure directly solves the problem, and then returns.

Corollary 21 bounds the number of page faults that a Cilk program incurs when run on P processors using Cilk's scheduler and the BACKER coherence algorithm. Specifically, for a given Cilk program, 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 an LRU cache of C pages. In addition, for any number $P \geq 2$ 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 Cilk's scheduler and BACKER on P processors, each with an LRU cache of C pages. Corollary 21 then says that the expectation of $F_P(C, n)$ is at most $F_1(C, n) + O(CPT_\infty(n))$, where $T_\infty(n)$ is the critical path of the computation on a problem of size 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 Cilk programs, however, analysis can provide good asymptotic bounds on $F_1(C, n)$, and hence on $F_P(C, n)$.

Theorem 22 *Consider any regular divide-and-conquer Cilk program executed on 1 processor with an 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 procedure, 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 $a \geq 1$ and $b > 1$. Additionally, suppose each procedure 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]

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

1. *If* $p(n) = O(n^{\log_b a - \epsilon})$ *for some constant* $\epsilon > 0$*, then* $F_1(C, n) = O(C(n/n_C)^{\log_b a})$*, if* $p(n)$ *further satisfies the regularity condition that* $p(n) \leq a\gamma p(n/b)$ *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* $p(n) = \Omega(n^{\log_b a + \epsilon})$ *for some constant* $\epsilon > 0$*, then* $F_1(C, n) = O(C(n/n_C)^{\log_b a} + p(n))$*, if* $p(n)$ *further satisfies the regularity condition that* $p(n) \geq a\gamma p(n/b)$ *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 procedure 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

$$F_1(C, n) \leq \begin{cases} aF_1(C, n/b) + p(n) & \text{if } n > n_C \ , \\ C & \text{if } n \leq n_C \ . \end{cases} \tag{7.1}$$

We can solve this recurrence using standard techniques [26, Section 4.4]. We iterate the recurrence, stopping as soon as we reach the first value of the iteration count k such that $n/b^k \leq n_C$ holds, or equivalently when $k = \lceil \log_b(n/n_C) \rceil$ holds. Thus, we have

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

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 $p(n_C) \leq C$, we obtain the stated results. ∎

7.4 Analysis of space utilization

This section provides upper bounds on the memory requirements of regular divide-and-conquer Cilk programs when the parallel execution is scheduled by a "busy-leaves" scheduler, such as the Cilk scheduler. A ***busy-leaves*** scheduler is a scheduler with the property that at all times during the execution, if a procedure has no living children, then that procedure has a processor working on it. Cilk's work-stealing scheduler is a busy-leaves scheduler [11, 16]. We shall proceed through a series of lemmas that provide an exact characterization of the space used by "simple" Cilk programs when executed by a busy-leaves scheduler. A ***simple*** Cilk program is a program in which each procedure'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 Cilk programs.

Previous work [11, 16] has shown that a busy-leaves scheduler can efficiently execute a Cilk program on P processors using no more space than P times the space required to execute the program on a single processor. Specifically, for a given Cilk program, if S_1 denotes the space used by the program to solve a given problem with the standard, depth-first, serial execution order, then for any number P of processors, a busy leaves scheduler uses at most PS_1 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. The bound then follows, because each of the at most P leaves living at any time is responsible for at most S_1 space, for a total of PS_1 space. For many programs, however, the bound PS_1 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 programs. We start by considering the more general case of simple Cilk programs.

We first introduce some terminology. Consider any simple Cilk program and input problem, and let T be the spawn tree of the program 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 (procedure) $u \in T$ 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 Cilk programs when executed by a busy leaves scheduler.

Lemma 23 *Let T be the spawn tree of a simple Cilk program, 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. ∎

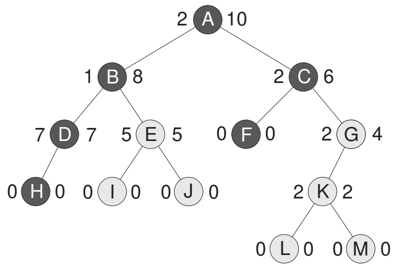


Figure 7-1: An illustration of the definition of a dominator set. For the tree shown, let f be given by the labels at the left of the nodes, and let $\Lambda = \{F, H\}$. Then, the serial space S is given by the labels at the right of the nodes, $\mathcal{C}(\Lambda) = \{A, B, C, D, F, H\}$ (the shaded nodes), and $\mathcal{D}(\Lambda, G) = \{C, D\}$. The space required by Λ is $\mathcal{S}(\Lambda) = 12$.

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 Cilk program, 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:

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

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

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

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

Lemma 24 *Let T be the spawn tree of a simple Cilk program 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 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 25 *Let T be the spawn tree of a simple Cilk program 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 24 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 Cilk programs, as introduced in Section 7.3. In a regular divide-and-conquer Cilk program, each procedure, 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 26 *Let T be the spawn tree of a regular divide-and-conquer Cilk program 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 25 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 Cilk program when it is scheduled using a busy-leaves scheduler.

Theorem 27 *Consider any regular divide-and-conquer Cilk program executed on P processors using a busy-leaves scheduler. Suppose that each procedure, when spawned to solve a problem of size n, allocates $s(n)$ space, and if n is larger than some constant,* *then the procedure 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 $s(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $S_P(n) = \Theta(Ps(n/P^{1/\log_b a}))$, 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 $s(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, 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 Cilk program that results when the program 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 23 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 26 which characterizes the set Λ at which this maximum occurs. Lemma 26 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

$$S_P(n) \leq \sum_{i=0}^{\lfloor \log_a P \rfloor - 1} a^i s(n/b^i) + \Theta(PS_1(n/P^{1/\log_b a})) \ . \tag{7.2}$$

To determine which term in Equation (7.2) dominates, we must evaluate $S_1(n)$,

[3]Other cases exist besides those given here.

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 [26, 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
- $S_1(n) = \Theta(s(n))$, if $s(n) = \Omega(n^\epsilon)$ for some constant $\epsilon > 0$ and in addition satisfies the regularity condition that $s(n) \geq \gamma s(n/b)$ for some constant $\gamma > 1$.

The theorem follows by evaluating Equation (7.2) 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. ∎

7.5 Example analyses of Cilk programs

In this section we show how to apply the analysis techniques of this chapter to specific Cilk programs. We focus first on analyzing matrix multiplication, and then we examine LU-decomposition. We show that the algorithms given for these problems in Section 4.1 are efficient with respect to the measures of time, page faults, and space. In our analyses, we shall assume that the cache memory of each of the P processors contains C pages and that each page holds 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 $T_P(C) = O(T_1(C)/P + mCT_\infty)$ and $F_P(C) = F_1(C) + O(CPT_\infty)$ are good models for the performance of Cilk programs.

Let us first apply our results to the naive "blocked" serial matrix multiplication algorithm for computing $R = AB$ in which the three matrices A, B, and R 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 R, j through the column blocks of R, and k through the row blocks of A and column blocks of B—updating $R[i,j] \leftarrow R[i,j] + A[i,k] \cdot B[k,j]$ on the matrix blocks. This algorithm can be parallelized to obtain computational work $T_1(n) = \Theta(n^3)$ and critical-path length $T_\infty(n) = \Theta(\lg n)$ [65]. If the matrix B does not fit into the cache, that is, $mC < n^2$, then in the serial execution, 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 R never fault.

The divide-and-conquer `blockedmul` algorithm from Section 4.1.1 uses the processor cache much more effectively. To see why, we can apply Theorem 22 to analyze the page faults of `blockedmul` using $a = 8$, $b = 2$, $n_C = \sqrt{mC}$, and $p(n) = \Theta(n^2/m)$. Case 1 of the theorem applies with $\epsilon = 1$, 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 `blockedmul`, we use Theorem 27. 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 $S_P(n) = \Theta(P(n/P^{1/3})^2) = \Theta(n^2 P^{1/3})$.[4] Note that this space bound is better than the $O(n^2 P)$ bound obtained by just using the $O(S_1 P)$ bound from [11, 16].

We have already computed the computational work and critical path length of the `blockedmul` algorithm in Section 4.1.1. Using these values we can compute the total work and estimate the total running time $T_P(C,n)$. The computational work of `blockedmul` is $T_1(n) = \Theta(n^3)$, so the total work is $T_1(C,n) = T_1(n) + mF_1(C,n) = \Theta(n^3)$. The critical path is $T_\infty = \Theta(\lg^2 n)$, so using our performance model, the

[4] In recent work, Blelloch, Gibbons, and Matias [10] 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.

total expected time for `blockedmul` on P processors is $T_P(C,n) = O(T_1(C,n)/P + mCT_\infty(n)) = O(n^3/P + mC \lg^2 n)$. Consequently, if we have $P = O(n^3/(mC \lg^2 n))$, the algorithm runs in $O(n^3/P)$ time, obtaining linear speedup. 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 `blockedmul` 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 the behavior of algorithms.

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.

The divide-and-conquer algorithm presented in Section 4.1.2 for LU decomposition incurrs fewer page faults, at the cost of a slightly longer critical path. To bound the number of page faults, we first bound the page faults during the back substitution step. Observe that page faults in the one step of back substitution are dominated by the $\Theta(n^3/m^{3/2}\sqrt{C})$ page faults in the matrix multiplication, and hence we obtain the recurrence $F_1(C,n) = 4F_1(n/2) + \Theta(n^3/m^{3/2}\sqrt{C})$. Therefore, we can apply Case 3 of Theorem 22 with $a = 4$, $b = 2$, $n_C = \sqrt{mC}$, and $p(n) = O(n^3/m^{3/2}\sqrt{C})$ to obtain the solution $F_1(C,n) = \Theta(n^3/m^{3/2}\sqrt{C})$.

We now analyze the page faults of the LU-decomposition algorithm as a whole. The number of serial page faults satisfies $F_1(C,n) = 2F_1(C,n/2) + \Theta(n^3/m^{3/2}\sqrt{C})$, due to the matrix multiplications and back substitution costs, which by Case 3 of Theorem 22 with $a = 2$, $b = 2$, $n_C = \sqrt{mC}$, and $p(n) = O(n^3/m^{3/2}\sqrt{C})$ has solution $F_1(C, n) = \Theta(n^3/m^{3/2}\sqrt{C})$.

Using our performance model, the total expected time for LU-decomposition on P processors is therefore $T_P(C, n) = O(T_1(C, n)/P + mCT_\infty(n)) = O(n^3/P + mCn \lg^2 n)$. If we have $P = O(n^3/mCn \lg^2 n)$, the algorithm runs in $O(n^3/P)$ time, obtaining linear speedup. As with `blockedmul`, many fewer page faults occur for the divide-and-conquer algorithm for LU-decomposition than for the corresponding standard algorithm. The penalty we pay is a slightly longer critical path—$\Theta(n \lg^2 n)$ versus $\Theta(n)$—which decreases the available parallelism. The critical path can be shortened to $\Theta(n \lg n)$ by using the more space-intensive `blockedmul` algorithm during back and forward substitution, however.

We leave it as an open question whether Cilk programs with optimal critical paths can be obtained for matrix multiplication and LU-decomposition without compromising the other performance parameters.

7.6 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 [63] and various forms of processor consistency [47] are too expensive to implement in a distributed setting. (Even modern SMP's do not typically implement sequential consistency.) Relaxed models, such as Gao and Sarkar's location consistency [43] (not the same as Frigo's location consistency [40]) and various forms of release consistency [1, 33, 44], 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 [20, 59, 61, 90], though some implement a fixed collection of consistency models [8], while others merely implement a collection of mechanisms on top of which users write their own DSM consistency policies [64, 86]. 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, Backer in conjunction with Cilk's work-stealing scheduler is able to execute Cilk programs with the performance bounds shown here. The BLAZE parallel language [70] and the Myrias parallel computer [7] 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 also currently working on supporting dag-consistent shared memory in our Cilk-NOW runtime system [11] 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 several analytical questions regarding the performance of Cilk programs 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 20 holds when pages are hashed to backing store provided the algorithm is EREW in the sense that concurrent procedures 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 Cilk programs that are different from, or more general than, the class of regular divide-and-conquer programs analyzed here.

Chapter 8

Distributed Cilk

8.1 Introduction

This chapter describes our implementation of Cilk on a cluster of SMP's. In particular, we define MultiBacker, an extension of the Backer algorithm from Chapter 6 which takes advantage of hardware support for sharing within an SMP while maintaining the advantages of Backer across SMP's. Also, we give our "local bias" scheduling policy for modifying the Cilk scheduler to improve the locality of scheduling decisions without breaking the provable properties of Backer given in Chapter 7. With these two modifications to Backer and the Cilk scheduler, we show that Cilk programs achieve good speedups on networks of SMP's.

8.2 Multilevel shared memory

In this section, we describe modifications to the Backer algorithm from Chapter 6 to operate on clusters of SMP's. The original Backer algorithm is designed for a network of uniprocessors, that is, a network where each processor has its own shared-memory cache. In a cluster of SMP's, however, multiple processors on an SMP have the opportunity to share a single shared-memory cache. The design of MultiBacker, our multilevel shared memory protocol, attempts to share a shared-memory cache among several processors on an SMP and provide hardware-based memory consistency between those processors.

One obvious solution to implementing BACKER on a cluster of SMP's is to give each processor its own separate shared-memory cache. We call this solution ***disjoint*** BACKER. Disjoint BACKER ignores the fact that the processors in an SMP are connected by hardware support for shared memory. Thus, it has two inefficiencies. The first is that if a particular shared-memory page is referenced by more than one processor, multiple copies of the page are stored in one SMP, one in each cache of each referencing processor. These multiple copies artificially reduce the amount of shared memory that can be cached in the case when the processors on an SMP exhibit some commonality of reference, which we expect to be the case. Second, by keeping the shared-memory caches separate for each processor, we lose the opportunity to use the hardware consistency mechanism provided by the SMP. Because references to the same virtual page by two different processors go to two different physical pages, no hardware sharing is possible between processors.

Another solution is to treat the SMP as a single processor within the BACKER algorithm. We call this solution ***unified*** BACKER. The SMP has a single unified cache for all processors on the SMP. The processors of the SMP operate on the cache just as if they were executing as timeslices of a single processor. Most distributed shared memory protocols use this model for their SMP caches [61, 35], because it does not cache redundant pages and allows hardware sharing within an SMP.

Unified BACKER does have a drawback, however. In particular, when one processor requires a consistency operation to be performed on a page, it must be performed with respect to all processors. For instance, if a processor needs to invalidate a page because of a consistency operation, it must be invalidated by all processors on the SMP. Erlichson et al [35] call this the TLB synchronization problem, because invalidations are done by removing entries from a page table, and cached page table entries in the TLB must be removed as well. Invalidating TLB entries on multiple processors requires some form of interprocessor synchronization and can be expensive. Erlichson et al found that TLB synchronization on a 12-processor SGI machine takes over 350μs.

For many parallel programming environments, in particular those with frequent global barriers, requiring TLB synchronization is not too cumbersome because when one processor is required to invalidate a page, all processors are required to invalidate that page. In BACKER, however, most synchronization operations are bilateral in nature, from one processor on one SMP to one processor on another SMP. This limited synchronization means that processors on one SMP do not necessarily need to invalidate pages simultaneously. Therefore, forcing all processors to invalidate a page that only one processor needs to invalidate can lead to high TLB synchronization overheads. Additionally, processors who must unnecessarily invalidate a page may be forced to refetch that page and thus generate additional memory traffic.

The third alternative, called MULTIBACKER, is used in our implementation of distributed Cilk. In the MULTIBACKER protocol, we keep one central cache for each SMP, but control the access to each page of the cache on a per-processor basis. This solution has all of the advantages of unified BACKER. In particular, no multiple copies of a page are kept and sharing within a machine is provided in hardware. However, each processor keeps a separate set of access permissions (in practice, a separate virtual memory map) that it can update individually. No processor on an SMP needs to synchronize its TLB with any other processor, except when pages need to be removed from the cache because of capacity constraints. In this case, however, the TLB synchronization overhead can be amortized by evicting several pages from the cache at once. By maintaining separate access permissions, processors only need to invalidate, and consequently refetch, pages which actually require invalidation.

The flexibility of MULTIBACKER requires a slightly more complicated protocol than either disjoint or unified BACKER, however. Because some processors may have a page mapped invalid while others have it mapped valid, a page fetch cannot simply copy a page into the cache, because pages with dirty entries cannot be overwritten. Instead, we use a procedure called ***two-way diffing***, described in [95] as "outgoing" and "incoming" diffs. Just as with the BACKER algorithm described in Section 6.4, an ***outgoing*** diff compares the working copy of a page with its twin, and sends any data that have been written since the twin was created to the backing store. An

incoming diff, on the other hand, compares a page fetched from the backing store with the twin, and writes any changes that have occurred at the backing store since the twin was fetched to the working copy (and to the twin itself). In this way, changes can be propegated both from a processor cache to the backing store via outgoing diffs, and from the backing store to a processor cache via incoming diffs.

The MultiBacker protocol also needs to maintain timestamps to keep track of which processors are allowed to access which pages. Each processor has a ***permission number*** which identifies how recently it has performed an off-SMP synchronization operation. Similarly, each page in the cache has a ***timestamp*** which records when the page was last updated from the backing store. The invariant maintained by MultiBacker is that a processor may access a page in the cache if its permission number is smaller than the timestamp of the page. When a page is fetched from backing store, its timestamp is set to be greater than the permission numbers of all processes in the SMP, giving all processors on the SMP permission to access the up-to-date page. When a processor does an off-SMP synchronization operation, its permission number is set to be greater than the timestamp of any page in the cache, thereby invalidating all of that processor's pages. When processors synchronize within an SMP, they set their permission numbers to the maximum permission number of the synchronizing processors, ensuring that if the page is out-of-date with respect to any synchronizing processor, it becomes out of date with respect to all synchronizing processors.

8.3 Distributed scheduling

The work-stealing scheduler presented in Section 3.2 makes no distinction between stealing from a processor on the same SMP or a processor on a different SMP. As a practical matter, however, we would like to schedule related threads on the same SMP whenever possible. This section outlines our strategy for providing some measure of locality of scheduling while at the same time retaining the provably-good properties of the work-stealing scheduler from Chapter 7.

The obvious way to increase the locality of scheduling is to always steal from a processor on the local SMP if there is any work on the SMP. Only when the whole SMP runs out of work are any steal attempts allowed to go to another SMP. Although this strategy, which we call the ***maximally local*** strategy, provides good locality, it breaks the provable bounds of the Cilk scheduler. In particular, the following scenario demonstrates how the maximally local scheduling algorithm can go wrong. Let our cluster consist of M SMP's, each with P processors. The first $M-1$ SMP's have a single, long-running, nearly serial piece of work. The last SMP has a large chunk of work with lots of parallelism. In this situation, because no steal attempts leave an SMP, the $M-1$ machines never realize that the last SMP has a lot of work that they can steal and work on. Thus, this configuration executes only $M-1+P$ instructions at every time step when there is the potential to execute MP instructions per time step. For a large cluster ($M \gg P$), this configuration forces such a maximally local scheduler to use only one processor on every machine.

How can we modify the Cilk scheduler to improve scheduling locality without introducing such performance anomalies? Our solution lies examining the proof of the optimality of the Cilk scheduler given in Chapter 7. If we modify the scheduling algorithm in such a way that the proof is preserved, then we are guaranteed that no such performance anomalies can occur.

We modify the scheduling algorithm as follows. We use a ***local bias*** strategy where instead of stealing randomly with a uniform distribution, we steal with a biased distribution. We bias the choice of the victim processor to be weighted in favor of the local processors. Let α be the ratio of the probability of picking a local processor versus the probability of picking a remote processor. We observe that the running time bound obtained in Chapter 7 depends linearly on the minimum probability of any processor being chosen as a steal target. Since the local bias strategy underweights the remote processors by a factor of at most $O(\alpha)$, the running time bounds increase by a factor of $O(\alpha)$. As long as we choose α to be a constant, instead of ∞ as was done with the maximally local strategy, we maintain provable performance (albeit with a larger constant hidden in the big-Oh) while augmenting the locality of Cilk's scheduler.

Now that we have identified the local bias strategy as the proper modification to make to Cilk's stealing algorithm, we need to decide on the scheduling parameter α. A natural choice for α is the largest α possible while still maintaining a remote stealing rate close to the rate obtained when $\alpha = 1$. Maintaining the remote stealing rate is important for distributing work quickly at the beginning of the computation or after a serial phase. This value of α is just the ratio of the latency of a remote steal to the latency of a local steal, because if α local steals are performed per remote steal, the rate of remote stealing drops by only a factor of 2. (Larger values of α lower the remote stealing rate proportionally to the increase in α.) For our implementation on a cluster of 466MHz Alpha SMP's connected with Digital's Memory Channel, this choice of α is about 100, derived from a local steal latency of about $0.5\mu s$ and a remote steal latency of about $50\mu s$.

8.4 Distributed Cilk performance

We have a preliminary version of distributed Cilk running with MultiBacker and our local bias scheduling strategy on a cluster of SMP's. Distributed Cilk runs on multiple platforms and networks. There are currently implementations for Sun Ultra 5000 SMP's, Digital Alpha 4100 SMP's, and Penium Pro SMP's. Currently supported networks include UDP over Ethernet, Digital's Memory Channel, and MIT's Arctic network [18, 56]. This section gives some preliminary performance numbers for our Alpha 4100 SMP implementation running with the Memory Channel interconnect.

Figure 8-1 shows the performance of the Fibonacci program from Figure 2-1. Importantly, no changes were made to the source code of the program to run it on distributed Cilk. Except for the presence of a relaxed consistency model,[1] the programming environment is the same as Cilk on one SMP. We can see from Figure 8-1 that for a simple program like Fibonacci, near-linear speedup is achievable even when dynamically load balancing across a cluster of SMP's.

[1] And, as a consequence, the absence of locking primitives

processors	configuration	speedup
1	1	1.00
2	2	2.00
2	1,1	1.99
3	2,1	2.98
3	1,1,1	2.97
4	2,2	3.97
4	2,1,1	3.93
5	2,2,1	4.93
6	2,2,2	5.87

Figure 8-1: Performance of Fibonacci program from Figure 2-1 for various machine configurations. These experiments were run on a cluster of 3 Alpha 4100 machines with 4 466MHz processors each connected with Digital's Memory Channel network. The configuration column shows how many processors on each machine were used for computation. An additional processor on each machine (not included in the above numbers) was dedicated to polling.

We are currently working on evaluating the performance of our cluster of SMP version of distributed Cilk on programs with more demanding shared-memory requirements.

Chapter 9

Conclusion

This thesis presents a parallel multithreaded language called Cilk. I believe that Cilk can go a long way to provide a robust programming environment for contemporary SMP machines. Here is a summary of Cilk's features and how each one simplifies the life of a programmer wishing to take advantage of his new commodity SMP (or SMP's):

- **simple language semantics** By providing a simple, faithful, parallel extension of C, Cilk makes the transition from serial C programming to parallel Cilk programming easier. Also, by leveraging the power of modern C compilers, Cilk retains all of the efficiencies of serial C programming.

- **low parallel overhead** Historically, a common obstacle to programming in parallel has been that parallel programs are not efficient to run on one, or even a few, processors. Only with very large parallel computers was the effort of programming in parallel worthwhile. Because the parallel overhead of Cilk is so small, however, programmers can now program for very small SMP's, only a few processors, and expect to get good performance. In fact, the overhead of Cilk is so small that it makes sense to use the Cilk program even when running on a uniprocessor. That way, a programmer needs to maintain only a single code base, simplifying program development considerably.

- **automatic load balancing** Cilk's provably efficient work-stealing scheduler alleviates the programmer from having to worry about which part of his program executes on which processor. The programmer needs only to *express* the parallelism in his application, the Cilk system does the rest.

- **good speedup** The efficiency of the Cilk language together with the efficiency of its scheduler combine to give good overall application speedups when a serial program is converted to a Cilk program. A programmer can be fairly confident that if he can expose enough parallelism in his code using Cilk, the system will be able to exploit that parallelism to obtain good speedups.

- **parallel debugging** Programming in parallel is difficult because reasoning about all possible behaviors of a program can become intellectually intractable. Cilk provides a debugging tool called the Nondeterminator-2 that can detect many of the misbehaviors of parallel programs quickly and robustly. This tool greatly simplifies the intellectual effort required to design and debug a parallel program.

- **distributed implementation** Because a Cilk program can be easily ported to a cluster of SMP's, a programmer can convieniently scale from a small SMP to a large cluster of SMP's without having to rewrite his application. Thus, the concept of the single code base extends even to very large, distributed memory computers.

Appendix A

Connecting deadlock-free programs and computations

In this appendix, we prove Lemma 14, which shows that a deadlock in a data-race free computation of an abelian program always corresponds to a deadlock in the program.[1]

In our current formulation, proving that a deadlock scheduling of a computation is true is not sufficient to show that the machine actually deadlocks. A deadlock scheduling is one that cannot be extended in the computation, but it may be possible for the machine to extend the execution if the next machine instruction does not correspond to one of the possibilities in the dag. In this appendix, in order to prove machine deadlocks, we think of a LOCK instruction as being composed of two instructions: LOCK_ATTEMPT and LOCK_SUCCEED. Every two LOCK_SUCCEED instantiations that acquire the same lock must be separated by an UNLOCK of that lock, but multiple interpreters can execute LOCK_ATTEMPT instantiations for the same lock without an intervening UNLOCK. In other words, LOCK_ATTEMPT instructions can always be executed by the interpreter, but LOCK_SUCCEED instructions cannot be executed unless no other interpreter holds the lock. If an interpreter executes a LOCK_ATTEMPT instruction, the next instruction executed by the interpreter must be a LOCK_SUCCEED instruction for the same lock. A true deadlock scheduling is therefore an actual ma-

[1]The contents of this appendix are joint work with Andrew Stark.

chine deadlock, because the LOCK_SUCCEED instantiations that come next in the dag are always the same as the next possible instantiations for the machine.

A LOCK_ATTEMPT instantiation commutes with any other parallel instantiation. For convenience, we still use the instantiation LOCK to mean the sequence LOCK_ATTEMPT LOCK_SUCCEED.

To prove Lemma 14, we first introduce new versions of Lemmas 10, 11, and 12 that assume a deadlock-free program instead of a deadlock-free dag. We then prove Lemma 14.

Lemma 28 (Reordering) *Let G be a data-race free computation resulting from the execution of a deadlock-free abelian program, and let R_1 and R_2 be two parallel regions in G. Then:*

1. *Let X be a partial scheduling of G of the form $X_1 R_1 R_2 X_2$. The partial scheduling X and the partial scheduling $X_1 R_2 R_1 X_2$ are equivalent.*
2. *Let Y be a true partial scheduling of G of the form $Y = Y_1 R_1 R_2'$, where R_2' is a prefix of R_2. Then then the partial scheduling $Y_1 R_2'$ is true.*

Proof: We prove the lemma by double induction on the nesting count of the regions. Our inductive hypothesis is the theorem as stated for regions R_1 of nesting count i and regions R_2 of nesting count j. The proofs for part 1 and part 2 are similar, so sometimes we will prove part 1 and provide the modifications needed for part 2 in parentheses.

Base case: $i = 0$. Then R_1 is a single instantiation. Since R_1 and R_2 (R_2') are parallel and are adjacent in X (Y), no instantiation of R_2 (R_2') can be guarded by a lock that guards R_1, because any lock held at R_1 is not released until after R_2 (R_2'). Therefore, since G is data-race free, either R_1 and R_2 (R_2') access different memory locations or R_1 is a READ and R_2 (R_2') does not write to the location read by R_1. In either case, the instantiations of each of R_1 and R_2 (R_2') do not affect the behavior of the other, so they can be executed in either order without affecting the final memory state.

Base case: $j = 0$. Symmetric with above.

Inductive step: In general, R_1 of count $i \geq 1$ has the form LOCK(A) $\cdots$ UNLOCK(A), and R_2 of count $j \geq 1$ has the form LOCK(B) $\cdots$ UNLOCK(B). If A = B, then R_1 and R_2 commute by the definition of abelian. Parts 1 and 2 then both follow from the definition of commutativity. Otherwise, there are three possible cases.

Case 1: Lock A does not appear in R_2 (R_2'). For part 1, we start with the sequence $X_1R_1R_2X_2$ and commute pieces of R_1 one at a time with R_2: first, the instantiation UNLOCK(A), then the immediate subregions of R_1, and finally the instantiation LOCK(A). The instantiations LOCK(A) and UNLOCK(A) commute with R_2, because A does not appear anywhere in R_2. Each subregion of R_1 commutes with R_2 by the inductive hypothesis, because each subregion has lower nesting count than R_1. After commuting all of R_1 past R_2, we have an equivalent execution $X_1R_2R_1X_2$. For part 2, the same procedure can be used to drop pieces of R_1 in the true partial schedule $Y_1R_1R_2'$ until the true partial schedule Y_1R_2' is reached.

Case 2: Lock B does not appear in R_1. The argument for part 1 is symmetric with Case 1. For part 2, we break up R_2' into its constituents: $R_2' =$ LOCK(B)$R_{2,1} \ldots R_{2,n}R_2''$, where $R_{2,1}$ through $R_{2,n}$ are complete regions, and R_2'' is a prefix of a region. The instantiation LOCK(B) commutes with R_1 because B does not appear in R_1, and the complete regions $R_{2,1}$ through $R_{2,n}$ commute with R_1 by induction. From the schedule Y_1LOCK(B)$R_{2,1} \ldots R_{2,n}R_1R_2''$, we again apply the inductive hypothesis to drop R_1, which proves that Y_1LOCK(B)$R_{2,1} \ldots R_{2,n}R_2'' = Y_1R_2'$ is true.

Case 3: Lock A appears in R_2 (R_2'), and lock B appears in R_1. For part 1, if both schedulings $X_1R_1R_2X_2$ and $X_1R_2R_1X_2$ are false, then we are done. Otherwise, we prove a contradiction by showing that the program can deadlock. Without loss of generality, let the scheduling $X_1R_1R_2X_2$ be a true scheduling. Because $X_1R_1R_2X_2$ is a true scheduling, the partial scheduling $X_1R_1R_2$ is true as well.

We now continue the proof for both parts of the lemma. Let α_1 be the prefix of R_1 up to the first LOCK_ATTEMPT(B) instantiation, let β_1 be the rest of R_1, and let α_2 be the prefix of R_2 (R_2') up to the first LOCK_ATTEMPT of a lock acquired in R_2 (R_2') that is acquired but not released in α_1. At least one such lock exists, namely A, so α_2 is not all of R_2 (R'_2).

We show that the partial scheduling $X_1\alpha_1\alpha_2$ is also true. This partial scheduling, however, cannot be completed to a full scheduling of the program because α_1 and α_2 each hold the lock that the other is attempting to acquire.

We prove the partial scheduling $X_1\alpha_1\alpha_2$ is true by starting with the true partial scheduling $X_1R_1\alpha_2 = X_1\alpha_1\beta_1\alpha_2$ and dropping complete subregions and unpaired unlocks in β_1 from in front of α_2. The sequence β_1 has two types of instantiations, those in regions completely contained in β_1, and unpaired unlocks.

Unpaired unlocks in β_1 must have their matching lock in α_1, so that lock does not appear in α_2 by construction. Therefore, an unlock instantiation just before α_2 commutes with α_2 and thus can be dropped from the schedule. Any complete region just before α_2 can be dropped by the inductive hypothesis. When we have dropped all instantiations in β_1, we obtain the true partial scheduling $X_1\alpha_1\alpha_2$ which cannot be completed, and hence the program has a deadlock. ∎

Lemma 29 (Region grouping) *Let G be a data-race free computation resulting from the execution of a deadlock-free abelian program. Let X_1XX_2 be a scheduling of G, for some instantiation sequences X_1, X, and X_2. Then, there exists an instantiation sequence X' such that $X_1X'X_2$ is equivalent to X_1XX_2 and every region entirely contained in X' is contiguous.*

Proof: As a first step, we create X'' by commuting each LOCK_ATTEMPT in X to immediately before the corresponding LOCK_SUCCEED. In this way, every complete region begins with a LOCK instantiation. If there is no corresponding LOCK_SUCCEED in X, we commute the LOCK_ATTEMPT instantiation to the end of X''.

Next, we create our desired X' by grouping all the complete regions in X'' one at a time. Each grouping operation will not destroy the grouping of already grouped regions, so eventually all complete regions will be grouped.

Let R be a noncontiguous region in X'' that completely overlaps no other noncontiguous regions in X''. Since region R is noncontiguous, other regions parallel with R must overlap R in X''. We first remove all overlapping regions which have exactly one endpoint (an endpoint is the bounding LOCK or UNLOCK of a region) in R, where by "in" R, we mean appearing in X'' between the endpoints of R. We shall show how to remove regions which have only their UNLOCK in R. The technique for removing regions with only their LOCK in R is symmetric.

Consider the partially overlapping region S with the leftmost UNLOCK in R. Then all subregions of S which have any instantiations inside R are completely inside R and are therefore contiguous. We remove S by moving each of its (immediate) subregions in R to just left of R using commuting operations. Let S_1 be the leftmost subregion of S which is also in R. We can commute S_1 with every instruction I to its left until it is just past the start of R. There are three cases for the type of instruction I. If I is not a LOCK or UNLOCK, it commutes with S_1 by Lemma 28 because it is a region in parallel with S_1. If $I =$ LOCK(B) for some lock B, then S_1 commutes with I, because S_1 cannot contain LOCK(B) or UNLOCK(B). If $I =$ UNLOCK(B), then there must exist a matching LOCK(B) inside R, because S is chosen to be the region with the leftmost UNLOCK without a matching LOCK. Since there is a matching LOCK in R, the region defined by the LOCK/UNLOCK pair must be contiguous by the choice of R. Therefore, we can commute S_1 with this whole region at once using Lemma 28.

We can continue to commute S_1 to the left until it is just before the start of R. Repeat for all other subregions of S, left to right. Finally, the UNLOCK at the end of S can be moved to just before R, because no other LOCK or UNLOCK of that same lock appears in R up to that UNLOCK.

Repeat this process for each region overlapping R that has only an UNLOCK in R. Then, remove all regions which have only their LOCK in R by pushing them to just after R using similar techniques. Finally, when there are no more unmatched LOCK or UNLOCK instantiations in R, we can remove any remaining overlapping regions by pushing them in either direction to just before or just after R. The region R is now contiguous.

Repeating for each region, we obtain an execution $X_1X'X_2$ equivalent to X_1XX_2 in which every region completely contained in X' is contiguous. ■

Lemma 30 *Let G be a data-race free computation resulting from the execution of a deadlock-free abelian program. Then every scheduling of G is true and yields the same final memory state.*

Proof: The proof is identical to the proof of Lemma 12, using the Reordering and Region Grouping lemmas from this appendix in place of those from Section 5.5.

We restate and then prove Lemma 14.

Lemma 14 *Let G be a computation generated by a deadlock-free abelian program. If G is data-race free, then it is deadlock free.*

Proof: By contradiction. Assume that a deadlock-free abelian program P can generate a data-race free computation G that has a deadlock. We show that P can deadlock, which is a contradiction.

The proof has two parts. In the first part, we generate a true scheduling Y of G that is "almost" a deadlock scheduling. Then, we show that Y can be modified slightly to generate a deadlock scheduling Z which is also true, which proves the contradiction.

Every deadlock scheduling contains a set of threads $e_1, e_2, \ldots e_n$, some of which are completed and some of which are not. Each thread e_i has a ***depth***, which is the length of the longest path in G from the initial node to the last instantiation in e_i. We can define the ***depth*** of a deadlock scheduling as the n-tuple $\langle \mathit{depth}(e_1), \mathit{depth}(e_2), \ldots, \mathit{depth}(e_n) \rangle$, where we order the threads such that $\mathit{depth}(e_1) \geq \mathit{depth}(e_2) \geq \ldots \geq \mathit{depth}(e_n)$. Depths of deadlocked schedulings are compared in the dictionary order.[2]

We generate the scheduling Y of G which is almost a deadlock scheduling by modifying a particular deadlock scheduling of G. We choose the deadlock scheduling X from which we will create the scheduling Y to have the maximum depth of any deadlock scheduling of G.

[2] The dictionary order $<_D$ is a partial order on tuples that can be defined as follows: The size 0 tuple is less than any other tuple. $\langle i_i, i_2, \ldots, i_m \rangle <_D \langle j_1, j_2, \ldots, j_n \rangle$ if $i_1 < j_1$ or if $i_1 = j_1$ and $\langle i_2, i_3, \ldots, i_m \rangle <_D \langle j_2, j_3, \ldots, j_n \rangle$.

Let us examine the structure of X in relation to G. The deadlock scheduling X divides G into a set of completely executed threads, X_1, a set of unexecuted threads X_2, and a set of partially executed threads $T = \{t_1, \ldots, t_n\}$, which are the threads whose last executed instantiation in the deadlock scheduling is a LOCK_ATTEMPT. We divide each of the threads in T into two pieces. Let $A = \{\alpha_1, \ldots, \alpha_n\}$ be the parts of the t_i up to and including the last executed instantiation, and let $B = \{\beta_1, \ldots, \beta_n\}$ be the rest of the instantiations of the t_i. We say that α_i ***blocks*** β_j if the first instantiation in β_j is a LOCK_SUCCEED on a lock that is acquired but not released by α_i.

X is a deadlock scheduling containing the instantiations in $X_1 \cup A$. To isolate the effect of the incomplete regions in A, we construct the legal scheduling X' which first schedules all of the instantiations in X_1 in the same order as they appear in X, and then all of the instantiations in A in the same order as they appear in X.

The first instantiations of the β_i cannot be scheduled in X' because they blocked by some α_j. We now prove that the blocking relation is a bijection. Certainly, a particular β_i can only be blocked by one α_j. Suppose there exists an α_j blocking two or more threads in B. Then by the pigeonhole principle some thread α_k blocks no threads in B. This contradicts that fact that X has maximum depth, because the deadlock scheduling obtained by scheduling the sequence $X_1 t_k$, all subsequently runnable threads in X_2 in any order, and then the $n-1$ partial threads in $A - \{\alpha_k\}$ is a deadlock scheduling with a greater depth than X.

Without loss of generality, let α_2 be a thread in A with a deepest last instantiation. Since the blocking relation is a bijection, only one thread blocks β_2; without loss of generality, let it be α_1. Break α_1 up into two parts, $\alpha_1 = \alpha_1^L \alpha_1^R$, where the first instantiation of α_1^R attempts to acquire the lock that blocks β_2. (α_1^L may be empty.) To construct a legal schedule, we start with X' and remove the instantiations in α_1^R from X'. The result is still a legal scheduling because we did not remove any unlock without also removing its matching lock. We then schedule the first instantiation of β_2, which we know is legal because we just unblocked it. We then complete the scheduling of the threads in T by scheduling the remaining instantiations in T (α_1^R and all instantiations in B except for the first one in β_2). We know that such a scheduling exists, because if it didn't, then there would be a deeper deadlock schedule (because we executed one additional instantiation from β_2, the deepest incomplete thread, and we didn't remove any completed threads). We finish off this legal scheduling by completing X_2 in topological sort order.

As a result, the constructed schedule consists of four pieces, which we call Y_1, Y_2, Y_3', and Y_4. The instantiation sequence Y_1 is some scheduling of the instantiations in X_1, Y_2 is some scheduling of the instantiations in $\alpha_1^L \cup \alpha_2 \cup \ldots \cup \alpha_n$, Y_3' is some scheduling of the instantiations in $\alpha_1^R \cup \beta_1 \cup \ldots \cup \beta_n$, and Y_4 is some scheduling of the instantiations in X_2. To construct Y, we first group the complete regions in Y_3' using Lemma 29 to get Y_3, and then define Y to be the schedule $Y_1Y_2Y_3Y_4$. Since Y is a (complete) scheduling of G, it is true by Lemma 30.

The true scheduling Y is almost the same as the deadlock scheduling X, except α_1^R is not in the right place. We further subdivide α_1^R into two pieces, $\alpha_1^R = \alpha_1'\alpha_1''$, where α_1' is the maxmimum prefix of α_1^R that contains no LOCK_SUCCEED instantiations of locks that are held but not released by the instantiations in $\alpha_1^L, \alpha_2, \ldots, \alpha_n$. (Such an α_1' must exist in α_1^R by choice of α_1^R, and furthermore α_1' is contiguous in Y because β_1 completes the region started at α_1', and both β_1 and α_1' are part of Y_3.) We now drop all instantiations after α_1' to make a partial scheduling. We then commute α_1' to the beginning of Y_3, dropping instantiations as we go, to form the true scheduling $Y_1Y_2\alpha_1'$. Two types of instantiations are in front of α_1'. Complete regions before α_1' are contiguous and can be dropped using Lemma 28. Unlock instantiations can be dropped from in front of α_1' because they are unlocks of some lock acquired in $\alpha_1^L, \alpha_2, \ldots, \alpha_n$, which do not appear in α_1' by construction. By dropping instantiations, we arrive at the true scheduling $Y_1Y_2\alpha_1'$, which is a deadlock scheduling, as every thread is blocked. This contradiction completes the proof. ∎

Bibliography

[1] Sarita V. Adve and Mark D. Hill. Weak ordering — a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA)*, 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] Andrew W. Appel and Zhong Shao. Empirical and analytic study of stack versus heap cost for languages with closures. *Journal of Functional Programming*, 6(1):47–74, 1996.

[4] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Puerto Vallarta, Mexico, June 1998. To appear.

[5] Joshua E. Barnes. A hierarchical $O(N \log N)$ N-body code. Available on the Internet from `ftp://hubble.ifa.hawaii.edu/pub/barnes/treecode/`.

[6] Philippe Bekaert, Frank Suykens de Laet, and Philip Dutre. Renderpark, 1997. Available on the Internet from `http://www.cs.kuleuven.ac./cwis/research/graphics/RENDERPARK/`.

[7] 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.

[8] 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.

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

[10] 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 (SPAA)*, pages 1–12, Santa Barbara, California, July 1995.

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

[12] 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 (IPPS)*, pages 132–141, Honolulu, Hawaii, April 1996.

[13] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 297–308, Padua, Italy, June 1996.

[14] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

[15] 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.

[16] 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 (FOCS)*, pages 356–368, Santa Fe, New Mexico, November 1994.

[17] Ronald F. Boisvert, Roldan Pozo, Karin Remington, Richard Barrett, and Jack J. Dongarra. The matrix market: A web resource for test matrix collections. In Ronald F. Boisvert, editor, *Quality of Numerical Software, Assessment and Enhancement*, pages 125–137. Chapman & Hall, London, 1997. Web address `http://math.nist.gov/MatrixMarket`.

[18] G. A. Boughton. Arctic routing chip. In *Proceedings of Hot Interconnects II*, August 1994.

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

[20] 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 (SOSP)*, pages 152–164, Pacific Grove, California, October 1991.

[21] David Chaiken and Anant Agarwal. Software-extended coherent shared memory: Performance and cost. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA)*, pages 314–324, Chicago, Illinois, April 1994.

[22] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP)*, pages 147–158, Litchfield Park, Arizona, December 1989.

[23] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Puerto Vallarta, Mexico, June 1998. To appear.

[24] Cilk-5.1 (Beta 1) Reference Manual. Available on the Internet from `http://theory.lcs.mit.edu/~cilk`.

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

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

[27] Daved E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Supercomputing '93*, pages 262–273, Portland, Oregon, November 1993.

[28] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 164–175, Santa Clara, California, April 1991.

[29] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

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

[31] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–96. ACM Press, May 1991.

[32] Anne Carolyn Dinning. *Detecting Nondeterminism in Shared Memory Parallel Programs.* PhD thesis, Department of Computer Science, New York University, July 1990.

[33] 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.

[34] Perry A. Emrath, Sanjoy Ghosh, and David A. Padua. Event synchronization analysis for debugging parallel programs. In *Supercomputing '91*, pages 580–588, November 1991.

[35] Andrew Erlichson, Neal Nuckolls, Greg Chesson, and John Hennessy. Soft-FLASH: Analyzing the performance of clustered distributed shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 210–220, Cambridge, Massachusetts, October 1996.

[36] Marc Feeley. Polling efficiently on stock hardware. In *Proceedings of the 1993 ACM SIGPLAN Conference on Functional Programming and Computer Architecture*, pages 179–187, Copenhagen, Denmark, June 1993.

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

[38] Yaacov Fenster. Detecting parallel access anomalies. Master's thesis, Hebrew University, March 1998.

[39] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 201–213, Monterey, California, November 1994.

[40] Matteo Frigo. The weakest reasonable memory model. Master's thesis, Massachusetts Institute of Technology, 1997.

[41] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, June 1998. To appear.

[42] Matteo Frigo and Victor Luchangco. Computation-centric memory models. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Puerto Vallarta, Mexico, June 1998. To appear.

[43] Guang R. Gao and Vivek Sarkar. Location consistency: Stepping beyond the memory coherence barrier. In *Proceedings of the 24th International Conference on Parallel Processing*, Aconomowoc, Wisconsin, August 1995.

[44] 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 (ISCA)*, pages 15–26, Seattle, Washington, June 1990.

[45] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 136–146, Berkeley, California, 28–30 May 1986.

[46] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, August 1996.

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

[48] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[49] Dirk Grunwald. Heaps o' stacks: Time and space efficient threads without operating system support. Technical Report CU-CS-750-94, University of Colorado, November 1994.

[50] Dirk Grunwald and Richard Neves. Whole-program optimization for time and space efficient threads. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 50–59, Cambridge, Massachusetts, October 1996.

[51] Michael Halbherr, Yuli Zhou, and Chris F. Joerg. MIMD-style parallel programming with continuation-passing threads. In *Proceedings of the 2nd International Workshop on Massive Parallelism: Hardware, Software, and Applications*, Capri, Italy, September 1994.

[52] E. A. Hauck and B. A. Dent. Burroughs' B6500/B7500 stack mechanism. *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 245–251, 1968.

[53] David P. Helmbold, Charles E. McDowell, and Jian-Zhong Wang. Analyzing traces with anonymous synchronization. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages II70–II77, August 1990.

[54] John L. Hennessy and David A. Patterson. *Computer Architecture: a Quantitative Approach.* Morgan Kaufmann, San Mateo, CA, 1990.

[55] John L. Hennessy and David A. Patterson. *Computer Architecture: a Quantitative Approach.* Morgan Kaufmann, San Francisco, CA, second edition, 1996.

[56] James C. Hoe. StarT-X: A one-man-year exercise in network interface engineering. In *Proceedings of Hot Interconnects VI*, August 1998.

[57] Richard C. Holt. Some deadlock properties of computer systems. *Computing Surveys*, 4(3):179–196, September 1972.

[58] 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.

[59] 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 (SOSP)*, pages 213–228, Copper Mountain Resort, Colorado, December 1995.

[60] 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.

[61] 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.

[62] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford Flash multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA)*, pages 302–313, Chicago, Illinois, April 1994.

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

[64] 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 (ASPLOS)*, pages 208–218, San Jose, California, October 1994.

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

[66] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM-5. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 272–285, San Diego, California, June 1992.

[67] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[68] Phillip Lisiecki and Alberto Medina. Personal communication.

[69] Victor Luchangco. Precedence-based memory models. In *Eleventh International Workshop on Distributed Algorithms (WDAG97)*, number 1320 in Lecture Notes in Computer Science, pages 215–229. Springer-Verlag, September 1997.

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

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

[72] Barton P. Miller and Jong-Deok Choi. A mechanism for efficient debugging of parallel programs. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 135–144, Atlanta, Georgia, June 1988.

[73] James S. Miller and Guillermo J. Rozas. Garbage collection is fast, but a stack is faster. Technical Report Memo 1462, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1994.

[74] Robert C. Miller. A type-checking preprocessor for Cilk 2, a multithreaded C language. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1995.

[75] Sang Lyul Min and Jong-Deok Choi. An efficient cache-based access anomaly detection scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 235–244, Palo Alto, California, April 1991.

[76] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[77] Joel Moses. The function of FUNCTION in LISP or why the FUNARG problem should be called the envronment problem. Technical Report memo AI-199, MIT Artificial Intelligence Laboratory, June 1970.

[78] Greg Nelson, K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Extended static checking home page, 1996. Available on the Internet from `http://www.research.digital.com/SRC/esc/Esc.html`.

[79] Robert H. B. Netzer and Sanjoy Ghosh. Efficient race condition detection for shared-memory programs with post/wait synchronization. In *Proceedings of the 1992 International Conference on Parallel Processing*, St. Charles, Illinois, August 1992.

[80] Robert H. B. Netzer and Barton P. Miller. On the complexity of event ordering for shared-memory parallel program executions. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages II: 93–97, August 1990.

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

[82] Rishiyur Sivaswami Nikhil. Parallel Symbolic Computing in Cid. In *Proc. Wkshp. on Parallel Symbolic Computing, Beaune, France, Springer-Verlag LNCS 1068*, pages 217–242, October 1995.

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

[84] Dejan Perković and Peter Keleher. Online data-race detection via coherency guarantees. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, Washington, October 1996.

[85] Keith H. Randall. Solving Rubik's cube. In *Proceedings of the 1998 MIT Student Workshop on High-Performance Computing in Science and Engineering*, January 1998.

[86] 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 (ISCA)*, pages 325–336, Chicago, Illinois, April 1994.

[87] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 54–67, Philadelphia, Pennsylvania, May 1996.

[88] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Jade: A high-level machine-independent language for parallel programming. *Computer*, 26(6):28–38, June 1993.

[89] Stefan Savage, Michael Burrows, Greg Nelson, Patric Sobalvarro, and Thomas Anderson. Eraser: A dynamic race detector for multi-threaded programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, October 1997.

[90] 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.

[91] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 297–306, San Jose, California, October 1994.

[92] Jaswinder Pal Singh. Personal communication.

[93] David Singmaster. *Notes on Rubik's Magic Cube.* Enslow Publishers, Hillside, New Jersey, 1980.

[94] Per Stenström. VLSI support for a cactus stack oriented memory organization. *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences, volume 1*, pages 211–220, January 1988.

[95] Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen Hunt, Leonidas Kontothanassis, Srinivasan Parthasarathy, and Michael Scott. Cashmere-2L: Software coherent shared memory on a clustered remote-write network. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malo, France, October 1997.

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

[97] Andrew S. Tanenbaum, Henri E. Bal, and M. Frans Kaashoek. Programming a distributed system using shared objects. In *Proceedings of the Second International Symposium on High Performance Distributed Computing*, pages 5–12, Spokane, Washington, July 1993.

[98] Robert Endre Tarjan. Applications of path compression on balanced trees. *Journal of the Association for Computing Machinery*, 26(4):690–715, October 1979.

[99] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[100] David S. Wise. Representing matrices as quadtrees for parallel processors. *Information Processing Letters*, 20(4):195–199, July 1985.

[101] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[102] Matthew J. Zekauskas, Wayne A. Sawdon, and Brian N. Bershad. Software write detection for a distributed shared memory. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 87–100, Monterey, California, November 1994.