The Cilk System for Parallel Multithreaded Computing

by

Christopher F. Joerg

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 1996

Author ....[illegible]..
Department of Electrical Engineering and Computer Science
January, 1996

Certified by ....[illegible]..
Charles E. Leiserson
Professor
Thesis Supervisor

Accepted by ....[illegible]..
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

The Cilk System for Parallel Multithreaded Computing

by

Christopher F. Joerg

Submitted to the Department of Electrical Engineering and Computer Science
on January, 1996, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Although cost-effective parallel machines are now commercially available, the widespread use of parallel processing is still being held back, due mainly to the troublesome nature of parallel programming. In particular, it is still difficult to build efficient implementations of parallel applications whose communication patterns are either highly irregular or dependent upon dynamic information. Multithreading has become an increasingly popular way to implement these dynamic, asynchronous, concurrent programs. Cilk (pronounced "silk") is our C-based multithreaded computing system that provides provably good performance guarantees. This thesis describes the evolution of the Cilk language and runtime system, and describes applications which affected the evolution of the system.

Using Cilk, programmers are able to express their applications either by writing multithreaded code written in a continuation-passing style, or by writing code using normal call/return semantics and specifying which calls can be performed in parallel. The Cilk runtime system takes complete control of the scheduling, load-balancing, and communication needed to execute the program, thereby insulating the programmer from these details. The programmer can rest assured that his program will be executed efficiently since the Cilk scheduler provably achieves time, space, and communication bounds all within a constant factor of optimal. For distributed memory environments, we have implemented a software shared-memory system for Cilk. We have defined a "dag-consistent" memory model which is a lock-free consistency model well suited to the needs of a multithreaded program. Because dag consistency is a weak consistency model, we have been able to implement coherence efficiently in software.

The most complex application written in Cilk is the ⋆Socrates computer chess program. ⋆Socrates is a large, nondeterministic, challenging application whose complex control dependencies make it inexpressible in many other parallel programming systems. Running on an 1824-node Paragon, ⋆Socrates finished second in the 1995 World Computer Chess Championship.

Currently, versions of Cilk run on the Thinking Machines CM-5, the Intel Paragon, various SMPs, and on networks of workstations. The same Cilk program will run on all of these platforms with little, if any, modification. Applications written in Cilk include protein folding, graphic rendering, backtrack search, and computer chess.

Thesis Supervisor: Charles E. Leiserson
Title: Professor

Acknowledgments

I am especially grateful to my thesis supervisor, Professor Charles Leiserson, who has led the Cilk project. I still remember the day he came to my office and recruited me. He explained how he realized I had other work to do but he wanted to know if I would like to help out "part time" on implementing a chess program using PCM. It sounded like a interesting project, so I agreed, but only after making it clear that I could only work part time because I had my thesis project to work on. Well, "part time" became "full time", and at times "full time" became much more than that. Eventually, the chess program was completed, and the chess tournament came and went, yet I still kept working on the PCM system (which was now turning into Cilk). Ultimately, I realized that I should give up on my other project and make Cilk my thesis instead. Charles is a wonderful supervisor and under his leadership, the Cilk project has achieved more than I ever expected. Charles' influence can also be seen in this write-up itself. He has helped me turn this thesis into a relatively coherent document, and he has also pointed out some of my more malodorous grammatical constructions.

The Cilk project has been a team effort and I am indebted to all the people who have contributed in some way to the Cilk system: Bobby Blumofe, Feng Ming Dong, Matteo Frigo, Shail Aditya Gupta, Michael Halbherr, Charles Leiserson, Bradley Kuszmaul, Rob Miller, Keith Randall, Rolf Riesen, Andy Shaw, Richard Tauriello, and Yuli Zhou. Their contributions are noted throughout this document.

I thank, along with the members of the Cilk team, the past and present members of the Computation Structures Group. These friends have made MIT both a challenging and a fun place to be. In particular I should thank Michael Halbherr. He not only began the work that lead to the PCM system, but he tried many times to convince me to switch my thesis to this system. It took a while, but I finally realized he was right.

I am also indebted to Don Dailey and Larry Kaufman, both formerly of Heuristic Soft-

The research described in this document was supported in part by the Advanced Research Projects Agency of the Department of Defense under grants N0014-94-1-0985 and N0014-92-J-1310. This work was also supported by the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champagne (NCSA) who, under NCSA Grant TRA930289N, provided us access to their 512-processor CM-5 for the 1994 chess tournament and by Sandia National Laboratories who provided us access to their 1824-node Intel Paragon for the 1995 tournament.

ware. They wrote the serial Socrates program on which ⋆Socrates is based. In addition, Don and I spent many long nights debugging, testing, and improving (or at least trying to improve) ⋆Socrates. Most of this time we even had fun.

Professor Arvind, Dr. Andy Boughton, and Dr. Greg Papadopoulus also deserve many thanks. They provided me the freedom, encouragement, and support to work on a wide range of exciting projects throughout my years at MIT.

I am also grateful to my parents and my family. Their love and support has always been important to me.

Last, but not least, I thank Constance Jeffery. Whether we were together, apart, or off on one of our many trips ranging from Anchorage to Zurich, her continuing friendship over the past decade has made these years enjoyable and memorable.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Researchers have long worked to bring parallel hardware and software into widespread use. Recently there has been progress on the hardware front. Serial microprocessors have been used as cost-effective building blocks for medium and large scale parallel machines. Now many high-volume serial processors contain hooks, such as snoopy buses [KEW$^+$85], for implementing multiprocessor systems. These hooks make it quite simple and cheap for commercial computer manufacturers to build inexpensive, entry-level, multiprocessor machines. This trend towards including multiprocessor support in standard microprocessors occurred first with processors used in workstations (e.g. MIPS R4000[MWV92], Sparc[Sun89], PowerPC 601 [Mot93]) and more recently with processors for PCs (e.g. Intel's Pentium P54C [Gwe94]). As with any other commodity, as parallel machines drop in price, they become cost-effective in new areas, leading to parallel machines being installed at more and more sites. If this trend wasn't enough, high-speed networks and lower-overhead software are threatening to turn every LAN into a potential parallel machine [ACP95]. We may finally be witnessing the move of parallel machines into the mainstream.

Although building parallel computers has become easier, programming parallel computers can still be quite difficult. To see that parallel programming has not moved into the mainstream, just take note of the way in which existing small-scale SMPs are being used. This usage pattern is hard to document quantitatively, but as an example, "Open Computing" estimates that of all the high end, 2-8 processor PCs sold, 70% of them are used as file and print servers [EL94]. It seems that most small scale SMPs are destined to spend their lives as "throughput machines," never to run a single parallel job. We should not be too negative, however, since clearly some progress has been made on the software front. Unfortunately much of this progress has been in programming languages which are suited to static programs. A static program is one whose control behavior is relatively data-independent, so the computation can be fairly well mapped out before the computation begins. Typical of static programs are the large scientific and numeric codes that were the the *raison d'être* of supercomputers and of early, expensive parallel machines. Naturally, much of the early research in parallel programming was directed towards programming these applications. Even today, the suites commonly used to benchmark parallel machines (e.g. Perfect [BCK$^+$89], NAS [BBB$^+$94], and Linpack [DMBS79]) are representative of such programs. Less progress has been made for easing the task of writing parallel programs for dynamic applications. These programs are ones where the execution of the program is heavily influenced by the data input to the program and by the data computed by the program. These applications include compilers, simulators, graphic packages, and optimization packages. There has been less work on building systems suited for dynamic applications such as these, yet these are exactly the applications that new users of cheap parallel machines will want to run. The goal of the work presented in this thesis is to partially address this inadequacy by building a system that allows a programmer to easily and efficiently implement certain types of dynamic parallel algorithms.

To reach this goal, the Cilk team at MIT's Laboratory for Computer Science has designed Cilk (pronounced "silk"), a C-based runtime system for multithreaded parallel programming. Using Cilk, programmers are able to express their applications either by writing multithreaded code written in a continuation-passing style, or by writing code using normal call/return semantics and specifying which calls can be performed in parallel. In the latter case a type-checking preprocessor automatically breaks up the code into a multithreaded program. The Cilk runtime system takes complete control of the scheduling, load balancing, and communication needed to execute the multithreaded program, thereby completely insulating the programmer from these details. The programmer can rest assured that his program will be executed efficiently, since the Cilk scheduler provably achieves time, space, and communication bounds all within a constant factor of optimal. The Cilk system reports the "work" and "critical path" of a Cilk computation. Armed with these parameters, the user can understand and accurately predict the performance of a program. For distributed memory environments, we have implemented a software shared memory system for Cilk.

We have defined a "dag-consistent" memory model which is a lock-free consistency model well suited to the needs of a multithreaded program. Because dag consistency is a relaxed consistency model, we were able to implement coherence efficiently in software. Currently, versions of Cilk run on the Thinking Machines CM-5 [Thi92], the Intel Paragon [Int94], various SMPs, and on networks of workstations [Blu95]. The same Cilk program will run on all of these platforms with few, if any, modifications. Applications written in Cilk include protein folding, graphic rendering, backtrack search, and the ⋆Socrates chess program, which won second prize in the 1995 World Computer Chess Championship.

I could now describe how having decided upon all the nice features Cilk should have, we went off and designed and built a system that contained those features. It would make a nice story. It just wouldn't be a true story. When we first started this project, we had little idea what the final system would look like. In fact, had we sat down and set as our goal to build a system that looks like Cilk does today, we probably would have decided against pursuing the project, and instead worked on a different project, one which had a better chance of being done in a reasonable time frame.

Instead, the story of Cilk is one of incremental improvement. We had some goals and some ideas on how to reach those goals. Throughout our work on Cilk, we used applications to help drive the development process. Applications were useful both in pointing out bugs and weaknesses in the system, as well as in helping us decide where to focus our energies next. We also tried to keep a firm theoretical footing, so that we could truly understand the performance of the system.

We began by building a simple multithreaded runtime system, and then we stepped back and asked ourselves what were the biggest deficiencies with this system, and what could be done to improve them. We then chose one of the problems that we thought we could remedy and went off and focused on it, being careful not to reintroduce the problems we had previously solved. Once this deficiency was addressed we repeated the process and again stepped back, examined the system, and looked for the next target area. This picture of the development of Cilk is somewhat simplistic, since in actuality we were always thinking about the bigger picture and at times we were working in several directions at once, but in general the paradigm of incremental improvement is a good model for the evolution of Cilk.

In the rest of this chapter we give an overview of the development of the Cilk system. In Section 1.1 we briefly look at some other parallel programming systems and summarizing what desirable characteristics we think a programming system should have. Then in Section 1.2 we describe the evolution of Cilk. We begin with PCM, a simple multithreaded runtime system based on continuation-passing threads. The PCM system evolved into the Cilk-1 system with the addition of a provably good scheduler which allowed us to provide performance guarantees. Next, the Cilk-2 system extended Cilk-1 by allowing the user to program using call-return semantics instead of continuation passing. Then, in Cilk-3, we added shared memory support in order to extend the range of applications we could implement. Lastly, we added high-level support for speculative computations via inlets and aborts.

1.1 Life Before Cilk

In April 1993, a 128 node Thinking Machines CM-5 was installed at MIT. We were, of course, eager to begin using this machine in our research. We wanted to use this machine to experiment with parallel algorithms and wanted a programming language/environment in which to do this. Preferably we wanted an environment where we could focus on our application, not on the low-level protocols necessary to implement the application. At the same time we wanted our application to run efficiently. In short, we wanted the best of both worlds. Before describing the system we designed, we will first take a look at what other parallel programming paradigms were available.

Data Parallel

One of the most successful parallel programming models is the data-parallel programming paradigm[HS86]. This model is useful for taking advantage of the large amounts of data parallelism that is available in many scientific/numeric applications. This data parallelism is exploited by performing the same operation on a large amount of data, distributed across the processors of the machine. Data-parallel languages, such as CM Fortran [Thi91a], C* [Thi93], and *Lisp [Thi91b], all of which were available on the CM-5, are similar to sequential languages. The main difference is that certain data types are defined to be parallel. Parallel data values consist of a collection of standard, scalar data values. These languages contain predefined operations on parallel variables that either operate on the parallel variable element-wise (e.g. negating every element), or operate on the parallel value as a whole (e.g. summing all elements of the parallel variable).

The data-parallel programming model has two main virtues that have led to its success. The first virtue of this model is that data-parallel codes are fairly easy to write and debug [HS86]. Just as in a serial program, the programmer sees a sequential flow of control. The values making up a parallel value are automatically spread across the machine, although typically the programmer does have the option of influencing how data is placed. Any synchronization or communication that is needed to perform an operation on a parallel value is automatically added by the compiler/runtime system. The second virtue of this model is that it is easy for a programmer to understand the performance of a program. Given the size of a parallel value to be operated on, the execution time for an operation is fairly predictable. Since the execution of each operation is independent of the others, the execution time for the program as a whole is predictable as well. A careful programmer can therefore write a program and be confident that the program's performance will scale as the machine size grows. Blelloch has taken this a step further in NESL [Ble93], where every built in function has two complexity measures, which a programmer can use to derive the asymptotic running time of his program.

Although the data-parallel paradigm is quite popular, it has two significant drawbacks. The first is the limited range of applications for which data parallel is well suited. Applications with data parallelism tend to be static in nature, the control flow of a data-parallel program is mostly data independent, and the program's data layout and load balancing can be done at compile time. Many applications are more dynamic in nature and do not have these characteristics. To run in parallel, these dynamic applications need to exploit control parallelism by performing independent operations at the same time. These applications, which may be as simple as recursively computing Fibonacci numbers or as complex as computer chess, are nearly impossible to express in data-parallel languages. The second drawback of this model is that data-parallel programs tend to be inefficient. Even when a data-parallel program gets a good speedup, if one scales the program down to one processor, and compares it to a sequential program, the performance may be disappointing. This phenomenon occurs because the data-parallel paradigm is not always a good model for taking full advantage of the sequential processors that make up most of today's parallel machines. This deficiency is particularly acute for languages such as CM Fortran, where the code generated uses "virtual processors". A virtual processor mechanism allows the same code to run on a machine of any size, but it adds significant inefficiencies [HKT93].

Message Passing

Another common paradigm for writing parallel programs is message passing. Message-passing models present the programmer with one thread of control in each processor, and these processors communicate by sending messages. This model is a good representation of the actual implementation of current parallel machines. Since this model is so close to the hardware, a good programmer is able to write efficient codes, just as a good assembly language programmer is able to write assembly language code that is more efficient than code written in a high-level language. The drawback of this model is the same as the drawback of programming in assembly language: writing a large program at such a low level can be overwhelming. The user must answer all the low-level questions himself, namely questions such as how to partition the program's data, when to perform communication, and how to load balance the computation. Not only must the user make all these decisions, but he must then write all the protocols necessary to carry them out. For most nontrivial programs the user spends more time writing protocols that writing the actual application. I believe we should aspire to a higher level of parallel programming.

There are three strategies for message-passing programming.

The simplest message-passing models are blocking. The sending processor issues a send request, and the receiving processor issues a receive request. Whichever processor issues its request first blocks and sits idle until the other processor issues its command. At that point, communication begins. Only after communication completes can the processors continue executing. It can be difficult to program well in this model, because inefficiencies occur unless both of the processors involved in a communication issue their requests at the same time. Moreover, this style of programming is prone to deadlock.

To make programming simpler, many systems implement a second type of message passing: "asynchronous" message passing. In this model, when a processor performs a send, the send executes immediately, regardless of whether or not a corresponding receive has been issued, and the sending processor can continue executing. The system uses buffers (often on both the sending and receiving side) to hold the message until it is requested by the receiver. Asynchronous message passing eases the programmer's job, but adds significant overhead to each communication due to the copying and buffering that the system invisibly performs.

Active Messages [vECGS92], the third strategy for message passing, reduces this overhead by providing asynchronous message passing without the automatic buffering. An active message contains a header which points to a *handler*, which is a piece of user code that specifies what to do with the data in the message. The user can specify many handlers, typically one for each message type. When a message arrives, rather than having a generic system-defined routine handle the message, which will typically copy the message into a buffer, the system instead executes this user-defined handler to process the arrived message. Active Messages eases the task of writing message-passing codes because it allows a programmer to write programs using low-overhead, asynchronous message passing and because the paradigm of having the message itself know how it should be handled turns out to be quite useful in practice.

Although Active Messages simplifies the task of writing an efficient message-passing program, it is still just a message-passing paradigm, and is therefore too low-level to be a general parallel programming language. We should point out that the authors of the Active Messages paper themselves state that Active Messages was not designed as a new parallel programming paradigm, but rather as a primitive communication mechanism with which other paradigms could be implemented. In fact, we use Active Messages in just this way. The Cilk system is implemented on top of Active Messages.

The Split-C [CDG+93] parallel programming language is an attempt to merge some of the features of the data-parallel and message-passing paradigms. As in the message-passing paradigm, Split-C exposes to the user one thread of control for each processor. Unlike the message-passing model, however, Split-C provides a range of somewhat higher-level primitives. Split-C has a global address space and provides the programmer with a variety of operations which operate on global data. How global data is distributed among the processors is totally up to the user. Naturally, Split-C can use the global-memory primitives to implement most message-passing algorithms.

Split-C can also express many of the applications that can be expressed in data-parallel languages. As in data-parallel languages, Split-C contains operations which act on an entire global data structure as a whole. (e.g. summation). These system-supplied functions are not really necessary, however, since these operations could easily be built by the user out of the memory-access primitives. In Split-C the user can write data-parallel programs by directly specifying what computation each processor should perform on its portion of the data. Using this method the user can write more efficient programs than can be written in a standard data-parallel language. For example, given a sequence of operations on parallel data, a data-parallel language typically performs each operation on all the data before moving on to the next operation. But, it is often more efficient to do a series of operations on one slice of the data before moving on to the next slice and repeating the same operations. Whereas Split-C gives the programmer this control, data-parallel languages in general do not. A Split-C programmer can use such techniques to exploit locality and to write codes that are optimized for the sequential processors that make up the parallel machine.

Split-C provides a programmer with more power and flexibility than typical data-parallel or message-passing languages, but it still has the drawback that it best suited mainly for static programs. As with message-passing languages, Split-C allows dynamic programs to be written, but this requires the programmer to write at a lower level, and thus the user is back to programming protocols, not the application. In all the models we have seen so far, the user decides, either at compile time or early in the execution of the program, where all his data is going to reside, and how the computation is going to be spread amongst the processors. None of these systems deal well with dynamic programs where it is not known in advance how the computation will unfold and what the data will look like. None of these systems are able to take advantage of control parallelism, which a system must be able to do in order to execute dynamic programs. In order to execute such programs we need a very different system.

Multithreading

In order to execute unstructured programs, we need a system that can take advantage of control parallelism. Data-parallel models present the user with a single thread of control. Models based on message passing increase this to one thread of control per processor. To take full advantage of control parallelism, we must virtualize the number of threads of control so that whenever the program discovers several independent tasks, those tasks can be executed in parallel, each with its own thread of control. When using such a multithreaded programming model, the runtime system must schedule these tasks and dynamically spread them across the machine in order to load balance the computation.

The most ambitious of the multithreaded languages are the implicitly parallel languages, such as Id[Nik91]. In these languages the programmer expresses his algorithm at a high level without any mention of parallelism. Then, a sophisticated compiler automatically breaks the program up into a fine-grained multithreaded program. In this model every memory reference and every interprocedural communication is a potential nonlocal, long-latency operation, which leads to small thread lengths and frequent communication. Executing efficiently under these conditions requires a platform with cheap thread creation and scheduling, as well as a high-bandwidth, low-overhead communication infrastructure. There are several machines which have been designed with these characteristics in mind, such as HEP [Smi78], Tera [AAC$^+$92], and dataflow machines such as Monsoon [PC90] and the EM-4 [SKY91]. Most existing machines do not have these characteristics, however. As analysis techniques improve, compilers are becoming better able to exploit locality in these programs and to increase the thread lengths. These improvements may eventually allow these programs to run on traditional machines efficiently, but at present implicitly parallel programs running on traditional machines incur significant overheads.

More common are the explicit multithreaded languages. In these systems the user must explicitly specify what can be done in parallel. There are a wide range of such multithreaded systems [CRRH93, CGH94, CAL$^+$89, CD88, CSS$^+$91, EAL93, FLA94, Hal85, HWW93, Kal90, KC93, KHM89, Nik94, TBK93]. These systems provide the programmer with a means to create, synchronize, and schedule threads. In order to reduce the overhead of the program, thread creation, synchronization, and scheduling is typically done by user-level runtime system code, without the involvement of the native operating system. Since the user can cheaply and dynamically spawn off tasks as they arise, and then let the runtime system take care of all the details of executing these tasks, these systems make it easy for the user to take full advantage of the control parallelism inherent in many programs.

These systems differ in how the user specifies threads, in what support is provided for shared objects, and in how the scheduling and load balancing of the computation takes place. One thing these multithreaded systems all have in common is that none of them provide performance guarantees for execution time, space or communication, as some of the data-parallel languages do.

As mentioned earlier, the main goal of this work is to make it easier for a programmer to efficiently implement dynamic parallel algorithms. In this section we have described desirable features of other systems. Let us now summarize what characteristics we would like our system to have:

- **Minimize the gap between applications and languages:** A programmer should focus on his application, not on protocols. Details that do not have to do with the application should be hidden from the programmer.

- **Provide predictable performance:** There should be no surprises. A programmer should have a good idea of how his application will perform, and how it will scale, before he even executes it.

- **Execute efficiently:** The system should not add too much overhead to the execution of a user's program. The performance of the parallel code, when run on one processor, should be comparable to the best serial code when run on the same processor.

- **Scale well:** When possible, increasing the number of processors used by a program should improve the performance proportionally.

- **Portable:** Our system should be portable to a variety of machines, from serial machines, to small-scale SMPs, to networks of workstations, to large-scale message-passing machines.

- **Leverage existing codes:** We would like to convert a serial program to a parallel program with the least effort possible. We should therefore be able to include standard serial code in our parallel program, so that only the parts to be parallelized need to be rewritten.

- **Be expressive:** We should be able to implement a wide variety of applications in our system.

1.2 The Evolution of Cilk

This thesis describes the evolution of the Cilk system, which is outlined in Table 1.1. This table shows the various versions of Cilk, and the key features of each version. In this thesis we focus on the key ideas of each version of Cilk and we do not attempt to describe all the details needed to write an application in Cilk. Those interested in using the system are referred to the Cilk Reference Manual [BFJ$^+$95].

System	Novel Features
PCM	Basic multithreaded system
Cilk-1	Provably good scheduler
Cilk-2	Call/return semantics
Cilk-3	Shared memory
Cilk-4	Inlets + aborts

Table 1.1: Evolution of the Cilk System.

This work began with a basic multithreaded programming system called the Parallel Continuation Machine, or PCM for short. The intent of this system was to provide a simple system with which a user could write efficient multithreaded programs. This system provides the user with the basic primitives for creating threads and specifying how the threads communicate and synchronize. The PCM system hides from the user the details of scheduling and executing these threads, thus simplifying the task of writing explicit multithreaded applications. The user writes his multithreaded application as a set of threads, wired together in continuation-passing style. Each thread is a nonblocking piece of code that that may contain calls to standard C functions. After performing some computation, a thread sends its result to another thread, potentially enabling that thread to begin work on the rest of the computation.

A simple preprocessor takes the user's code, which consists of definitions of threads and C functions, and converts it into standard C code with calls to PCM runtime primitives. The system represents a thread using a data structure called a *closure* which contains a description of a thread and all its arguments. A closure is a self-contained unit containing all the information needed to execute an instance of a user's thread, and therefore the computation described by a closure is free to be executed on any processor. The runtime system uses a randomized work-stealing scheduler [BS81, Hal84, BL94] to schedule and load balance the computation. A processor typically works locally, mimicking the serial execution order. When a processor runs out of work, it chooses a processor at random and steals a ready closure from the chosen processor. This work-stealing scheduling strategy tends to provide good load balancing without requiring excessive communication [KZ93, RSAU91, ZO94]. We wrote several applications in PCM, including a ray tracer based on the serial POVRAY program [POV93], and a protein-folding code [PJGT94], which is still being used to investigate various models of protein formation. The PCM system performed well on these applications, achieving nearly linear speedup without adding significant overhead. This initial system showed us that we could easily build a powerful multithreaded system. The PCM system is described in Chapter 2. Although little actual code from the PCM system remains in the current Cilk system, many of the concepts used in PCM have persisted throughout the various Cilk systems.

Having shown that an efficient multithreaded system was buildable, we then focused on providing a more rigorous foundation for the system. Although the applications we coded in the original PCM system achieved nearly linear speedups, it was apparent that not all programs could be executed with such good results. Our desire was to have a system which achieved good performance over as wide a range of programs as possible. We also wanted to have a system which made it clear what properties a program must have in order to achieve good performance. The Cilk-1 system meets these goals.

The Cilk-1 system is an enhanced version of PCM which gives the user predictable and provably good performance. The theoretical work of Blumofe and Leiserson [BL94] presented a work-stealing scheduling algorithm which, for a class of well-structured programs, is provably efficient. By adding some structure to our programs and making a change to our scheduler, we were able to extend the proofs in [Blu95] to cover our scheduler as well. With these changes Cilk's work-stealing scheduler achieves space, time, and communication bounds all within a constant factor of optimal. The two parameters which predict how well a program will perform are the "work" and "critical path" of the program. The work of a program is the time it would take one processor to execute the program, and the critical path is the time it would take an infinite number of processors to execute the program. We extended our system to measure the work and critical path when a user runs a program. With these two measures a user is able to understand why the program performed as it did, and the user is also able to predict the performance of the program on machines of different sizes. We describe the Cilk-1 system in Chapter 3.

It was while designing the Cilk-1 system that we wrote our largest application, the ⋆Socrates chess program. Up to this point we had no application that made full use of the complicated control structure allowed by our system. In order to showcase the power of our system and to point out any improvements that the system needed, we wanted to build a challenging, dynamic application that could not easily be implemented in other parallel programming paradigms. Computer chess is such an application. Our chess program uses large global data structures, is nondeterministic, and performs speculative computations, some of which are aborted. This work was in part a natural follow on of StarTech [Kus94], a parallel chess program designed by Bradley Kuszmaul which had the scheduler and search algorithm intertwined. We wanted to show that the scheduler and search algorithm could be separated, thereby greatly simplifying the programmer's job, without sacrificing performance. Our work on ⋆Socrates led to several enhancements to the runtime system which were included in the Cilk-1 system. Despite the use of low-level Cilk-1 features that void Cilk's performance guarantees, ⋆Socrates still achieves efficient, predictable performance over a range of machines. ⋆Socrates remains our flagship program and continues to drive the development of Cilk. Chapter 4 describes the ⋆Socrates program and how it influenced the Cilk-1 system.

Although the continuation-passing style required by Cilk-1 allowed a wide range of programs to be expressed, writing programs in this style was quite error prone and tedious. In Cilk-2, our next major release of the system, we focused on making the system easier to program. As a stepping stone towards this goal we first introduced a type-checking preprocessor [Mil95]. Previously, Cilk programs were converted to C via a standard, but simple, macro preprocessor. This preprocessor limited the constructs we could use in the language, occasionally forcing us to expose to the programmer details we would rather have kept hidden. Introducing the type-checking preprocessor allowed us to hide some of these details, thus cleaning up the language. More importantly, the new preprocessor can deduce semantic information about the source Cilk program, thereby allowing us to perform transformations we could not consider previously. One alternative to the type-checking preprocessor would have been to build a full-fledged compiler. A compiler would have allowed us to do everything the preprocessor could do and more. Building a compiler is a significant undertaking, however. This option would have required resources and time that we could not afford, and it would have made the system less portable as well.

The power of our type-checking preprocessor allowed the programmer to write some codes using traditional call/return semantics, thus making Cilk programs substantially easier to write. With this change users can write a parallel Cilk program without dealing with threads or continuation passing. A program previously written as many Cilk-1 threads tediously wired together by the programmer could now be expressed as a single Cilk-2 procedure. A single Cilk-2 procedure can spawn off child procedures, suspend until all the children complete, and then continue executing, perhaps spawning off children again. The new preprocessor automatically breaks these Cilk-2 procedures into several threads which can then be executed using the basic Cilk-1 runtime system. This Cilk-2 style of programming is somewhat more restrictive than that allowed by other multithreaded languages, but we find it simple for the programmer, and sufficient for almost all algorithms we have tried to write. This release, which was a major step towards making Cilk suitable for widespread use, is described in Chapter 5.

The improvements made for the Cilk-2 system made Cilk programs easier to write, but they did not increase the range of programs which could be written. One of the biggest drawbacks of the Cilk systems described so far is that it is difficult to write applications where significant amounts of data need to be shared throughout the computation. With the Cilk-2 release, for those applications for which Cilk was "well suited" it was now fairly easy to write a Cilk program and get good, predictable performance. The applications for which Cilk-2 was well suited, however, were somewhat limited, mainly consisting of applications which could be expressed using a tree-like algorithm where the nodes of the tree were fairly independent. The different parts of the computation must be fairly independent of each other because in Cilk-2 the only way to share data throughout the computation is to explicitly pass the data from procedure to procedure. For any reasonably large data set a programmer would be forced to go outside of Cilk and implement a shared data structure, probably by taking advantage of the low-level features supported by the particular platform on which the code was being developed. To solve this problem and increase the range of programs which could be expressed in Cilk, the aspect we focused on for the next release of Cilk was adding shared memory support.

The Cilk-3 release includes a shared-memory system implemented totally in software. Rather than attempting to build a shared-memory system that can solve all problems, we focused on building one that would be sufficient for the types of problems that are naturally expressed in a multithreaded programming environment such as Cilk. Instead of using one of the consistency models derived from sequential consistency, we used our own, relaxed, consistency model. Our model, which we call "dag consistency," is a lock-free consistency model which, rather than forcing a total order on global-memory operations, instead ensures only that the constraints of the dag are enforced. Because dag consistency is a relaxed consistency model, we were able to implement coherence in software efficiently for Cilk. With this shared-memory system we are able to express applications, such as matrix multiply and Barnes-Hut, which make use of global data structures The definition of dag consistency, and our implementation of it for Cilk are described in Chapter 6.

Cilk-4, the last release of the Cilk system that I will describe in this thesis, is intended to remedy a deficiency in the Cilk-2 language. When we designed Cilk-2 and added support for procedures with call/return semantics, we were able to rewrite almost all existing programs using the new, simpler, Cilk-2 syntax. The only application which could not be expressed using the Cilk-2 syntax was $\star$Socrates, due in large part to the complex control structure of the parallel search algorithm. Specifically, $\star$Socrates generates speculative work which may sometimes be killed off. Therefore, unlike all other programs we have written, the amount of work performed by a run of the chess program depends on the order in which the user's threads are executed. The Cilk-2 syntax does not give the user enough control over the execution of his code to write an efficient speculative algorithm, so the chess code is still written with Cilk-1 style syntax.

For the Cilk-4 release, which is currently under development, we have proposed an extension to Cilk that should allow the chess program to be written without resorting to any of the lower level Cilk-1 constructs. The extension allows the programmer to specify a piece of code called an *inlet* when spawning a child. This inlet code is run immediately after the child finishes. The search routine in $\star$Socrates will use inlets to receive the result of one search, and depending on the result the search routine may spawn off a more precise search, may update the parameters for other searches, or perhaps may abort a group of searches. The proposed extensions also support an abort primitive which will allow a procedure to abort all of the its spawned children. Currently, $\star$Socrates must implement the abort mechanism as user-level code, which is quite tedious, this new feature will allow this code to be removed from the user program. An overview of the proposed changes are given in Chapter 7.

The design of the Cilk system is an ongoing project. The final chapter of this thesis describes some of the improvements we would still like to make to the system, and gives some concluding remarks.

History of Cilk

We conclude this section with a description of how work on Cilk has progressed over the last two and a half years. The preceding paragraphs gave an overview of the technical evolution of Cilk. The following give a more historical view of how the Cilk project came about.

The original PCM system grew out of work begun by Michael Halbherr. Michael was doing research on parallel I/O, and in order to perform some experiments, he implemented a simple system for executing parallel programs. This parallel runtime system became interesting in its own right, and in the middle of 1993 Yuli Zhou and I began working on this system as well. All efforts on parallel I/O were soon forgotten, as we focused solely on the parallel runtime system. This multithreaded runtime system eventually became the PCM system.

While we were improving the PCM system and writing applications that used it, we began interacting with Professor Charles Leiserson and his students, who were working on several related projects. The first of these related projects was theoretical work by Robert Blumofe and Charles Leiserson on scheduling multithreaded applications. The second was StarTech, a parallel chess program built by Bradley C. Kuszmaul, another of Leiserson's students.

In April of 1994, Charles suggested that we join forces, beginning by implementing a parallel chess program in PCM. Unlike StarTech, which intertwined the search code and the scheduler, this new program would build the chess search completely on top of the general-purpose PCM runtime system. In May, we obtained a serial chess program from Heuristic Software and began porting it to our system. During June, Don Dailey, then of Heuristic Software, joined us at MIT to work on the program's chess knowledge, and at the end of June we entered the program in the 1994 ACM International Chess Championship, where, running on a 512-node CM-5, we finished third.

After the chess tournament, Keith Randall joined the Cilk team and we made the changes necessary to incorporate some of the theoretical results into the system. The resulting "provably good" system was renamed Cilk. Then, in the fall of 1994, we recruited several undergraduates (Greg Hudson, Rob Miller, Richard P. Tauriello, Daricha Techopitayakul, and John Yu) to program in Cilk. This experience helped us learn more about programming in Cilk, and two of these students, Rob and Richard, eventually wound up contributing to the Cilk system itself. During late 1994, we focused on making Cilk easier to use and began designing the Cilk-2 system. Also during 1994, Robert Blumofe, with the help of two undergraduates, Phil Lisiecki and Howard Lu, began working on the fault tolerant version of Cilk for networks of workstations.

In 1995, Cilk progressed on many fronts. In January, Matteo Frigo, who had recently joined the Cilk team, completed reworking much of the Cilk code to make it more portable. Rolf Riesen of Sandia National Laboratories later ported this reworked version of Cilk to the Intel Paragon, and we ported the chess program to the Paragon as well. Don Dailey again joined us to work on the chess aspects of *Socrates, and in May 1995, we ran on a 1824 node Paragon in the 1995 World Computer Chess Championship, finishing second. In June 1995, the Cilk-2 implementation, which had been fairly stable for several months, was officially released. Early in 1995 we had begun working on shared memory support for Cilk. The implementation of shared memory was fairly stable by September, and this implementation is the one that we describe in this thesis. Lastly, during the second half of the year we worked out the design of inlets and aborts for the Cilk-4 system. This system is currently being implemented.

Chapter 2

The PCM System

This chapter describes the PCM system. PCM, the precursor to Cilk, is a simple multithreaded runtime system based on continuation-passing threads. This system was our initial attempt to produce a system with which a user could write efficient multithreaded programs. The PCM system grew out of a system initially designed by Michael Halbherr for research on I/O. Michael Halbherr, Yuli Zhou and I implemented the PCM runtime system. Yuli Zhou implemented the preprocessor used by the PCM system. The protein folding code described in Section 2.4 was written by myself based on discussions with Vijay Pande of the Center for Material Sciences and Engineering at MIT.

2.1 Introduction

This chapter presents the parallel continuation machine (PCM), a parallel runtime system designed to efficiently execute dynamic, multithreaded programs on today's message-passing architectures. We will first concentrate on explaining the key ideas underlying the implementation, and then demonstrate how they give rise to extremely efficient parallel programs via two real-world examples.

Parallel programs can be classified along several dimensions, such as grain-size, communication regularity, and whether the execution depends on runtime data. We believe that existing programming models, such as data parallel programming and explicit message passing, have been successful in addressing the needs of programs with simple static

Much of the work described in this section was reported on by Michael Halbherr, Yuli Zhou and myself in an earlier paper [HZJ94].

communication patterns. For these programs it is usually possible to carefully orchestrate communication and computation to statically optimize the overall performance.

On the other hand, it proves far more difficult to find static solutions leading to high machine utilizations for parallel applications whose communication patterns are either highly irregular or dependent on dynamic information. In this work, we are mostly interested in investigating the needs and characteristics of these classes of programs, which must rely on runtime mechanisms to enable efficient solutions.

Multithreaded computation models have typically been proposed as a general solution to exploit dynamic, unstructured parallelism. In such a model, dynamically created instances of sequential threads of execution cooperate in solving the problem at hand. To efficiently execute such an application, it is necessary to have efficient runtime thread placement and scheduling techniques. Although finding the optimal thread placement is known to be an NP-hard problem [GJ79], it is possible to implement schedulers based on simple heuristics that achieve good machine utilizations at reasonable cost. These heuristics usually work well for a broad class of applications, making it possible to implement the scheduling and placement task as a fairly generic service that resides at the core of the runtime system.

Several research machines, such as HEP [Smi78], the Monsoon dataflow system [PC90], and the forthcoming Tera machine [AAC+92], have been designed expressly to support multithreaded computations. These machines provide highly integrated, low overhead, message interfaces as well as hardware support for scheduling and synchronization. Disregarding the debate of whether such machines are commercially or technically viable, the problem of programming most of the current parallel machines, which have no special hardware support for multithreading, still remains. The programming challenge, in view of the above difficulties, is to minimize network communication and to provide longer sequential threads to offset the runtime scheduling and synchronization overhead.

The static set of sequential threads making up the multithreaded program can either be generated *implicitly* by a sophisticated compiler, or *explicitly* by the programmer. Programming languages advocating the implicit style, such as Id [Nik91] and Sisal [MSA+85], usually take a high-level, functional, description of the actual problem, extract the available parallelism from the declaration and partition it into sequential threads. While implicit programming languages simplify the programming task for some problems, other problems are difficult to efficiently express in a functional style. In addition, some of these systems fail to produce threads long enough to adequately amortize the overhead introduced by a dynamic execution model.

To help reduce network communication, the execution of threads must exhibit communication locality. This requirement precludes scheduling policies such as round-robin or random placement, but favors solutions such as *work stealing* where all threads are created locally per default, but may later migrate to other nodes upon demand.

The PCM model presented in this chapter is aimed at solving the aforementioned problems. The intended target architectures are simple message-passing machines which support the implementation of low-overhead communication layers such as *Active Messages* [vECGS92]. We do not assume any additional hardware support.

We have provided C language extensions where threads can be specified along with conventional sequential C code. A program consists simply of a collection of *threads*, which are pieces of sequential code which are guaranteed to terminate once they have been scheduled. Threads represent the basic scheduling units that can be executed on any processor. By exposing threads in this way, we can experiment with various static and dynamic scheduling policies to optimize the overall machine utilization. In particular, we have found two strategies which can have a great effect on the performance of a program.

- The scheduler uses work stealing as its default policy. This strategy creates patterns of locality in which threads can pass arguments through local memory most of the time, rather than across the network, thereby greatly reducing the communication frequency.

- A thread can be made to directly transfer control to another thread, this mechanism is similar to the control transfer mechanism used to implement tail-recursion in serial codes. This mechanism bypasses dynamic scheduling entirely, thus avoiding all of its associated costs.

We show the effect of these optimizations in Figure 2-1. As this diagram suggests, there are three communication levels, namely register communication, memory communication, and network communication. Preferably, we would like to transfer data through registers as much as possible, but the very nature of a dynamic execution model will force us to resort to memory communication or, even worse, to network communication. Note that an application increases the working set whenever it exposes additional parallelism, making

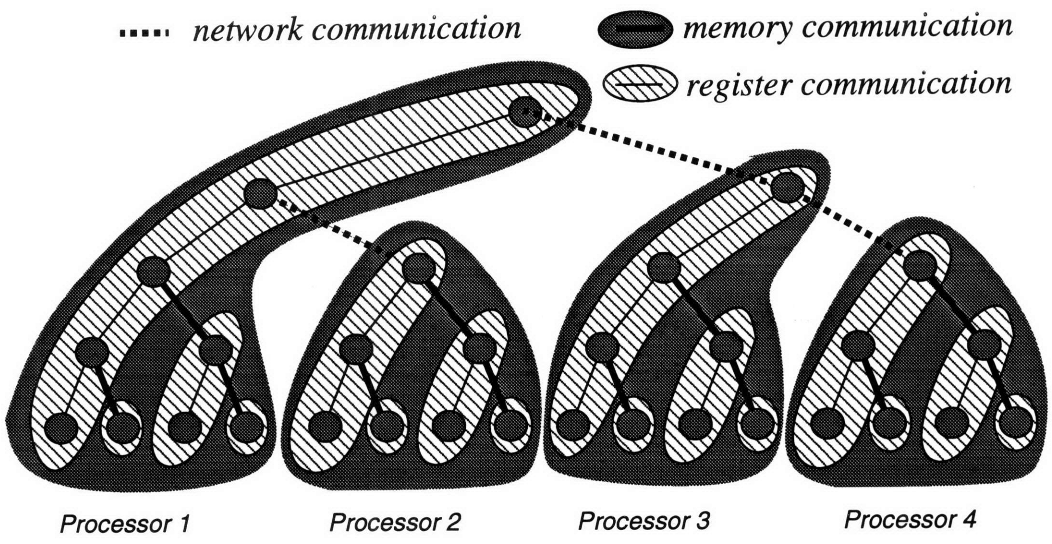

Figure 2-1: Maximizing Communication Locality

it impossible to keep the entire working set in registers. The implementation goal will be to provide an optimal compromise between increasing the sequentiality of the application to increase its locality and exposing enough parallelism to enable dynamic load balancing. The work stealing scheduler will then attempt to minimize the work migration frequency, thereby minimizing network communication.

The rest of this chapter is structured as follows: Section 2.2 introduces the PCM thread specification language and presents its components. Section 2.3 introduces a cost model, intended to clarify the costs involved in dynamic execution. Section 2.4 contains in depth explanations of two applications implemented with the PCM package. Section 2.5 concludes the chapter.

2.2 The Parallel Continuation Machine

The parallel continuation machine implements an SPMD programming model, where all processors keep their own local copies of the entire code. The execution itself is completely asynchronous, meaning that each node may execute entirely different pieces of the program.

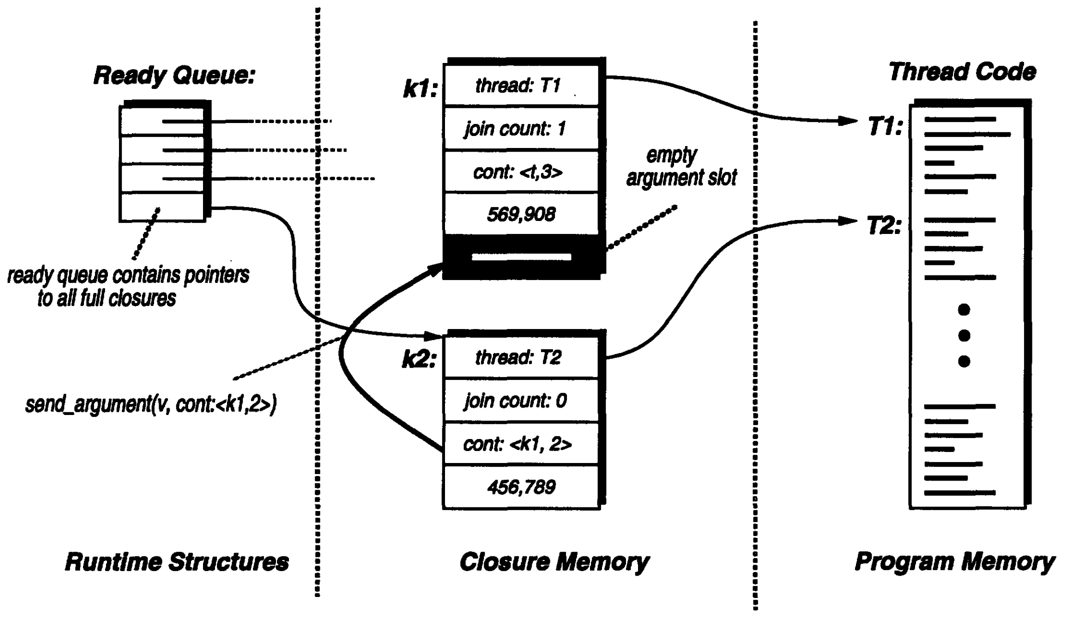

Figure 2-2: Elements of the PCM Model

2.2.1 Elements of the PCM

A PCM program consists of a collection of *threads* that cooperatively solve a single problem. Statically, a thread identifies nothing more than a sequence of instructions, written in the machine language of the processor. At runtime, an application can create arbitrary numbers of dynamic instances of a static thread, each with its own set of arguments.

The PCM thread specification language, which is explained in section 2.2.2, allows the programmer to define threads and to specify how threads communicate with each other. All machine-specific execution details, such as dynamic load balancing or the mechanism required to enable transparent inter-thread communication, are part of the PCM runtime system and do not have to be specified by the user. By taking care of these details, the system simplifies the job of writing explicit multithreaded applications without sacrificing the programmer's power for experimentation.

The key elements of PCM's execution environment are illustrated in Figure 2-2. New dynamic instances of threads can be created by making closures. *Closures* form the contract between the application code and the runtime system. Closures contain a pointer to the thread code and all the arguments needed to execute the code. The thread code may include calls to arbitrary C procedures. A thread is ready to execute when its closure is *full*; in other words, when the closure has all its arguments. Full closures are passed to the scheduler, which keeps them in a *ready queue* and will later schedule them for the encapsulated threads to run. All threads in PCM are *nonblocking*, which means that once a thread begins execution, it can run to completion without suspending.

In order to enable points of synchronization between threads, a closure can be created with some of the arguments missing. These arguments will be filled in later with a value sent by another thread. A thread supplying an argument to a non-ready closure must obtain a reference to where the argument is to be sent. Such references will be called *continuations*; a continuation is just a pointer to a closure plus an integer offset into the closure. Note that we are slightly abusing the term continuation, which in sequential computations is used to refer to *the rest of the computation* as seen from a particular program control point. In parallel programs there is usually no such thing as "the rest of the computation" from a single control point, but the idea is the same.

In order to detect when a closure becomes full, every closure has an additional slot containing a *join counter* that indicates the number of missing arguments the closure has. The join counter is initialized with an integer equal to the number of missing arguments at closure creation time, and decremented each time the closure receives an argument. A closure is given to the scheduler when the join counter reaches zero.

2.2.2 The Thread Specification Language

The thread specification language, called Threaded-C, was implemented for the PCM system as an extension to C. In this language a program consists of C functions, written just as in standard C, and threads, which are marked by the specifier **thread**. A preprocessor expands the threads into C functions, while copying the rest of the C code literally. The resulting C program can then be compiled and linked with the PCM runtime library.

Runtime primitives are called by threads to create closures, send arguments, and transfer closures to the scheduler:

- `make_closure` (*Thread*, arg_1, ..., arg_n) allocates a closure of size $n+2$, and returns a pointer to the closure. The two additional slots are reserved for the code pointer, which is initialized to *Thread*, and the join counter. Closures can be created without specifying all the arguments, in which case missing arguments are indicated by "_". The join counter is implicitly initialized to the number of missing arguments.

- **post** (k) hands the closure k over to the scheduler. Only a full closure can be posted inside the thread that created it. Closures created with empty slots will be posted later by a **send_argument** when the join counter reaches zero.

- **send_argument** (c, v) sends the 32-bit value v to continuation c and decrements the join counter of the target closure. The closure is posted if the join counter becomes zero. Continuations have the type **Cont** and contain two fields: a pointer to a closure, and an integer offset within the closure. A new continuation can be constructed, for example, using the expression **cont{k1, sum:x}**, where **k1** is a closure pointer and **sum:x** a symbolic reference to offset for argument **x** of a **sum** thread.

An example PCM program to compute the Fibonacci function is shown in Figure 2-3. A sequential fib function makes two recursive calls to fib and then sums the results together. Our parallel version first creates a **sum** closure which will receive two results and add them together. It then creates and posts two full **fib_main** closures for the two recursive calls of fib. It gives each of these closures a continuation pointing to a slot in the **sum** closure where the result should be sent.

When the PCM preprocessor is run on the fib code, the preprocessor expands each of these threads into a C function that takes a single argument, namely a closure structure. The preprocessor inserts code into the function to fetch all of the user's arguments from the closure before starting execution of the code specified by the user.

2.2.3 Executing a PCM Program

Figure 2.2.3 illustrates the sequence of events when the Fibonacci program runs on a single processor. When there are multiple processors the only difference is that full closures may be migrated to ensure a balanced load across all available processors. These scheduling issues will be discussed in more detail in section 2.3.

The execution of a PCM program can be divided into three phases: initialization, computation, and termination. The first and third phases are usually very short, with the computation phase constituting the bulk of the overall execution.

During the *initialization* phase, shown in frame (A) of Figure 2.2.3, the program creates two closures. One closure specifies the start of the computation, in this example a ready instance of **fib_main**, and the other closure specifies the actions to be taken when the

```
thread sum (Cont parent, int x, int y) {
    send_argument (parent, x+y);
}

thread fib_main (Cont parent, int n){
    if (n<2) send_argument (parent, n);
    else
    {    closure k1, s1, s2;

         k1 = make_closure (sum, parent, _, _);
         s1 = make_closure (fib_main, cont{k1,sum:x}, n-1);
         s2 = make_closure (fib_main, cont{k1,sum:y}, n-2);
         post (s1);
         post (s2);
    }
}
```

Figure 2-3: A PCM program to compute Fibonacci

computation ends, in this example a non-ready instance of a special thread called `top`. This `top` thread is supplied by the runtime system and its responsibilities are to terminate the computation and to print the result(s). The `top` closure can be instructed to expect any number of results and must be the last closure to execute.

During the *computation* phase, the scheduler enters a perpetual loop. It pops a full closure from the ready queue and then calls the thread function, such as `fib_main` or `sum`, specified in the closure with the closure pointer as its only argument. Frames (B) through (H) of Figure 2.2.3 show snapshots of the machine state, one after each closure has executed. For example, in frame (B) the thread `fib_main` with argument 3 has just terminated. It created three closures: two full ones for `fib_main` with arguments 2 and 1 respectively, and a closure for `sum` waiting for two arguments. The full closures were immediately posted. These full closures contain continuations which point to the slot in the sum closure where they will send their results. Similarly, the sum closure contains a continuation pointing to the `top` closure, which is where its result should be sent.

During the *termination* phase, shown in frame (I), the top thread is run. It will print the result of the computation and then cause the scheduler to exit the loop, thereby terminating the computation. For multiprocessor computations, after the `top` thread executes on a processor, that processor signals the schedulers on other processors to exit their work loops.

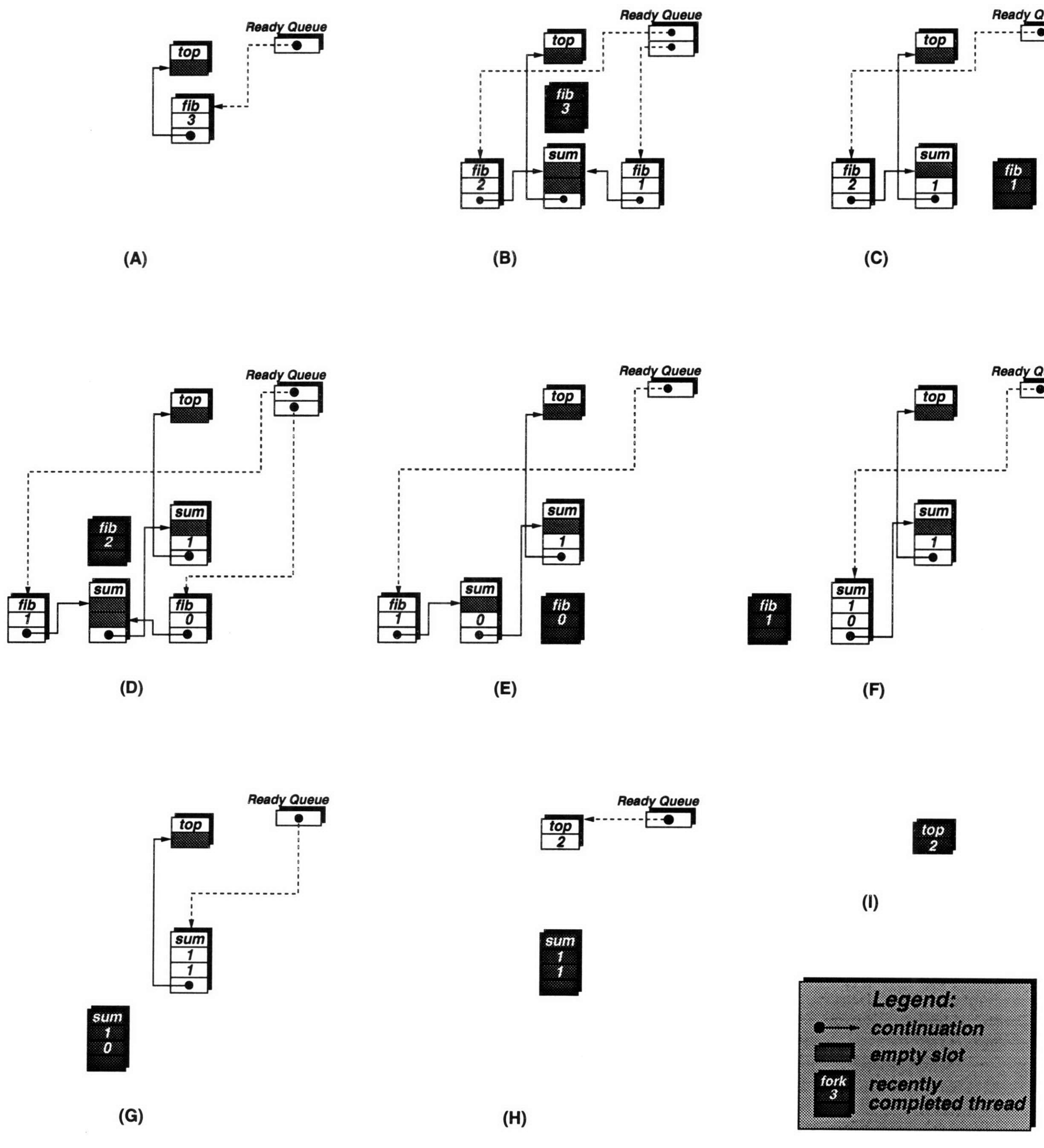

Figure 2-4: Snapshots of the state of the PCM system after each thread completion.

2.2.4 Tail Calls

As a performance optimization a thread may directly call other threads via the following runtime primitive:

- **`tail_call`** (*Thread*, $arg_1, \ldots, arg_n$) executes the thread *Thread* without the overhead of creating a closure and calling the scheduler. This function must be called as the last action of a thread and must be called with no missing arguments.

A tail-call represents a more efficient invocation of a thread, avoiding any of the the dynamic execution overheads incurred otherwise. In the example shown in Figure 2-3, the thread `fib_main` creates two full closures `s1` and `s2`, packs the arguments into the closures and then posts both closures before releasing control and returning to the scheduler. After receiving control, the immediate action of the scheduler will be to pop the `s2` closure and to call its thread function `fib_main`, which will unpack the `s2` closure prior to doing actual work. We can avoid this costly detour through the scheduler by rewriting the else clause of `fib_main` to use a tail-call as follows:

```
{    closure k1, s1;
     k1 = make_closure (sum, parent, _, _);
     s1 = make_closure (fib_main, cont{k1, sum:x}, n−1);
     post (s1);
     tail_call (fib_main, cont{k1, sum:y}, n−2);
}
```

To implement this mechanism, the thread preprocessor actually expands a thread into two C functions: a *general entry version*, which is what we described above, and a *fast entry version* which receives all arguments directly. A `tail_call` is thus converted into a standard C function call to the fast entry version. The actual performance improvements obtained with the tail-call mechanism can be quite impressive, especially for fine-grained applications, such as the Fibonacci example, where the performance improved by almost twenty-five percent. The effects of using the tail-call mechanism show up in Figure 2-1 as the register communication between threads.[1]

[1]The name of the `tail_call` primitive was later changed to just `call`. The primitive `tail_call` was then restricted to the case of recursive calls, in which case the preprocessor is able to implement the recursive call simply by inserting a jump to the beginning of the function.

2.2.5 Passing Vectors in Closures

An additional mechanism provided by the thread language allows vectors to be passed in closures. One of these vectors may even be of arbitrary length. These vectors are passed by value and can be referenced within a thread like any other local variable. The one vector argument which is allowed to be of arbitrary length needs to be specified as the last argument, to make sure it is packed into the tail of the closure.

`thread` *foo* (..., *type vect1*[10], ..., *type vect2*[])

declares a vector argument *vect* of type *type*. It is the responsibility of the creator of the closure to initialize the vector. For example, the expression

`make_closure` (*foo*, ..., *vect1*, ..., *vect2* = [*size*])

creates a closure for *foo*, dynamically defining *vect2* to consist of *size* entries. In addition, *vect1* and *vect2* will be declared to be pointers initialized to the zeroth word of the corresponding vector arguments. These pointers must be used subsequently to initialize the vectors, which would otherwise be left empty. If we use "_" instead of a *vector name* when allocating a closure, then the vector is intentionally left empty, and its *size* will be added to the initial join count.

2.3 Scheduling PCM Threads on a Multiprocessor

This section introduces a cost model for PCM in order to motivate the work stealing scheduling system that we implemented for PCM on the CM-5. The CM-5 is a massively parallel computer consisting of 32MHz SPARC processors wired together by a fat-tree interconnection network [LAD$^+$92]. All communication mechanisms required to implement the work stealer and the inter-thread communication have been built using a version of Active Messages [BB94].

As Figure 2-5 illustrates, we equate useful computation with what a sequential program would have to do and classify everything else, such as communication, synchronization, and dynamic scheduling as additional overhead. The goal of this classification is to study the factors that determine the efficiency of a parallel computation with respect to its sequential counterpart.

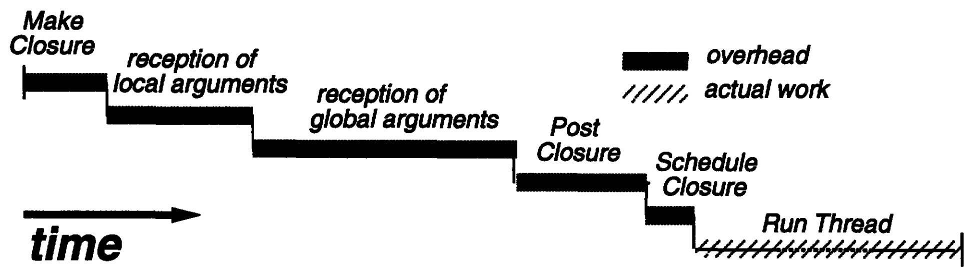

Figure 2-5: Anatomy of a PCM Thread

To simplify the analysis, we will ignore any idle time and assume that each processing element is executing either a thread or one of the overhead tasks depicted in Figure 2-5. Under this assumption we can reduce the analysis to that of an average thread. The corresponding efficiency, ϵ, defined as the fraction of the overall execution time actually spent executing the useful computation, can then be computed

There are three important ratios needed for the computation, reflecting the effects of tail-recursions (π_1), global send arguments (π_2) and closure migration (π_3) on the overall efficiency. π_1 equals the fraction of threads not called by the tail-call mechanism, π_2 equals the fraction of arguments that have to be sent across the interconnection network and π_3 equals the fraction of closures that migrate from one processing element to another. R_t defines the average run length of a PCM thread and k defines the threads arity.

The overhead in executing a thread can be broken into the following pieces:

Make Closure (M_c): At thread creation time, a closure must be allocated and the thread pointer and join-counter must be initialized ($M_c \approx 10$ cycles).

Local Send Argument (S_l): A Local send argument is fairly cheap and reduces to a simple memory-to-memory transfer plus an additional check to see whether the closure has become ready for execution ($S_l \approx 10$ cycles).

Global Send Argument ($S_l + T_a$): For arguments that must be sent across the network, we have to add an additional overhead factor T_a (≈ 100 cycles) to the constant costs of S_l to account for the transfer costs on the sending and receiving sides.

Post closure ($P_c + \pi_3 \cdot T_c$): After receiving all of its arguments a closure is posted and becomes subject to dynamic scheduling ($P_c \approx 10$). If migrated to a remote processor, additional transfer costs of T_c (≈ 500 cycles for a closure consisting of eight words) need to be charged in addition to the constant cost P_c.

Schedule closure (S_c): The cost for transferring control to the thread at the beginning of its execution and back to the scheduler after its termination is S_c ($\approx$ 15 cycles).

With these definitions, we can define the efficiency of a thread as:

$$\epsilon = \frac{R_t}{\pi_1 \cdot (M_c + k \cdot (S_l + \pi_2 \cdot T_a) + (P_c + \pi_3 \cdot T_c) + S_c) + R_t} \tag{2.1}$$

With the communication costs T_a and T_c ranging in the hundreds of cycles, it becomes imperative to reduce both π_2 and π_3 in order to avoid disappointing efficiencies. π_1, on the other hand, cannot be reduced to arbitrarily small values, thus the only remaining alternative to amortize the non-transfer related overhead is to increase the thread run length R_t.

We can simplify equation (2.1) by assuming a typical value of $k = 2$ arguments per thread, and by assuming that the tail_call optimization is not used (ie. $\pi_1 = 1$). We also make the reasonable assumption that $\pi_2 = \pi_3$, which just says that the percentage of `send_arguments` which are nonlocal is the same as the percentage of closures which are migrated. We can then transform equation (2.1) into

$$\epsilon = \frac{R_t}{55 + 600\pi_3 + R_t} \tag{2.2}$$

This shows that the efficiency will depend on R_t, the average run length, and on π_3, the percentage of closures migrated. The scheduler can effect only π_3, so an important job of the scheduler is to minimize π_3.

Implementation

To achieve minimal values for π_3 we have adopted a lazy scheduling policy known as *work stealing*. In such a system, each processor maintains a local queue of full closures, called the *ready queue*. When a closure becomes full, it is posted to its local ready queue. A processor works out of its local ready queue for as long as there are closures in it. When a processor runs out of local work, it will send a *steal request* to a randomly chosen processor. If the processor receiving this steal request has any closures in its ready queue, it will migrate one of the closures to the requesting processor.

In our current implementation, a computation executes locally using a depth-first scheduling policy. This heuristic, which mimics serial execution order, can be expected to result in lower resource requirements for most computations than a breadth-first policy would. Steal requests, on the other hand, will always be served using a breadth-first policy (see Figure 2-1). Such a steal policy can be expected to result in significantly reduced steal frequencies for computations. For example, both examples considered in the remainder of this chapter typically migrated less than one percent of all dynamically created closures. The Cilk-1 system, described in Chapter 3 addresses these performance issues more concretely.

2.4 Two Case Studies

In the following section we present two applications implemented with the PCM thread package. We use these applications to document the efficiency of the PCM system. The first application is ray tracing. In this application the input is a description of objects and lighting in a scene, and the program must produce a high quality image of that scene as seen from a specified point in three space. The second application is protein folding. In this application a sequence of monomers (i.e. an unfolded protein) is input, and the program computes some or all possible foldings of that protein, give certain assumptions about which foldings are legal. All performance experiments described below were performed on the CM-5.

2.4.1 Ray Tracing

The parallel ray tracer presented here is an optimal example to illustrate the virtues of the PCM thread package. First, the task of tracing a complicated picture of reasonable size requires enough computation to justify the use of a powerful parallel processor. Second, the variance in the number of processor cycles required to trace individual rays necessitates the use of a dynamic load balancing scheme to guarantee acceptable utilizations and to ensure scalability. Third, we can break the ray-tracing computation into threads and obtain threads with grain-sizes coarse enough to offset the overhead introduced by a dynamic execution model.

There are several algorithms that can be used to implement a ray tracer. The simplest of all ray tracing algorithms intersects a ray with every object surface and displays the

```
void Trace(){
    int x, y;

    for (y = First_Line; y < Last_Line; y++)
        for (x = First_Column ; x < Last_Column ; x++) {
            pixel = calculate_intersections(x, y);
            write_pixel(x, y, pixel);
        }
}
```

Figure 2-6: Kernel of Sequential Ray Tracer

object whose intersection is closest to the position of the observer. This algorithm is known as exhaustive ray tracing, since it calculates all possible ray-surface intersections. The ray-tracer we used improves upon this basic algorithm. It uses a bounding volume which requires relatively simple intersection calculations, such as a sphere, to enclose more complex objects. If a ray does not pierce the bounding volume then all the objects contained within can be eliminated from consideration. This reduction substantially reduces the average costs of ray surface calculations. This technique is further improved by arranging bounding volumes into a tree hierarchy. In such a scheme a number of bounding volumes could themselves be enclosed within an even larger bounding volume. If a ray does not intersect with a given bounding volume then all the objects in that volume and in all child volumes can be eliminated.

This algorithmic improvement reduces the linear time complexity of exhaustive ray tracing to one which is logarithmic in the number of objects. However, this optimization also creates a large variation in the time needed to trace a ray, making it difficult to find a static work distribution that sufficiently balances the available work.

Ray Tracer Parallelization

To show the power of our thread package as a tool to retarget existing sequential programs for parallel processors, we took the serial POV-Ray package[POV93] that implements the optimized ray-tracing method described above and rewrote its kernel with our thread language. The serial POV-Ray program is quite large, the C source files consist of over 20,000 lines. Fortunately we did not have to modify, or even understand, much of the code. A simplified version of the original sequential kernel can be seen in Figure 2-6. This function

```
thread Join(Cont parent_join, int s1, int s2, int s3, int s4 ) {
    send_argument(parent_join, SIGNAL);
}

thread Trace(Cont parent_join, int sx, int ex, int sy, int ey) {
    if((sx == ex) && (sy == ey)) {
        pixel = calculate_intersections(sx, sy);
        write_pixel(sx,sy,pixel);
        send_argument(parent_join, SIGNAL);
    }
    else {
        closure k1, p1, p2, p3, p4;
        int xoff = (ex - sx)/2;
        int yoff = (ey - sy)/2;

        k1 = make_closure(Join, parent_join, _ , _ , _ , _ );
        p1 = make_closure(Trace, cont{k1,Join:s1}, sx, (sx + xoff), sy, (sy + yoff));
        p2 = make_closure(Trace, cont{k1,Join:s2}, sx, (sx + xoff), (sy + yoff +1 ), ey);
        p3 = make_closure(Trace, cont{k1,Join:s3}, (sx + xoff + 1), ex, sy, (sy + yoff));
        p4 = make_closure(Trace, cont{k1,Join:s4}, (sx + xoff + 1), ex, (sy + yoff +1 ), ey);
        post(p1); post(p2); post(p3); post(p4);
    }
}
```

Figure 2-7: This shows the kernel of the PCM ray tracing code. For simplicity this code assumes that the picture is of size 2^n by 2^n. The actual code does not assume this and is only slightly more complex.

just walks through all the pixels and calls `calculate_intersection` on each of them to determine the value of that pixel. This function is the portion of the code that we rewrote using PCM.

The threaded version of this procedure, which is shown in Figure 2-7, computes the value of a pixel using the exact same function, `calculate_intersection`, that the sequential version used. The `Trace` thread traces a sub-window of the original picture, as specified by the four coordinates in the argument list. The `Trace` thread accomplishes this task by recursively splitting its sub-window into smaller sub-windows until the size of the newly created windows reaches the size of a single pixel, at which point it calls the sequential `calculate_intersection` function to calculate the value of that pixel.

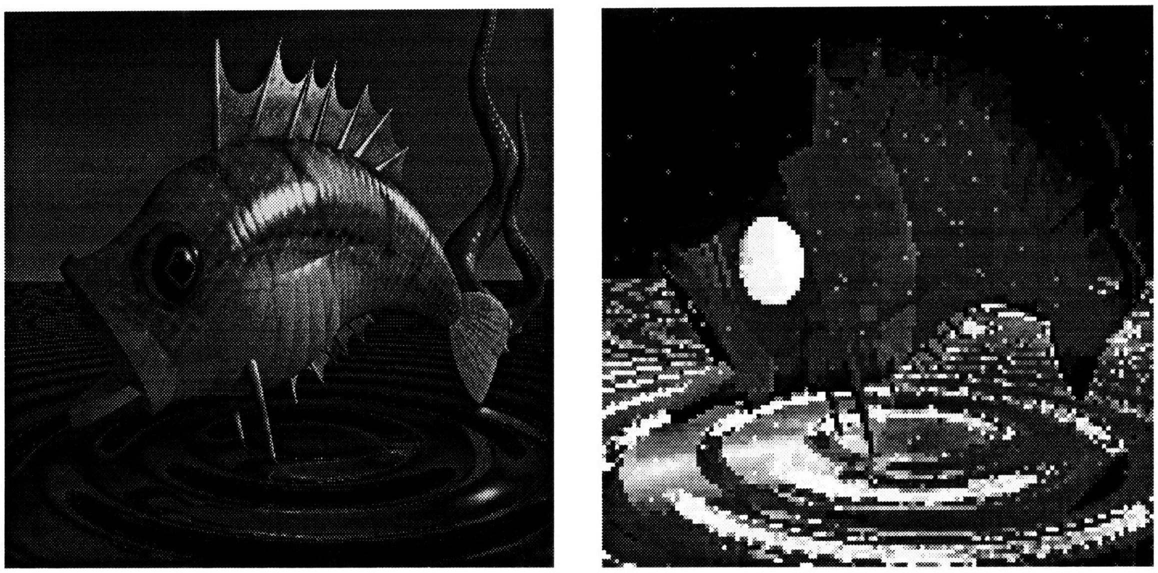

Figure 2-8: The picture on the left shows the ray traced image used in our experiments. The histogram on the right shows how much computation was needed for each section of the picture. Brighter points represent higher workloads, darker points represent lighter workloads.

Ray Tracer Results

To test our multithreaded implementation, we traced pictures of different complexities on various machine sizes. We adjusted the picture size so that enough parallelism would be generated to justify the use of the largest machine configuration used during our test runs.

We first compared the uniprocessor timings of the multithreaded code with those of the original sequential code, both running on the same CM-5 processing node. The results showed no measurable difference between the sequential and the multithreaded timings. To see why there was little difference, we need to look at the thread granularity. To trace the picture shown in Figure 2-8 with a resolution of 512×512 pixels, around 300,000 threads are created over the running time of about 1590 seconds, resulting in an average running time per thread of about 3 milliseconds ($\approx$ 100,000 Sparc cycles). Compared to this long thread run length, the average per-thread overhead is negligible.

As pointed out at the beginning of this section, the time required to trace an individual ray can vary significantly. To show this uneven work requirement, we calculated a work histogram for our example. The left part of Figure 2-8 shows the traced picture and the right part of the figure shows the work histogram. The work histogram shows how certain

	static load distribution			dynamic load distribution			
nodes	min time	max time	traced	time	max traced	min traced	impr.
1		1590 sec.	262144 rays	1590 sec.	262144 rays		
2	602 sec.	988 sec.	131072 rays	795 sec.	135312 rays	126832 rays	20 %
4	236 sec.	523 sec.	65536 rays	396 sec.	79073 rays	48867 rays	24 %
8	101 sec.	257 sec.	32768 rays	189 sec.	45175 rays	21464 rays	26 %
16	46 sec.	128 sec.	16384 rays	90 sec.	22616 rays	10711 rays	30 %
32	23 sec.	70 sec.	8192 rays	46 sec.	12543 rays	5859 rays	34 %
64	10 sec.	36 sec.	4096 rays	23 sec.	7210 rays	2595 rays	36 %

Table 2.1: This shows execution times for the static and PCM versions of the ray tracer. For the static version the rays are evenly distributed so we show the distribution of work, while for the PCM version the work is evenly distributed so we show the distribution of rays.

areas of the picture, such as the eye, contain much more work that other areas, such as the sky. To show how this uneven work distribution can affect execution time, we compared the speedup behavior of our multithreaded ray tracer with that of an implementation using a static load balancing scheme. This comparison is shown in the first four columns of Table 2.1. The static algorithm employs a simple work distribution that assigns exactly the same number of rays to each node. For the static case we listed two timings: the execution time of the fastest processor and the execution time of the slowest processor. We can see that even with just two nodes, the slower processor requires 64% more compute cycles than the faster processor. Even worse, this gap widens as we increase the number of processors. When run on 64 nodes, the slowest processor takes 3.6 times as long as the fastest processor.

The multithreaded ray tracer, on the other hand, distributes the points such that each processor performs almost the same amount of computation. Not only does the PCM code perform better for all machine configurations than the static solution, it even achieves perfect linear speedup. The improvement over the static program is shown in the last column of the table. The improvement is measured as 1-(PCM_time/static_time). On 64 processors the PCM program executes in only 64% of the time that the statically distributed program takes. We measured the range of the number of pixels traced per processor when run under PCM, and we have included this data in Table 2.1. These numbers reflect the effect of the dynamic load balancer. As expected, the difference between the maximum and minimum number of pixels traced per node was significant, and the percentage difference increases as we move to larger machine configurations.

2.4.2 Protein Folding

A second application that we implemented was protein folding. The reasons for choosing this application were similar to the reasons for choosing ray-tracing. The first is that the problems are large enough to warrant the use of parallelism. A common problem size takes over six hours when run sequentially, and we wanted to run a series of problems. The second reason is that initial attempts to parallelize the program did not make efficient use of the machine. These attempts statically broke the computation into subcomputations; but the subcomputations were too coarse, and their run times too variable, to keep all processors busy. An implementation using PCM avoids this problem.

The work on this problem was done in conjunction with Pande, Yu, Grosberg, and Tanaka of the Center for Material Sciences and Engineering at MIT. In their work[PYGT94] Pande, Yu, Grosberg, and Tanaka use the lattice model [SG90] to model protein folding. In this model a protein is described as a chain of monomers, and it is assumed that in a folded protein each monomer will sit on a point on a 3-dimensional lattice. Each possible folding of the polymer can then be described as some path along the set of lattice points. Figure 2-9 shows a polymer of length 26, with each shade representing a different type of monomer. The model assumes the polymer will take on the most compact possible paths, so it is only concerned with paths that completely fill some cube. In a folded polymer, a pair of monomers will exert some attractive or repulsive force on one another. This force depends on the types of the two monomers, and their distance. The energy of a folded polymer can be modeled as the sum of the forces between all pairs of monomers, or between all neighboring monomers. Of course, this energy value depends greatly on the way in which the polymer is folded. A typical computation consists of considering all possible foldings of a given polymer and computing a histogram of the energy values. We implemented this algorithm in PCM based on the problem description given to us by Pande.

For the rest of this section we will be concerned mainly with the implementation of this problem using PCM, focusing on the routine that enumerates all possible paths. More details on the algorithms used and the results obtained with this program are given in [PJGT94]. At its heart, this program is a search program that finds all possible unique paths that visit each node of the cube exactly once. This algorithm works by incrementally building up paths through the cube until complete paths are reached. The function

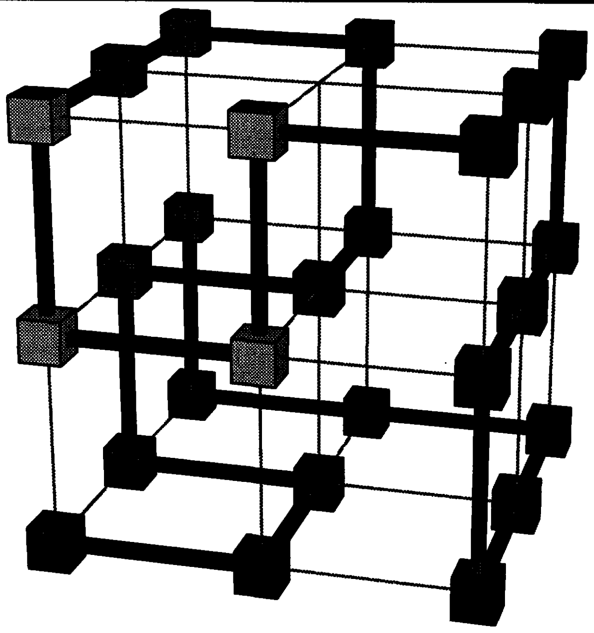

Figure 2-9: A Folded Polymer

Count_Entries performs the core of the search. An outline of the sequential code for this function is given in Figure 2-10.

The first argument to this function is a STATE structure. This structure defines the partial path that has been constructed so far. The contents of this structure depend on the particular calculation being performed, but it typically contains information describing which points are occupied, the type of monomer at each occupied point, and other data used to increase the efficiency of the search. The size of the STATE structure is typically on the order of 100 bytes. The second argument to the function is *point*, the lattice point to be added to the partial path. This function returns an integer, namely the number of paths found. The function first adds *point* to the partial path. If this completes the path, the function performs some calculation, typically updating a result histogram with the energy value of this new path, and then returns. Otherwise it calls itself recursively for each empty neighbor of *point*. At the end it sums up the number of complete paths found, and returns this total. We start the search by calling `Count_Entries` repeatedly on a set of starting paths. These starting paths are precomputed and are chosen to prevent the consideration of paths related by symmetry. Typically we run the program on several polymers at a time. Each time we find a complete path we calculate several energy values, one for each input polymer. This amortizes the time spent searching over several polymers. We improved

```
int Count_Entries(struct STATE *orig_st, int point) {
    struct STATE st_struct;                 /**local copy of state **/
    struct STATE *state = &st_struct;
    memcpy(state,orig_st,STATE_SIZE);
    add_point_to_path(point,state);         /** add point to path **/

    /** If we found a complete path update the result histogram **/
    /** and return 1 (the number of paths found.) **/
    if (complete_path(state)){
        update_result(state);
        return 1 ;
    }

    /** Otherwise call Count_Entries recursively on each neighbor **/
    sum = 0;
    for(i=0;i<num_neighbors;i++){
        next_neighbor=neighbor[i];
        if (not_occupied(next_neighbor,state)){
            sum += count_entries(state, next_neighbor);
        }
    }
    return sum;
}
```

Figure 2-10: Kernel of Sequential Protein Folding

Pande's original algorithm by adding checks which prevent the search from considering certain paths which cannot lead to a complete path. These changes, which are described in Appendix A, provided a performance improvement of 1 to 2 orders of magnitude on the problem sizes we have run.

The number of walks on a lattice increases exponentially with the size of the lattice, so significant speedups were needed in order to gain the necessary computational speed to calculate the number of walks on sublattices larger than $3 \times 3 \times 3$. An earlier attempt to parallelize this algorithm was made without using PCM. In this code each starting path was statically assigned to a node. Each node then executed the sequential code for its subset of the starting paths. The number of complete paths reachable from different starting paths can differ by many orders of magnitude. Therefore the work was not evenly divided between the nodes, and the speedups obtained by this program were disappointing.

```
thread Count_Entries(cont parent; int point, char st_vec[STATE_SIZE,) {
      struct STATE *state = (struct STATE *) st_vec;
      add_point_to_path(point,state);
      if (complete_path(state)){
            update_result_histogram(state);
            send_argument(parent, 1);
            return;
      }

      /** Determine number of neighbor nodes to be visited **/
      num_ntv = f(state,point);     /** num_ntv = num of neighbors to visit **/
                                    /** nbrs_to_visit[i] = 'i'th neighbor to visit **/

      /** Case 0: If no paths to search, then return 0 (no paths found) **/
      if (num_ntv==0) send_argument(parent, 0);

      /** Case 1: If exactly 1 neighbor to visit - only try that one  **/
      else if (num_ntv==1)
            tail_call(Count_Entries,parent,nbrs_to_visit[0],*state);
      else{
            /** General case - n neighbors to try [n>1] **/
            /** create a closure to sum results of all sub-computations **/
            /** post num_ntv-1 threads and perform a tail call for the last **/
            sum_closure = make_closure(sum,parent,num_ntv,_=[num_ntv]);
            for(i=0;i<(num_ntv-1);i++){
                  next_neighbor=nbrs_to_visit[i];
                  k1 = make_closure(Count_Entries,
                                    cont{sum_closure,sum:val[i]},
                                    next_neighbor, new_st=[STATE_SIZE]);
                  memcpy(new_st,st,STATE_SIZE);
                  post(k1);
                  }
            /**Perform a tail call for final neighbor **/
            new_parent = cont{sum_closure,sum:val[num_ntv-1]};
            tail_call(Count_Entries,new_parent,next_neighbor+1,*state);
      }
}
```

Figure 2-11: Kernel of Parallel Protein Folding

Protein Folding Parallelization

To get a more efficient parallelization, the computation needed to be broken into finer grains. PCM was ideal for this task. The procedure that makes use of the PCM primitives is the `Count_Entries` procedure. A skeleton of the code for this procedure is given in Figure 2-11. The major difference between the PCM code and the serial code is that rather than making recursive calls to `Count_Entries` the code instead creates and posts closure to execute the calls of `Count_Entries`. In addition, a `sum` closure is created which will receive the results of all the child threads and sum the results together. The unusual syntax in the call to make_closure (*i.e.,* " $\_ = [num\_ntv]$") signifies that a specified number of empty slots (here *num_ntv*) should be left in the closure. These slots will be filled in later with the results of the subcomputations. Most of the recursive calls are made by making and posting closures.

There are also other differences between the serial and parallel codes. Most of these differences were introduced for performance reasons, however, rather than correctness reasons. The first difference is that this version determines in advance the number of neighbors that will be visited. If there is just one neighbor that needs to be visited, then exactly one recursive call needs to be made. We do this by making use of a tail-call. In this instance the tail-call eliminates two overheads: first, the posting and scheduling of the closure, and second, the copying of the state argument into the new closure. Also, when more than one recursive call is needed, the final call makes use of a tail-call for the same reasons as given above.

When this code was first written the `tail_call` primitive did not exist. Originally we wrote the code where all the recursive calls to `Count_Entries` were implemented by creating and posting closures. Then we modified the code by hand to use the C `goto` statement to implement the final recursive call to `Count_Entries`. This improved the efficiency of the code by reducing the number of closures that had to be initialized, scheduled, and executed. Adding the `goto`'s by hand was fairly straightforward, but it made the code ugly and harder to read. In order to get these performance improvements without the user programming with `goto`'s directly, we introduced the `tail_call` primitive into the PCM language. Also notice that the state structure is passed around by treating it as an array. Passing structures in this way is somewhat cumbersome, so the language was later modified to allow structures to be passed to threads.

Number of processors	$3 \times 3 \times 3$ for (20 polymers)	$4 \times 3 \times 3$ (1 polymer)
serial	43.54 sec.	21334 sec.
1	46.14 sec.	22704 sec.
2	23.07 sec.	11302 sec.
4	11.55 sec.	5639 sec.
8	5.79 sec.	2818 sec.
16	3.00 sec.	1411 sec.
32	1.47 sec.	705 sec.
64	0.76 sec.	386 sec.
128		177 sec.

Table 2.2: This table gives execution times for the protein folding code

Protein Folding Results

Many variations of this program have been run on a range of problem and machine sizes. Results for two problem sizes are shown in Table 2.2. The second column shows the run times for runs on a $3 \times 3 \times 3$ cube which has 103,346 paths. This experiment was run with a typical input size of 20 polymers, which means that for each complete path found, energy calculations are performed for 20 polymers. The next column shows runs on a $4 \times 3 \times 3$ cube, which has over 84 million paths. For this run just one input polymer was specified. In each column the run time for the sequential code is given, followed by the run times for various parallel machine sizes.

The first observation is that the overhead added by the PCM model is fairly small. We measured the *efficiency* of the program by dividing the runtime of the serial program by the runtime of the parallel program when run on one processor. The efficiencies for both these runs were 94%. The overheads for this program, although still fairly small at only 6%, are larger than for the ray-tracer example because the length of the threads in the protein folding code are much shorter than in the ray-tracer.

The second observation is that the speedups for both these problems are quite good. We define *speedup* as the runtime of the parallel program on one processor divided by the speedup of the parallel program run on n processors. The smaller problem achieves speedups of 61 on 64 processors, while the larger problem achieves linear speedups up to 128 processors (the largest machine on which we ran the code).

With this program we were able to enumerate all of the 134,131,827,475 paths on a $4 \times 4 \times 3$ lattice. This computation was performed in several pieces on different partitions of various sizes, taking, in total, the equivalent of 128 hours on a 64-node CM-5.

2.5 Conclusions

The performance of any parallel program must scale over the performance of the best sequential program to be truly practical. Because of the high costs of dynamic scheduling and network communication in current message-passing architectures, this goal becomes a serious challenge when programming applications with unstructured parallelism.

As the outcome of experimenting with PCM, we identified two scheduling policies of general use which increase the efficiency of parallel applications run under a dynamic execution model. First, a work stealing scheduling policy enables almost-all-local computation, resulting in linear and near-linear speedups of the ray-tracing and protein-folding examples. Second, the tail-call mechanism gives the programmer the flexibility to glue short threads into longer ones. Tail-calls are especially important for very fine grained computation, such as the Fibonacci example. The parallel continuation-passing model presented in this chapter incorporates these two mechanisms.

The PCM model can either serve as a compilation target for a higher level language, or it can be used directly in conjunction with a sequential language, such as C. In the latter case it comes as a simple extension, providing the essential structures needed to synchronize computational threads and to optimize scheduling decisions. Although it could be argued that PCM is difficult to program because of its explicit continuation-passing style, we found it often the case that a program just has a small kernel that needs to be parallelized, leaving the rest of the program in its original sequential form.

Although PCM was successfully used for parallelizing several applications, we discovered that there were still many ways to improve this system. In particular, more work was needed on making the system easier to program. In addition to having to program in continuation-passing style, a programmer had to construct runtime system primitives, such as closures and continuations. These details should really have been hidden from the programmer. A second area that needed to be explored further was global data structures. PCM provides no support for global structures, yet many parallel applications need to make use of them.

Support for such structures was needed in order to increase the range of programs which could be written. Lastly, we needed to gain a better understanding of the work-stealing scheduler. Both of the examples presented in this chapter achieved nearly linear speedups. We wanted to know if all similar applications would also achieve these results, or if we just got lucky. Clearly not all programs could be executed with linear speedup. We also wanted to understand what properties a program must have in order to achieve linear speedups. All of these areas are addressed in future chapters.

Chapter 3

Cilk1: A Provably Good Runtime System

In the previous chapter we described the PCM system for multithreaded programming. With the PCM system a programmer writes his multithreaded program in a continuation-passing style by defining a group of threads and specifying how they communicate. The system then takes care of all the details of executing the program on the underlying parallel hardware. The system encapsulates the user's threads into closures so that they can be freely migrated between nodes. A work stealing scheduler is used to schedule the execution of the threads and to balance the work load amongst the processors. For the applications we implemented this system executes our code with little overhead and achieves good speedups.

Although the scheduler in PCM seems to perform well in practice, as with other runtime systems [ABLL91, CRRH93, CGH94, CAL+89, CD88, CSS+91, FLA94, Hal85, HWW93, JP92, Kal90, KC93, KHM89, Nik93, Nik94, RSL93, TBK93, VR88], the PCM system does not provide users with any guarantees of application performance. When a user writes a program, there is no way for him to know for sure what the performance of the code will be. If his code performs poorly, the user has no way of knowing why it performed that way, or even if the poor performance is due to the program itself or due to the runtime system.

To address this problem we incorporated a provably good scheduler into the PCM system and renamed the system Cilk-1. We also added additional structure to the language, and

The work described in this chapter was previously reported on in a paper [BJK+95] by Robert Blumofe, Bradley Kuszmaul, Charles Leiserson, Keith Randall, Yuli Zhou, and myself.

cleaned up the language a bit. With these changes Cilk-1's work-stealing scheduler achieves space, time, and communication bounds all within a constant factor of optimal. Moreover, the system gives the user an algorithmic model of application performance based on the measures of "work" and "critical path." This chapter describes the Cilk-1 system and demonstrates the efficiency of the Cilk-1 scheduler both empirically and analytically.

This chapter represents joint work by several people. The system described in this chapter was designed and implemented by the Cilk-1 team which was led by Prof. Charles Leiserson, and consisted of Robert Blumofe, Bradley Kuszmaul, Keith Randall, Yuli Zhou, and myself. Much of the theoretical work reported in Section 3.6 is based on work by Charles Leiserson and Robert Blumofe.

3.1 Cilk-1 Overview

A Cilk multithreaded computation can be viewed as a directed acyclic graph (dag) that unfolds dynamically, as is shown schematically in Figure 3-1. Unlike PCM, in which there were no constraints on the dag, Cilk views the dag as having some structure. A Cilk program consists of a collection of Cilk *procedures*, each of which is broken into a sequence of *threads*, which form the vertices of the dag. Each thread is a *nonblocking* C function, which means that once it has been invoked it can run to completion without waiting or suspending. As one of the threads from a Cilk procedure runs, it can *spawn* a child thread which begins a new child procedure. In the figure, downward edges connect threads and their procedures with the children they have spawned. A spawn is like a subroutine call, except that the calling thread may execute concurrently with its child, possibly spawning additional children. Since threads cannot block in the Cilk model, a thread cannot spawn children and then wait for values to be returned. Rather, the thread must additionally spawn a *successor* thread to receive the children's return values when they are produced. A thread and its successors are considered to be parts of the same Cilk procedure. In the figure, sequences of successor threads that form Cilk procedures are connected by horizontal edges. Return values, and other values sent from one thread to another, induce *data dependencies* among the threads, where a thread receiving a value cannot begin until another thread sends the value. Data dependencies are shown as upward, curved edges in the figure. Thus, a Cilk computation unfolds as a *spawn tree* composed of procedures and the spawn edges that connect them to

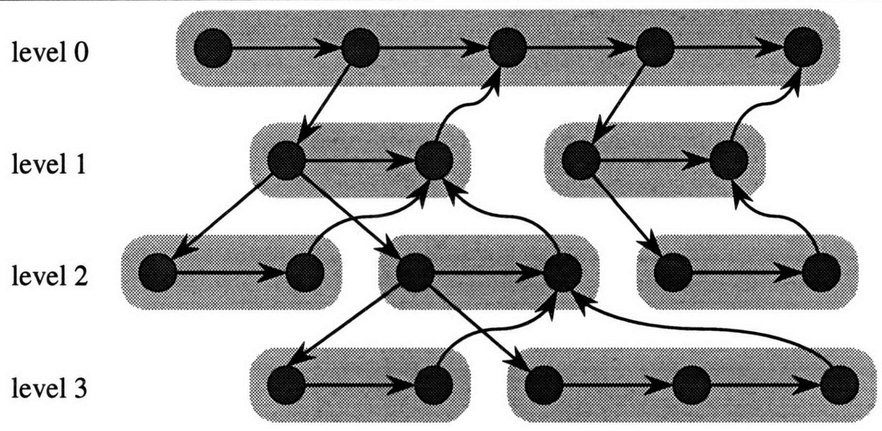

Figure 3-1: The Cilk model of multithreaded computation. Threads are shown as circles, which are grouped into procedures. Each downward edge corresponds to a spawn of a child, each horizontal edge corresponds to a spawn of a successor, and each upward, curved edge corresponds to a data dependency. The numbers in the figure indicate the levels of procedures in the spawn tree.

their children, but the execution is constrained to follow the precedence relation determined by the dag of threads.

The execution time of any Cilk-1 program on a parallel computer with P processors is constrained by two parameters of the computation: the *work* and the *critical path*. The work, denoted T_1, is the time used by a one-processor execution of the program, which corresponds to the sum of the execution times of all the threads. The critical path length, denoted T_∞, is the total amount of time required by an infinite-processor execution, which corresponds to the largest sum of thread execution times along any path. With P processors, the execution time cannot be less than T_1/P or less than T_∞. The Cilk-1 scheduler uses "work stealing" [BL94, BS81, FMM94, FM87, FLA94, Hal85, KZ93, KHM89, Kus94, Nik94, VR88] to achieve execution time very near to the sum of these two measures. Off-line techniques for computing such efficient schedules have been known for a long time [Bre74, Gra66, Gra69], but this efficiency has been difficult to achieve on-line in a distributed environment while simultaneously using small amounts of space and communication.

In this chapter we demonstrate the efficiency of the Cilk-1 scheduler both empirically and analytically. Empirically, we have been able to document that Cilk-1 works well for dynamic, asynchronous, tree-like, MIMD-style computations. To date, the applications we have programmed include protein folding, graphic rendering, backtrack search, and the ⋆Socrates chess program, which won second prize in the 1995 World Computer Chess

Championship. Many of these applications pose problems for more traditional parallel environments, such as message passing [Sun90] and data parallel [Ble92, HS86], because of the unpredictability of the dynamic workloads on processors. Analytically, we prove that for "fully strict" (well-structured) programs, Cilk-1's work-stealing scheduler achieves execution space, time, and communication bounds all within a constant factor of optimal.

The Cilk-1 language is an extension to C that provides an abstraction of threads in explicit continuation-passing style. A Cilk-1 program is preprocessed to C and then linked with a runtime library to run on the Connection Machine CM-5 MPP, the Intel Paragon MPP, or the Silicon Graphics Power Challenge SMP. In addition, Blumofe has designed a fault tolerant version of Cilk, called Cilk-NOW[Blu95, BP94], which runs on a network of workstations. In this chapter, we focus on the Connection Machine CM-5 implementation of Cilk-1. The Cilk-1 scheduler on the CM-5 is written in about 40 pages of C, and it performs communication among processors using the Strata [BB94] active-message library.

The remainder of this chapter is organized as follows. Section 3.2 describes Cilk-1's runtime data structures and the C language extensions that are used for programming. Section 3.3 describes the work-stealing scheduler. Section 3.4 documents the performance of several Cilk-1 applications. Section 3.5 shows how the work and critical path of a Cilk-1 computation can be used to model performance. Section 3.6 shows analytically that the scheduler works well. Finally, Section 3.7 offers some concluding remarks and describes our plans for the future.

3.2 Cilk Programming Environment and Implementation

In this section we describe the C-language extensions that we have developed to ease the task of coding Cilk-1 programs. We also explain the basic runtime data structures that Cilk-1 uses. The Cilk-1 language extensions are basically a cleaned-up version of the extensions in PCM. They hide more of the implementation details than the original PCM did, and they also allow the programmer to place more structure on the dag so that the threads of the computation can be treated as a being grouped into procedures, as was shown in Figure 3-1.

In the Cilk-1 language, a thread T is defined in a manner similar to a C function definition:

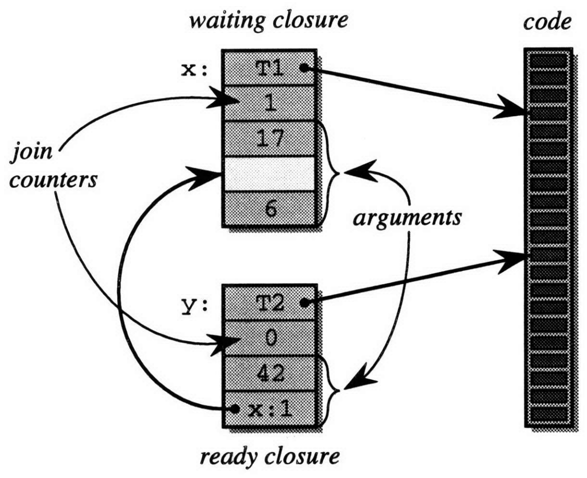

Figure 3-2: The closure data structure.

```
thread T (arg-decls ...) { stmts ...}
```

The Cilk-1 preprocessor translates T into a C function of one argument and void return type. The one argument is a pointer to a *closure* data structure, illustrated in Figure 3-2, which holds the arguments for T. A closure includes a pointer to the C function for T, a slot for each of the specified arguments, and a *join counter* indicating the number of missing arguments that need to be supplied before T is ready to run. A closure is *ready* if it has obtained all of its arguments, and it is *waiting* if some arguments are missing. To run a ready closure, the Cilk-1 scheduler invokes the thread as a procedure using the closure itself as its sole argument. Within the code for the thread, the arguments are copied out of the closure data structure into local variables. The closure is allocated from a simple runtime heap when it is created, and it is returned to the heap when the thread terminates.

The Cilk-1 language supports a data type called a *continuation*, which is specified by the type modifier keyword `cont`. A continuation is essentially a global reference to an empty argument slot of a closure, implemented as a compound data structure containing a pointer to a closure and an offset that designates one of the closure's argument slots. Continuations can be created and passed among threads, which enables threads to communicate and synchronize with each other. Continuations are typed with the C data type of the slot in the closure.

At runtime, a thread can spawn a child thread by creating a closure for the child.

Spawning is specified in the Cilk-1 language as follows:

```
spawn T (args ...)
```

This statement creates a child closure, fills in all available arguments, and initializes the join counter to the number of missing arguments. Available arguments are specified as in C. To specify a missing argument, the user specifies a continuation variable (of type `cont`) preceded by a question mark. For example, if the second argument of a spawned thread is `?k`, then Cilk-1 sets the variable `k` to a continuation that refers to the second argument slot of the created closure. If the closure is ready, that is, it has no missing arguments, then **spawn** causes the closure to be immediately posted to the scheduler for execution. In typical applications, child closures are spawned with no missing arguments.

To create a successor thread, a thread executes the following statement:

```
spawn_next T (args ...)
```

This statement is semantically identical to **spawn**, but it informs the scheduler that the new closure should be treated as a successor, as opposed to a child. Successor closures are usually created with some missing arguments, which are filled in by values produced by the children.

A Cilk-1 procedure does not ever return values in the normal way to a parent procedure. Instead, the programmer must code the parent procedure as two threads. The first thread spawns the child procedure, passing it a continuation pointing to the successor thread's closure. The child sends its "return" value explicitly as an argument to the waiting successor. This strategy of communicating between threads is called *explicit continuation passing.* Cilk-1 provides primitives of the following form to send values from one closure to another:

```
send_argument (k, value)
```

This statement sends the value *value* to the argument slot of a waiting closure specified by the continuation *k*. The types of the continuation and the value must be compatible. The join counter of the waiting closure is decremented, and if it becomes zero, then the closure is ready and is posted to the scheduler.

Figure 3-3 shows the familiar recursive Fibonacci procedure written in Cilk-1. It consists of two threads, `fib` and its successor **sum**. Reflecting the explicit continuation-passing style

```
thread fib (cont int k, int n)
{  if (n<2)
      send_argument (k, n)
   else
   {  cont int x, y;
      spawn_next sum (k, ?x, ?y);
      spawn fib (x, n-1);
      spawn fib (y, n-2);
   }
}

thread sum (cont int k, int x, int y)
{  send_argument (k, x+y);
}
```

Figure 3-3: A Cilk-1 procedure, consisting of two threads, to compute the nth Fibonacci number.

that Cilk-1 supports, the first argument to each thread is the continuation specifying where the "return" value should be placed.

When the `fib` function is invoked, it first checks to see if the boundary case has been reached, in which case it uses `send_argument` to "return" the value of `n` to the slot specified by continuation `k`. Otherwise, it spawns the successor thread `sum`, as well as two children to compute the two subcases. Each of these two children is given a continuation specifying to which argument in the `sum` thread it should send its result. The `sum` thread simply adds the two arguments when they arrive and sends this result to the slot designated by continuation `k`.

This code is similar to the PCM code for Fibonacci. The main difference is that the Cilk-1 version is written at a slightly higher level since the Cilk-1 system has abstracted away all the details about how threads are implemented. In the PCM version the user had to deal directly with the closures that the runtime system uses to represent threads.

Although writing in explicit continuation-passing style is somewhat onerous for the programmer, the decision to break procedures into separate nonblocking threads simplifies the Cilk-1 runtime system. Each Cilk-1 thread leaves the C runtime stack empty when it completes. Thus, Cilk-1 can run on top of a vanilla C runtime system. A common alternative [Hal85, KC93, MKH91, Nik94] is to support a programming style in which a thread suspends whenever it discovers that required values have not yet been computed, resuming when the values become available. When a thread suspends, however, it may leave temporary values on the runtime stack which must be saved, or each thread must have its own runtime stack. Consequently, this alternative strategy requires changes to the runtime system that depend on the C calling stack layout and register usage conventions. Another advantage of Cilk-1's strategy is that it allows multiple children to be spawned from a single nonblocking thread, which saves on context switching. In Cilk-1, r children can be spawned and executed with only $r + 1$ context switches, whereas the alternative of suspending whenever a thread is spawned causes $2r$ context switches. Since our primary interest is in understanding how to build efficient multithreaded runtime systems, but without redesigning the basic C runtime system, we chose the alternative of burdening the programmer with a requirement which is perhaps less elegant linguistically, but which yields a simple and portable runtime implementation.

Cilk-1 supports a variety of features that give the programmer greater control over runtime performance. For example, when the last action of a thread is to spawn a ready thread, the programmer can use the keyword `call` instead of **spawn**. Using `call` produces a "tail call" which runs the new thread immediately without invoking the scheduler. Cilk-1 also allows arrays and subarrays to be passed (by value) as arguments to closures. Other features include various abilities to override the scheduler's decisions, including on which processor a thread should be placed and how to pack and unpack data when a closure is migrated from one processor to another.

Cilk-1 can also automatically compute the critical path length and total work of a computation. As we will see later, these values are useful to a programmer trying to understand the performance of his program. The computation of the critical path is done by a system of time-stamping, as shown in Figure 3-4.

3.3 Cilk's Work-Stealing Scheduler

Cilk's scheduler uses the technique of work-stealing [BL94, BS81, FMM94, FM87, FLA94, Hal85, KZ93, KHM89, Kus94, Nik94, VR88] in which a processor (the thief) who runs out of work selects another processor (the victim) from whom to steal work, and then steals the shallowest ready thread in the victim's spawn tree. Cilk's strategy for selecting the victim processor is to have the thief choose the victim at random [BL94, KZ93, RSAU91].

At runtime, each processor maintains a local *ready queue* to hold ready closures. Each

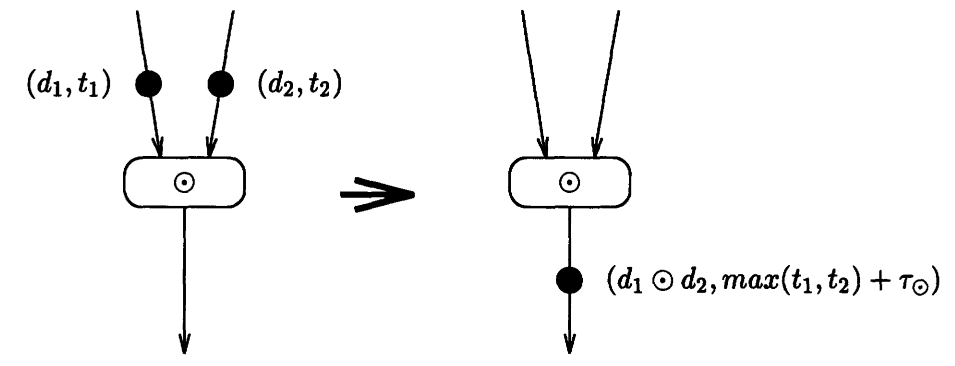

Figure 3-4: The time at which an instruction in a dataflow graph is executed in a perfect infinite-processor schedule can be computed by time-stamping the tokens. In addition to the normal data-value of a token (d_1, d_2, and $d_1 \odot d_2$ respectively in the figure), the token includes a time-stamp (t_1, t_2, and $\max(t_1, t_2) + \tau_\odot$ respectively.) The time-stamp on the outgoing token is computed as a function of the time-stamps of the incoming tokens and the time to execute the instruction.

closure has an associated *level*, which corresponds to the number of **spawn**'s (but not **spawn_next**'s) on the path from the root of the spawn tree. The ready queue is an array in which the Lth element contains a linked list of all ready closures having level L.

Cilk begins executing the user program by initializing all ready queues to be empty, placing the root thread into the level-0 list of Processor 0's queue, and then starting a scheduling loop on each processor. Within a scheduling loop, a processor first checks to see whether its ready queue is empty. If it is, the processor commences "work stealing," which will be described shortly. Otherwise, the processor performs the following steps:

1. Remove the thread at the head of the list of the deepest nonempty level in the ready queue.
2. Extract the thread from the closure, and invoke it.

As a thread executes, it may spawn or send arguments to other threads. When the thread terminates, control returns to the scheduling loop.

When a thread at level L spawns a child thread T, the scheduler executes the following operations:

1. Allocate and initialize a closure for T.
2. Copy the available arguments into the closure, initialize any continuations to point to missing arguments, and initialize the join counter to the number of missing arguments.

3. Label the closure with level $L + 1$.
4. If there are no missing arguments, post the closure to the ready queue by inserting it at the head of the level-$(L+1)$ list.

Execution of **spawn_next** is similar, except that the closure is labeled with level L and, if it is ready, posted to the level-L list.

A processor that executes **send_argument**(k, *value*) performs the following steps:

1. Find the closure and argument slot referenced by the continuation k.
2. Place *value* in the argument slot, and decrement the join counter of the closure.
3. If the join counter goes to zero, post the closure to the ready queue at the appropriate level.

When the continuation k refers to a closure on a remote processor, network communication ensues. The processor that initiated the **send_argument** function sends a message to the remote processor to perform the operations. The only subtlety occurs in step 3. If the closure must be posted, it is posted to the ready queue of the initiating processor, rather than to that of the remote processor. This policy is necessary for the scheduler to be provably good; so migrating a closure for this reason is called a *provably good steal.* As a practical matter, we have also had success with posting the closure to the remote processor's queue, which can sometimes save a few percent in overhead.

If the scheduler attempts to remove a thread from an empty ready queue, the processor becomes a thief and commences *work stealing* as follows:

1. Select a victim processor uniformly at random.
2. If the victim's ready queue is empty, go to step 1.
3. If the victim's ready queue is nonempty, extract a thread from the head of the list in the shallowest nonempty level of the ready queue, and invoke it.

Work stealing is implemented with a simple request-reply communication protocol between the thief and victim.

Why steal work from the shallowest level of the ready queue? The reason is two-fold. First, we would like to steal large amounts of work, and shallow closures are likely to execute for longer than deep ones. Stealing large amounts of work tends to lower the communication cost of the program, because fewer steals are necessary. Second, the closures at the shallowest level of the ready queue are also the ones that are shallowest in the dag, a key fact used in Section 3.6. Consequently, if processors are idle, the work they steal tends to make progress along the critical path.

3.4 Performance of Cilk-1 Applications

This section presents several applications that we have used to benchmark the Cilk-1 scheduler. We also present empirical evidence from experiments run on a CM-5 to document the efficiency of our work-stealing scheduler. The CM-5 is a massively parallel computer based on 32MHz SPARC processors with a fat-tree interconnection network [LAD+92].

The applications are described below:

- **fib(n)** is the same as was presented in Section 3.2, except that the second recursive spawn is replaced by a "tail call" that avoids the scheduler. This program is a good measure of Cilk-1 overhead, because the thread length is so small.
- **queens(N)** is a backtrack search program that solves the problem of placing N queens on a $N \times N$ chessboard so that no two queens attack each other. The Cilk-1 program is based on serial code by R. Sargent of the MIT Media Laboratory. Thread length was enhanced by serializing the bottom 7 levels of the search tree.
- **pfold(x,y,z)** is a protein-folding program [PJGT94] written in conjunction with V. Pande of MIT's Center for Material Sciences and Engineering. This program was described in more detail in Section 2.4. This program finds hamiltonian paths in a three-dimensional grid of size $x \times y \times z$. It was the first program to enumerate all hamiltonian paths in a $3 \times 4 \times 4$ grid. For this benchmark we timed the enumeration of all paths starting with a certain sequence.
- **ray(x,y)** is a parallel program for graphics rendering based on the serial POV-Ray program, which uses a ray-tracing algorithm. This program was described in Section 2.4. The core of POV-Ray is a simple doubly nested loop that iterates over each pixel in a two-dimensional image of size (x, y). For **ray** we converted the nested loops into a 4-ary divide-and-conquer control structure using spawns.[1] Our measurements do not include the approximately 2.4 seconds of startup time required to read and

[1]Initially, the serial POV-Ray program was about 5 percent slower than the Cilk-1 version running on one processor. The reason was that the divide-and-conquer decomposition performed by the Cilk-1 code provides better locality than the doubly nested loop of the serial code. Modifying the serial code to imitate the Cilk-1 decomposition improved its performance. Timings for the improved version are given in the table.

	fib (33)	queens (15)	pfold (3,3,4)	ray (500,500)	knary (10,5,2)	knary (10,4,1)	⋆Socrates (10) **32 proc.**	⋆Socrates (10) **256 proc.**
(application parameters)								
T_{serial}	8.487	252.1	615.15	729.2	288.6	40.993	1665	1665
T_1	73.16	254.6	647.8	732.5	314.6	45.43	3644	7023
T_{serial}/T_1	0.116	0.9902	0.9496	0.9955	0.9174	0.9023	0.4569	0.2371
T_∞	0.000326	0.0345	0.04354	0.0415	4.458	0.255	3.134	3.24
T_1/T_∞	224417	7380	14879	17650	70.56	178.2	1163	2168
threads	17,108,660	210,740	9,515,098	424,475	5,859,374	873,812	26,151,774	51,685,823
thread length	4.276μs	1208μs	68.08μs	1726μs	53.69μs	51.99μs	139.3μs	135.9μs
(32-processor experiments)								
T_P	2.298	8.012	20.26	21.68	15.13	1.633	126.1	-
$T_1/P + T_\infty$	2.287	7.991	20.29	22.93	14.28	1.675	117.0	-
T_1/T_P	31.84	31.78	31.97	33.79	20.78	27.81	28.90	-
$T_1/(P \cdot T_P)$	0.9951	0.9930	0.9992	1.0558	0.6495	0.8692	0.9030	-
space/proc.	70	95	47	39	41	42	386	-
requests/proc.	185.8	48.0	88.6	218.1	92639	3127	23484	-
steals/proc.	56.63	18.47	26.06	79.25	18031	1034	2395	-
(256-processor experiments)								
T_P	0.2892	1.045	2.590	2.765	8.590	0.4636	-	34.32
$T_1/P + T_\infty$	0.2861	1.029	2.574	2.903	5.687	0.4325	-	30.67
T_1/T_P	253.0	243.7	250.1	265.0	36.62	98.00	-	204.6
$T_1/(P \cdot T_P)$	0.9882	0.9519	0.9771	1.035	0.1431	0.3828	-	0.7993
space/proc.	66	76	47	32	48	40	-	405
requests/proc.	73.66	80.40	97.79	82.75	151803	7527	-	30646
steals/proc.	24.10	21.20	23.05	18.34	6378	550	-	1540

Table 3.1: Performance of Cilk on various applications. All times are in seconds, except where noted.

process the scene description file.

- `knary(n,k,r)` is a synthetic benchmark whose parameters can be set to produce a variety of values for work and critical path. It generates a tree of branching factor k and depth n in which the first r children at every level are executed serially and the remainder are executed in parallel. At each node of the tree, the program runs an empty "for" loop for 400 iterations.

- ⋆Socrates is a parallel chess program that uses the Jamboree search algorithm [JK94, Kus94] to parallelize a minmax tree search. We give performance numbers for the search of a position to depth 10. The work of the algorithm varies with the number of processors, because it does speculative work that may be aborted during runtime. For this reason we give complete data sets for the two machine configurations. This application is described in more detail in Chapter 4.

Table 3.1 shows typical performance measures for these Cilk-1 applications. Each column presents data from a single run of a benchmark application. We adopt the following notations, which are used in the table. For each application, we have an efficient serial

C implementation, compiled using `gcc -O2`, whose measured runtime is denoted T_{serial}. The work T_1 is the measured execution time for the Cilk-1 program running on a single node of the CM-5. The critical path length T_∞ of the Cilk-1 computation is measured by timestamping each thread and does not include scheduling or communication costs. The measured P-processor execution time of the Cilk-1 program running on the CM-5 is given by T_P, which includes all scheduling and communication costs. The row labeled "threads" indicates the number of threads executed, and "thread length" is the average thread length (work divided by the number of threads).

Certain derived parameters are also displayed in the table. The ratio T_{serial}/T_1 is the *efficiency* of the Cilk-1 program relative to the C program. The ratio T_1/T_∞ is the *average parallelism*. The value $T_1/P + T_\infty$ is a simple model of the runtime, which will be discussed in the next section. The *speedup* is T_1/T_P, and the *parallel efficiency* is $T_1/(P \cdot T_P)$. The row labeled "space/proc." indicates the maximum number of closures allocated at any time on any processor. The row labeled "requests/proc." indicates the average number of steal requests made by a processor during the execution, and "steals/proc." gives the average number of closures actually stolen.

The data listed for $\star$Socrates differs slightly from the data listed for the rest of the programs. Since $\star$Socrates performs speculative computations, the amount of work performed by this program on a given input will vary as the machine size changes. For this reason the data for $\star$Socrates is listed in two columns; one column gives the data for a 32 processor run, the other for a 256 processor run. Since the work varies with the machine size, for T_1 instead of giving the execution time on one processor, we give the total work performed when run on the appropriate machine size. This approximates what T_1 would be if the program on one processor executed the same threads that the n processor version did.

The data in Table 3.1 shows two important relationships: one between efficiency and thread length, and another between speedup and average parallelism.

Considering the relationship between efficiency T_{serial}/T_1 and thread length, we see that for programs with moderately long threads, the Cilk-1 scheduler induces very little overhead. The `queens`, `pfold`, `ray`, and `knary` programs have threads with average length greater than 50 microseconds and have efficiency greater than 90 percent. On the other hand, the `fib` program has low efficiency, because the threads are so short: `fib` does almost nothing besides `spawn` and `send_argument`.

Despite it's long threads, the $\star$Socrates program shows low efficiency, because its parallel Jamboree search algorithm [Kus94] is based on speculatively searching subtrees that are not searched by a serial algorithm. Consequently, as we increase the number of processors, the program executes more threads and, hence, does more work. For example, the 256-processor execution did 7023 seconds of work whereas the 32-processor execution did only 3644 seconds of work. Both of these executions did considerably more work than the serial program's 1665 seconds of work. Thus, although we observe low efficiency, it is due to the parallel algorithm and not to Cilk-1 overhead.

Looking at the speedup T_1/T_P measured on 32 and 256 processors, we see that when the average parallelism T_1/T_∞ is large compared with the number P of processors, Cilk-1 programs achieve nearly perfect linear speedup, but when the average parallelism is small, the speedup is much less. The `fib`, `queens`, `pfold`, and `ray` programs, for example, have in excess of 7000-fold parallelism and achieve more than 99 percent of perfect linear speedup on 32 processors and more than 95 percent of perfect linear speedup on 256 processors.[2] The $\star$Socrates program exhibits somewhat less parallelism and also somewhat less speedup. On 32 processors the $\star$Socrates program has 1163-fold parallelism, yielding 90 percent of perfect linear speedup, while on 256 processors it has 2168-fold parallelism yielding 80 percent of perfect linear speedup. With even less parallelism, as exhibited in the **knary** benchmarks, less speedup is obtained. For example, the **knary(10,5,2)** benchmark exhibits only 70-fold parallelism, and it realizes barely more than 20-fold speedup on 32 processors (less than 65 percent of perfect linear speedup). With 178-fold parallelism, **knary(10,4,1)** achieves 27-fold speedup on 32 processors (87 percent of perfect linear speedup), but only 98-fold speedup on 256 processors (38 percent of perfect linear speedup).

Although these speedup measures reflect the Cilk-1 scheduler's ability to exploit parallelism, to obtain application speedup, we must factor in the efficiency of the Cilk-1 program compared with the serial C program. Specifically, the application speedup T_{serial}/T_P is the product of efficiency T_{serial}/T_1 and speedup T_1/T_P. For example, applications such as `fib` and $\star$Socrates with low efficiency generate correspondingly low application speedup. The $\star$Socrates program, with efficiency 0.2371 and speedup 204.6 on 256 processors, exhibits application speedup of $0.2371 \cdot 204.6 = 48.51$. For the purpose of performance prediction,

[2]In fact, the `ray` program achieves superlinear speedup even when comparing to the efficient serial implementation. We suspect that cache effects cause this phenomenon.

we prefer to decouple the efficiency of the application from the efficiency of the scheduler. We should point out that for this test we chose a chess position and searched it to a depth that could be run in a reasonable about of time on a serial machine. Under tournament time controls we would do a deeper search which would would increase the available parallelism, and thereby improve parallel performance.

Looking more carefully at the cost of a **spawn** in Cilk-1, we find that it takes a fixed overhead of about 50 cycles to allocate and initialize a closure, plus about 8 cycles for each word argument. In comparison, a C function call on a CM-5 processor takes 2 cycles of fixed overhead (assuming no register window overflow) plus 1 cycle for each word argument (assuming all arguments are transferred in registers). Thus, a **spawn** in Cilk-1 is roughly an order of magnitude more expensive than a C function call. This Cilk-1 overhead is quite apparent in the `fib` program, which does almost nothing besides **spawn** and **send_argument**. Based on `fib`'s measured efficiency of 0.116, we can conclude that the aggregate average cost of a **spawn/send_argument** in Cilk-1 is between 8 and 9 times the cost of a function call/return in C.

Efficient execution of programs with short threads requires a low-overhead spawn operation. As can be observed from Table 3.1, the vast majority of threads execute on the same processor on which they are spawned. For example, the `fib` program executed over 17 million threads but migrated only 6170 (24.10 per processor) when run with 256 processors. Taking advantage of this property, other researchers [KC93, MKH91] have developed techniques for implementing spawns such that when the child thread executes on the same processor as its parent, the cost of the spawn operation is roughly equal the cost of a C function call. We hope to incorporate such techniques into future implementations of Cilk.

Finally, we make two observations about the space and communication measures in Table 3.1.

Looking at the "space/proc." rows, we observe that the space per processor is generally quite small and does not grow with the number of processors. For example, $\star$Socrates on 32 processors executes over 26 million threads, yet no processor ever has more than 386 allocated closures. On 256 processors, the number of executed threads nearly doubles to over 51 million, but the space per processors barely changes. In Section 3.6 we show formally that for Cilk-1 programs, the space per processor does not grow as we add processors.

Looking at the "requests/proc." and "steals/proc." rows in Table 3.1, we observe that the amount of communication grows with the critical path but does not grow with the work. For example, `fib`, `queens`, `pfold`, and `ray` all have critical paths under a tenth of a second long and perform fewer than 220 requests and 80 steals per processor, whereas `knary(10,5,2)` and $\star$Socrates have critical paths more than 3 seconds long and perform more than 20,000 requests and 1500 steals per processor. The table does not show any clear correlation between work and either requests or steals. For example, `ray` does more than twice as much work as `knary(10,5,2)`, yet it performs two orders of magnitude fewer requests. In Section 3.6, we show that for "fully strict" Cilk-1 programs, the communication per processor grows linearly with the critical path length and does not grow as function of the work.

3.5 Modeling Performance

In this section, we further document the effectiveness of the Cilk-1 scheduler by showing empirically that it schedules applications in a near-optimal fashion. Specifically, we use the `knary` synthetic benchmark to show that the runtime of an application on P processors can be accurately modeled as $T_P \approx T_1/P + c_\infty T_\infty$, where $c_\infty \approx 1.5$. This result shows that we obtain nearly perfect linear speedup when the critical path is short compared with the average amount of work per processor. We also show that a model of this kind is accurate even for $\star$Socrates, which is our most complex application programmed to date and which does not obey all the assumptions assumed by the theoretical analyses in Section 3.6.

A good scheduler should run an application with T_1 work in T_1/P time on P processors. Such *perfect linear speedup* cannot be obtained whenever $T_\infty > T_1/P$, since we always have $T_P \geq T_\infty$, or more generally, $T_P \geq \max\{T_1/P, T_\infty\}$. The critical path T_∞ is the stronger lower bound on T_P whenever P exceeds the average parallelism T_1/T_∞, and T_1/P is the stronger bound otherwise. A good scheduler should meet each of these bounds as closely as possible.

In order to investigate how well the Cilk-1 scheduler meets these two lower bounds, we used our `knary` benchmark (described in Section 3.4), which can exhibit a range of values for work and critical path.

Figure 3-5 shows the outcome of many experiments of running `knary` with various values for k, n, r, and P. The figure plots the speedup T_1/T_P for each run against the machine size

P for that run. In order to compare the outcomes for runs with different parameters, we have normalized the data by dividing the plotted values by the average parallelism T_1/T_∞. Thus, the horizontal position of each datum is $P/(T_1/T_\infty)$, and the vertical position of each datum is $(T_1/T_P)/(T_1/T_\infty) = T_\infty/T_P$. Consequently, on the horizontal axis, the normalized machine-size is 1.0 when the average available parallelism is equal to the machine size. On the vertical axis, the normalized speedup is 1.0 when the runtime equals the critical path, and it is 0.1 when the runtime is 10 times the critical path. We can draw the two lower bounds on time as upper bounds on speedup. The horizontal line at 1.0 is the upper bound on speedup obtained from the critical path, and the 45-degree line is the upper bound on speedup obtained from the work per processor. As can be seen from the figure, on the **knary** runs for which the average parallelism exceeds the number of processors (normalized machine size < 1), the Cilk-1 scheduler obtains nearly perfect linear speedup. In the region where the number of processors is large compared to the average parallelism (normalized machine size > 1), the data is more scattered, but the speedup is always within a factor of 4 of the critical-path upper bound.

The theoretical results from Section 3.6 show that the expected running time of an application on P processors is $T_P = O(T_1/P + T_\infty)$. Thus, it makes sense to try to fit the data to a curve of the form $T_P = c_1(T_1/P) + c_\infty(T_\infty)$. A least-squares fit to the data to minimize the relative error yields $c_1 = 0.9543 \pm 0.1775$ and $c_\infty = 1.54 \pm 0.3888$ with 95 percent confidence. The R^2 correlation coefficient of the fit is 0.989101, and the mean relative error is 13.07 percent. The curve fit is shown in Figure 3-5, which also plots the simpler curves $T_P = T_1/P + T_\infty$ and $T_P = T_1/P + 2 \cdot T_\infty$ for comparison. As can be seen from the figure, little is lost in the linear speedup range of the curve by assuming that $c_1 = 1$. Indeed, a fit to $T_P = T_1/P + c_\infty(T_\infty)$ yields $c_\infty = 1.509 \pm 0.3727$ with $R^2 = 0.983592$ and a mean relative error of 4.04 percent, which is in some ways better than the fit that includes a c_1 term. (The R^2 measure is a little worse, but the mean relative error is much better.)

It makes sense that the data points become more scattered when P is close to or exceeds the average parallelism. In this range, the amount of time spent in work stealing becomes a significant fraction of the overall execution time. The real measure of the quality of a scheduler is how much larger T_1/T_∞ must be than P before T_P shows substantial influence from the critical path. One can see from Figure 3-5 that if the average parallelism exceeds P by a factor of 10, the critical path has almost no impact on the running time.

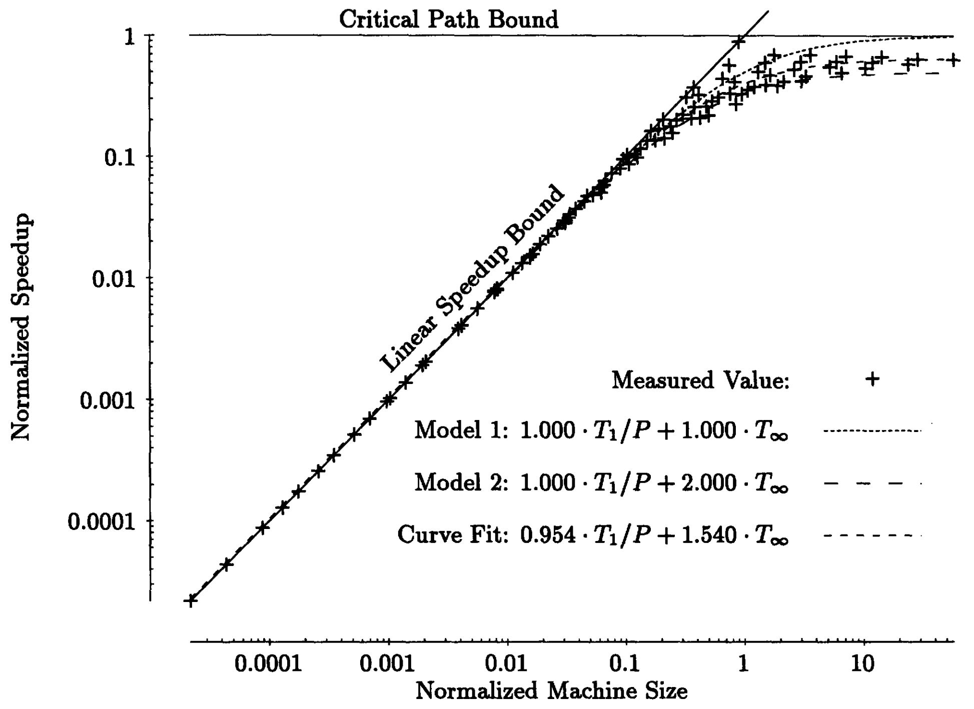

Figure 3-5: Normalized speedups for the **knary** synthetic benchmark using from 1 to 256 processors. The horizontal axis is P and the vertical axis is the speedup T_1/T_P, but each data point has been normalized by dividing the these parameters by T_1/T_∞.

To confirm our simple model of the Cilk-1 scheduler's performance on a real application, we ran ⋆Socrates on a variety of chess positions. Figure 3-6 shows the results of our study, which confirm the results from the **knary** synthetic benchmarks. The curve shown is the best fit to $T_P = c_1(T_1/P) + c_\infty(T_\infty)$, where $c_1 = 1.067 \pm 0.0141$ and $c_\infty = 1.042 \pm 0.0467$ with 95 percent confidence. The R^2 correlation coefficient of the fit is 0.9994, and the mean relative error is 4.05 percent.

Indeed, as some of us were developing and tuning heuristics to increase the performance of ⋆Socrates, we used work and critical path as our measures of progress. This methodology let us avoid being trapped by the following interesting anomaly. We made an "improvement" that sped up the program on 32 processors. From our measurements, however, we discovered that it was faster only because it saved on work at the expense of a much longer critical path. Using the simple model $T_P = T_1/P + T_\infty$, we concluded that on a 512-processor machine, which was our platform for tournaments, the "improvement" would yield a loss of performance, a fact that we later verified. Measuring work and critical path enabled us to use experiments on a 32-processor machine to improve our program for the 512-processor

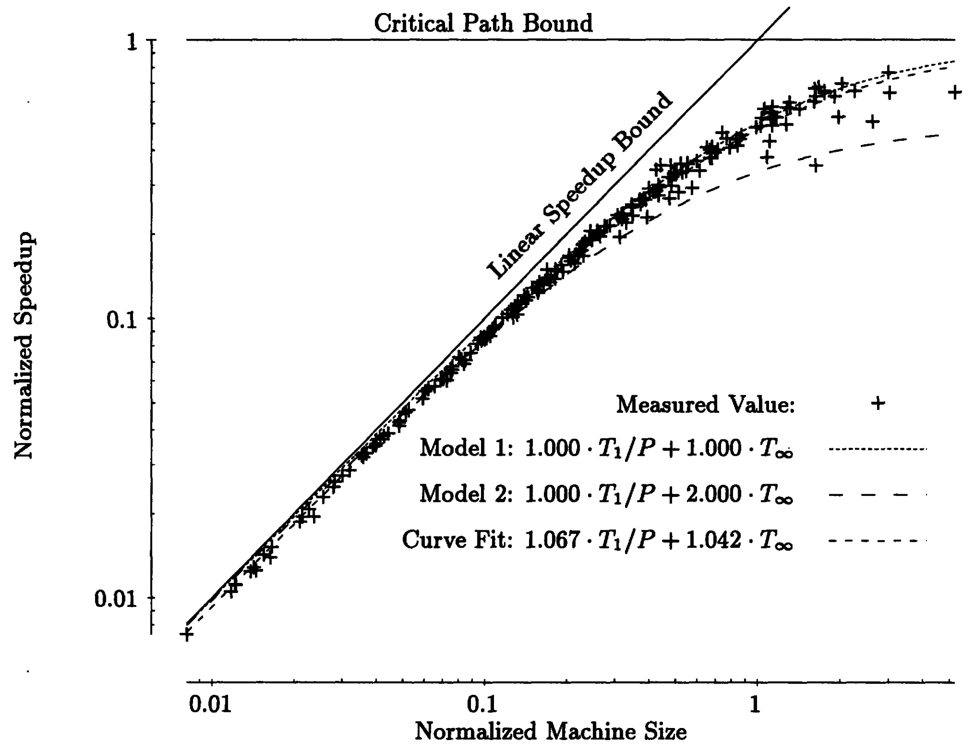

Figure 3-6: Normalized speedups for the ⋆Socrates chess program.

machine, but without using the 512-processor machine, on which computer time was scarce.

3.6 Theoretical Analysis of the Cilk-1 Scheduler

In this section we use algorithmic analysis techniques to prove that for the class of "fully strict" Cilk-1 programs, Cilk-1's work-stealing scheduling algorithm is efficient with respect to space, time, and communication. A *fully strict* program is one for which each thread sends arguments only to its parent's successor threads. In the analysis and bounds of this section, we further assume that each thread spawns at most one successor thread. For this class of programs, we prove the following three bounds on space, time, and communication:

Space The space used by a P-processor execution is bounded by $S_P \le S_1 P$, where S_1 denotes the space used by the serial execution of the Cilk-1 program. This bound is existentially optimal to within a constant factor [BL94].

Time With P processors, the expected execution time, including scheduling overhead, is bounded by $T_P = O(T_1/P + T_\infty)$. Since both T_1/P and T_∞ are lower bounds for any P-processor execution, our expected time bound is within a constant factor of optimal.

Communication The expected number of bytes communicated during a P-processor execution is $O(T_\infty P S_{\max})$, where $S_{\max}$ denotes the largest size of any closure. This bound is existentially optimal to within a constant factor [WK91].

The expected-time bound and the expected-communication bound can be converted into high-probability bounds at the cost of only a small additive term in both cases. Full proofs of these bounds, using generalizations of the techniques developed in [BL94], can be found in [Blu95]. We defer complete proofs and give outlines here.

The space bound can be obtained from a "busy-leaves" property that characterizes the allocated closures at all times during the execution. In order to state this property simply, we first define some terms. We say that two or more closures are *siblings* if they were spawned by the same parent, or if they are successors (by one or more **spawn_next**'s) of closures spawned by the same parent. Sibling closures can be ordered by age: the first child spawned is older than the second, and so on. At any given time during the execution, we say that a closure is a *leaf* if it has no allocated children, and we say that a leaf closure is a *primary leaf* if, in addition, it has no younger siblings allocated. The *busy-leaves property* states that every primary-leaf closure has a processor working on it.

Lemma 1 *Cilk's scheduler maintains the busy-leaves property.*

Proof: Consider the three possible ways that a primary-leaf closure can be created. First, when a thread spawns children, the youngest of these children is a primary leaf. Second, when a thread completes and its closure is freed, if that closure has an older sibling and that sibling has no children, then the older-sibling closure becomes a primary leaf. Finally, when a thread completes and its closure is freed, if that closure has no allocated siblings, then the youngest closure of its parent's successor threads is a primary leaf. The induction follows by observing that in all three of these cases, Cilk's scheduler guarantees that a processor works on the new primary leaf. In the third case we use the important fact that a newly activated closure is posted on the processor that activated it (and not on the processor on which it was residing). ■

Theorem 2 *For any fully strict Cilk program, if S_1 is the space used to execute the program on 1 processor, then with any number P of processors, Cilk's work-stealing scheduler uses* *at most S_1P space.*

Proof: We shall obtain the space bound $S_P \leq S_1P$ by assigning every allocated closure to a primary leaf such that the total space of all closures assigned to a given primary leaf is at most S_1. Since Lemma 1 guarantees that all primary leaves are busy, at most P primary-leaf closures can be allocated, and hence the total amount of space is at most S_1P.

The assignment of allocated closures to primary leaves is made as follows. If the closure is a primary leaf, it is assigned to itself. Otherwise, if the closure has any allocated children, then it is assigned to the same primary leaf as its youngest child. If the closure is a leaf but has some younger siblings, then the closure is assigned to the same primary leaf as its youngest sibling. In this recursive fashion, we assign every allocated closure to a primary leaf. Now, we consider the set of closures assigned to a given primary leaf. The total space of these closures is at most S_1, because this set of closures is a subset of the closures that are allocated during a 1-processor execution when the processor is executing this primary leaf, which completes the proof. ∎

We are now ready to analyze execution time. Our strategy is to mimic the theorems of [BL94] for a more restricted model of multithreaded computation. As in [BL94], the bounds assume a communication model in which messages are delayed only by contention at destination processors, but no assumptions are made about the order in which contending messages are delivered [LAB93]. For technical reasons in our analysis of execution time, the critical path is calculated assuming that all threads spawned by a parent thread are spawned at the end of the parent thread.

In our analysis of execution time, we use an accounting argument. At each time step, each of the P processors places a dollar in one of three buckets according to its actions at that step. If the processor executes an instruction of a thread at the step, it places its dollar into the WORK bucket. If the processor initiates a steal attempt, it places its dollar into the STEAL bucket. Finally, if the processor merely waits for a steal request that is delayed by contention, then it places its dollar into the WAIT bucket. We shall derive the running time bound by upper bounding the dollars in each bucket at the end of the computation, summing these values, and then dividing by P, the total number of dollars put into buckets on each step.

Lemma 3 *When the execution of a fully strict Cilk computation with work T_1 ends, the* WORK *bucket contains T_1 dollars.*

Proof: The computation contains a total of T_1 instructions. ■

Lemma 4 *When the execution of a fully strict Cilk computation ends, the expected number of dollars in the* WAIT *bucket is less than the number of dollars in the* STEAL *bucket.*

Proof: Lemma 5 of [BL94] shows that if P processors make M random steal requests during the course of a computation, where requests with the same destination are serially queued at the destination, then the expected total delay is less than M. ■

Lemma 5 *When the P-processor execution of a fully strict Cilk computation with critical-path length T_∞ and for which each thread has at most one successor ends, the expected number of dollars in the* STEAL *bucket is $O(PT_\infty)$.*

Proof sketch: The proof follows the delay-sequence argument of [BL94], but with some differences that we shall point out. Full details can be found in [Blu95].

At any given time during the execution, we say that a thread is *critical* if it has not yet been executed but all of its predecessors in the dag have been executed. For this argument, the dag must be augmented with "ghost" threads and additional edges to represent implicit dependencies imposed by the Cilk scheduler. We define a *delay sequence* to be a pair (P, s) such that P is a path of threads in the augmented dag and s is a positive integer. We say that a delay sequence (P, s) *occurs* in an execution if at least s steal attempts are initiated while some thread of P is critical.

The next step of the proof is to show that if at least s steal attempts occur during an execution, where s is sufficiently large, then some delay sequence (P, s) must occur. That is, there must be some path P in the dag such that each of the s steal attempts occurs while some thread of P is critical. We do not give the construction here, but rather refer the reader to [Blu95, BL94] for directly analogous arguments.

The last step of the proof is to show that a delay sequence with $s = \Omega(PT_\infty)$ is unlikely to occur. The key to this step is a lemma, which describes the structure of threads the processors' ready pools. This structural lemma implies that if a thread is critical, it is the next thread to be stolen from the pool in which it resides. Intuitively, after P steal attempts, we expect one of these attempts to have targeted the processor in which the critical thread of interest resides. In this case, the critical thread will be stolen and executed, unless, of course, it has already been executed by the local processor. Thus, after PT_∞ steal attempts, we expect all threads on P to have been executed. The delay-sequence argument formalizes this intuition. Thus, the expected number s of dollars in the STEAL bucket is at most $O(PT_\infty)$. ■

Theorem 6 *Consider any fully strict Cilk computation with work T_1 and critical-path length T_∞ such that every thread spawns at most one successor. With any number P of processors, Cilk's work-stealing scheduler runs the computation in expected time $O(T_1/P+T_\infty)$.*

Proof: We sum the dollars in the three buckets and divide by P. By Lemma 3, the WORK bucket contains T_1 dollars. By Lemma 4, the WAIT bucket contains at most a constant times the number of dollars in the STEAL bucket, and Lemma 5 implies that the total number of dollars in both buckets is $O(PT_\infty)$. Thus, the sum of the dollars is $T_1 + O(PT_\infty)$, and the bound on execution time is obtained by dividing by P. ■

In fact, it can be shown using the techniques of [BL94] that for any $\epsilon > 0$, with probability at least $1 - \epsilon$, the execution time on P processors is $O(T_1/P + T_\infty + \lg P + \lg(1/\epsilon))$.

Theorem 7 *Consider any fully strict Cilk computation with work T_1 and critical-path length T_∞ such that every thread spawns at most one successor. For any number P of processors, the total number of bytes communicated by Cilk's work-stealing scheduler has expectation $O(PT_\infty S_{\max})$, where $S_{\max}$ is the size in bytes of the largest closure in the computation.*

Proof: The proof follows directly from Lemma 5. All communication costs can be associated with steals or steal requests, and at most $O(S_{\max})$ bytes are communicated for each successful steal. ■

In fact, for any $\epsilon > 0$, the probability is at least $1 - \epsilon$ that the total communication incurred is $O(P(T_\infty + \lg(1/\epsilon))S_{\max})$.

The analysis and bounds we have derived apply to fully strict programs in the case when each thread spawns at most one successor. In [Blu95], the theorems above are generalized to handle situations where a thread can spawn more than one successor.

3.7 Conclusion

To produce high-performance parallel applications, programmers often focus on communication costs and execution time, quantities that are dependent on specific machine configurations. We argue that a programmer should think instead about work and critical path, abstractions that can be used to characterize the performance of an algorithm independent of the machine configuration. Cilk-1 provides a programming model in which work and critical path are observable quantities, and it delivers guaranteed performance as a function of these quantities. Work and critical path have been used in the theory community for years to analyze parallel algorithms [KR90]. Blelloch [Ble92] has developed a performance model for data-parallel computations based on these same two abstract measures. He cites many advantages of such a model over machine-based models. Cilk-1 provides a similar performance model for the domain of asynchronous, multithreaded computation.

Although Cilk-1 offers performance guarantees, its capabilities are somewhat limited. Programmers find its explicit continuation-passing style to be onerous. The higher level primitives described in Chapter 5 address this concern. Cilk-1 is good at expressing and executing dynamic, asynchronous, tree-like, MIMD computations, but it is not ideal for more traditional parallel applications that can be programmed effectively in, for example, a message-passing, data-parallel, or single-threaded, shared-memory style. To partially address this inadequacy, we have added "dag-consistent" shared memory to the Cilk system, which allows programs to operate on shared memory without costly communication or hardware support. This addition is described in Chapter 6.

Chapter 4

The ⋆Socrates Parallel Chess Program

The Cilk-1 system described in the previous chapter is a powerful system which allows a programmer to write complicated multithreaded programs and have them executed efficiently. The system was intended to have the flexibility to express programs with fairly complex control structures. Up to this point, however, we had only written applications with relatively simple control structures. For most of our programs the Cilk-1 portion consisted mainly of one Cilk thread which was recursively called many times. We therefore wanted to write a large, challenging application with complicated control dependencies, preferably one which would be difficult to express in other parallel programming paradigms. Such a program would not only showcase the power of Cilk, but would also stress the runtime system and the language, thereby identifying any of weaknesses Cilk may have.

We chose computer chess because it met all these criteria, and because it was an interesting application in its own right. Also, since we would test the program in actual competitions against both humans and computers, we would be forced to implement the best algorithms, not just whatever happened to be easiest to implement in our system. Our chess program, ⋆Socrates, uses the Jamboree [Kus94] algorithm to perform a parallel game-tree search. This search algorithm has a complex control structure which is nondeterministic and performs speculative computations, some of which need to be killed off before completing. In order to obtain good performance during this search, we use several mecha-

Part of this work was reported on by Kuszmaul and myself in an earlier article [JK94].

nisms not directly provided by Cilk, such as aborting computations and directly accessing the active message layer to implement a global transposition table distributed across the processors. The initial version of ⋆Socrates was implemented during the time PCM was evolving into the Cilk system and our work on this program led to several modifications to the Cilk system which gave the user more control over the execution of his program.

Many people contributed to the ⋆Socrates chess program. Robert Blumofe, Don Dailey, Michael Halbherr, Larry Kaufman, Bradley Kuszmaul, Charles Leiserson, and I contributed to the ⋆Socrates code itself. Don Dailey and Larry Kaufman, then of Heuristic Software Inc., wrote the serial Socrates program on which ⋆Socrates is based, and Don Dailey twice joined us at MIT to improve the chess knowledge in the program. I initially implemented the search algorithm in Cilk, and then took a lead role in tuning and testing all aspects of the system (including the search code, the chess code and the runtime system). At various times Bradley Kuszmaul and I took the lead in bringing the whole system together. Bradley Kuszmaul also invented the Jamboree search algorithm, which was first used in his StarTech program. Robert Blumofe implemented the transposition table. Michael Halbherr originally implemented the abort code. In addition, Robert Blumofe, Matteo Frigo, Michael Halbherr, Bradley Kuszmaul, Charles Leiserson, Keith Randall, Rolf Riesen, Yuli Zhou, and I contributed to the various PCM and Cilk runtime systems on which ⋆Socrates runs.

4.1 Introduction

Computer chess provides a good testbed for understanding dynamic MIMD-style computations. The parallelism in computer chess is derived from a dynamic expansion of a highly irregular game-tree, which makes computer chess difficult to express, for example, as a data-parallel program. To investigate how to program this sort of dynamic MIMD-style application, we engineered a parallel chess program called ⋆Socrates (pronounced "Star-Socrates".) The program, based on Heuristic Software's serial Socrates program, has an informally estimated rating of over 2400 USCF. ⋆Socrates, running on the 512-node CM-5 at the National Center for Supercomputing Applications (NCSA) at the University of Illinois, tied for third place in the 1994 ACM International Computer Chess Championship held at the end of June 1994 in Cape May, New Jersey. Cilk and ⋆Socrates were later ported to the Intel Paragon in March 1995, and running on Sandia National Laboratories' 1824-node

Paragon, ⋆Socrates finished second in the 1995 World Computer Chess Championship.

⋆Socrates is, in part, a continuation of earlier work performed here on the StarTech [Kus94] chess program. StarTech was based on Hans Berliner's serial Hitech[BE89] program. Although ⋆Socrates and StarTech are based on different serial programs and do not share any code, ⋆Socrates borrowed techniques originally developed for StarTech, such as the basic search algorithm. A major difference between the two is that in StarTech, the chess and scheduling algorithms were all wrapped together in a single piece of code. The work on ⋆Socrates was intended in part to show that the chess program could be separated from the problems of scheduling and load balancing and still execute efficiently. ⋆Socrates uses Cilk to address the scheduling problem, allowing the chess code to focus on only those issues which are unique to a chess program.

It was not clear from the outset how to predict the performance of a parallel chess program. Chess programs search a dynamically generated tree, and obtain their parallelism from that tree. Different branches of the tree have vastly different amounts of total work and available parallelism. ⋆Socrates uses large global data structures, performs speculative computations, and is nondeterministic. But we wanted predictable performance. For example, if one develops a program on a small machine, one would like to be able to instrument the program and predict how fast it will run on a big machine. How can predictable performance be salvaged from a program with these characteristics? We showed in the previous chapter that under certain assumptions the run time of a Cilk program can be predicted from the total work W and the critical path length C. But chess violates these assumptions, so it was not clear how well the scheduler would perform.

For most algorithms, the values of W and C depend on the parallel algorithm, and not on the scheduler. But for speculative computations, such as our game-tree search algorithm, the values of W and C are partially dependent on scheduling decisions made by the scheduler. Our work on ⋆Socrates led to several modifications to the Cilk system which gave the user additional control over these scheduling decisions.

This chapter explains how we implemented ⋆Socrates in Cilk such that we achieved predictable, efficient performance. Section 4.2 describes the Jamboree game-tree search algorithm and presents some analytical results describing the performance of Jamboree search. The modifications made to Cilk in order to run the chess program are described in Section 4.3. In Section 4.4 we outline several other mechanisms that were needed to implement a parallel chess program. Section 4.5 describes the performance of the ⋆Socrates program and shows that a chess program can execute efficiently when the scheduler is independent of the chess code. We make some concluding remarks in Section 4.6.

4.2 Parallel Game Tree Search

The ⋆Socrates chess program uses an efficient parallel game-tree search algorithm called "Jamboree" search [Kus94]. In this section we explain Jamboree search, starting with the basics of negamax search and serial α-β search, and present some analytical performance results for the algorithm.

The basic idea behind Jamboree search is to do the following operations on a position in the game tree that has k children:

- The value of the first child of the position is determined (by a recursive call to the search algorithm.)
- Then, in parallel, all of the remaining $k-1$ children are tested to verify that they are not better alternatives than the first child.
- Each child that turns out to be better than the first child is searched in turn to determine which is the best.

If the move ordering is best-first, i.e., the first move considered is always better than the other moves, then all of the tests succeed, and the position is evaluated quickly and efficiently. We expect that the tests will usually succeed, because the move ordering is often best-first due the the application of several chess-specific move-ordering heuristics.

4.2.1 Negamax Search Without Pruning

Before delving into the details of the Jamboree algorithm, let us review the basic search algorithms that are applicable to computer chess. (Readers who are familiar with the serial game tree search algorithms may wish to skip directly ahead to the description of the Jamboree algorithm in Section 4.2.4.) Most chess programs use some variant of negamax tree search to evaluate a chess position. The goal of the negamax tree search is to compute the value of position p in a tree T_p rooted at position p. The value of p is defined according

```
(N1)  Define negamax($p$) as
(N2)      If $n$ is a leaf then return static_eval($n$).
(N3)      Let $\vec{c} \leftarrow$ the children of $n$, and
(N4)          $b \leftarrow -\infty$.
(N5)        For $i$ from 0 below $|\vec{c}|$ do:
(N6)            Let $s \leftarrow -$negamax($\vec{c}_i$).       ;; Recursive Search
(N7)                if $s > b$ then set $b \leftarrow s$.       ;; New best score
(N8)          enddo
(N9)        return $b$.
```

Figure 4-1: Algorithm **negamax**.

to the negamax formula:

$$v_p = \begin{cases} \texttt{static\_eval(p)} & \text{if } p \text{ is a leaf in } T_p\text{, and} \\ \max\{-v_c : c \text{ a child of } p \text{ in } T_p\} & \text{if } p \text{ is not a leaf.} \end{cases}$$

The negamax formula states that the best move for player A is the move that gives player B, who plays the best move from B's point of view, the worst option. If there are no moves, then we use a static evaluation function. Of course, no chess program searches the entire game tree. Instead some limited game tree is searched using an imperfect static evaluation function. Thus, we have formalized the chess knowledge as T_p, which tells us what tree to search, and `static_eval`, which tells us how to evaluate a leaf position.

The naive Algorithm **negamax** shown in Figure 4-1 computes the negamax value v_p of position p by searching the entire tree rooted at p. It is easy to make Algorithm **negamax** into a parallel algorithm, because there are no dependencies between iterations of the *for* loop of Line (N5). One simply changes the *for* loop into a parallel loop. But negamax is not a efficient serial search algorithm, and thus, it makes little sense to parallelize it.

4.2.2 Alpha-Beta Pruning

The most efficient serial algorithms for game-tree search all avoid searching the entire tree by proving that certain subtrees need not be examined. In this section we review the α-β serial search algorithm in preparation for the explanation of how the Jamboree parallel search algorithm works.

An example of how pruning can reduce the size of a game tree that is searched can be seen in the chess position of Figure 4-2. Suppose White has determined that it can win

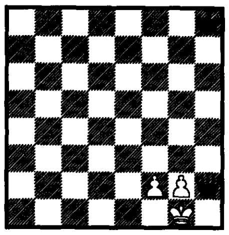

Figure 4-2: White to move and win. In this position, White need not consider all of Black's alternatives to 40. ♔f1, since almost any move Black makes will keep the queen, a worse outcome than just taking the queen with 40. ♔×h2.

```
(A1)   Define absearch($n, \alpha, \beta$) as
(A2)       If $n$ is a leaf then return static_eval($n$).
(A3)       Let $\vec{c} \leftarrow$ the children of $n$, and
(A4)           $b \leftarrow -\infty$.
(A5)         For $i$ from 0 below $|\vec{c}|$ do:
(A6)             Let $s \leftarrow -$absearch($\vec{c}_i, -\beta, -\alpha$).
(A7)                 If $s \geq \beta$ then return $s$.          ;; Fail High
(A8)                 If $s > \alpha$ then set $\alpha \leftarrow s$.       ;; Raise $\alpha$
(A9)                 If $s > b$ then set $b \leftarrow s$.
(A10)          enddo
(A11)        return $b$.
```

Figure 4-3: Algorithm **absearch**.

Black's queen with 40. ♔×h2. White's other legal move 40. ♔f1 fails to capture the queen. White does not need to consider every possible way for Black's queen to escape. Any one of a number of possibilities suffices. Thus, White can stop thinking about the move without having exhaustively searched all of Black's options.

The idea of pruning subtrees that do not need to be searched is embodied in the serial α-β search algorithm [KM75], which computes the negamax score for a node without actually looking at the entire search tree. The algorithm is expressed as a recursive subroutine with two new parameters α and β. If the value of any child, when negated, is as great as β, then the value of the parent is no less than β, and we say that the parent *fails high.* If the values of all of the children, when negated, are less than or equal to α, then the value of the parent is no greater than α, and we say that the parent *fails low.*

Procedure absearch is shown in Figure 4-3. When Procedure absearch is called, the parameters α and β are chosen so that if the value of a node is not greater than α and less than β, then we know that the value of the node cannot affect the negamax value of the root of the entire search tree. After the score is returned from the subsearch on Line (A6), the algorithm, on Line (A7), checks to see if the negated score is as great as β. If so, we know that the value of the node is at least as great as β and we can skip searching the remaining children; the node has failed high. Just because one of the children has a negated score less than α, however, does not mean that some other child might not be within the α-β window. The algorithm can only fail low after considering all of the children.

The α-β algorithm can substantially reduce the size of the tree searched. The α-β algorithm works best if the best moves are considered first, because if any move can make the position fail high, then certainly the best move can make the position fail high. Knuth and Moore [KM75] show that for searches of a uniform best-ordered tree of height H and degree D, the α-β algorithm searches only $O(\sqrt{D^H})$ leaves instead of D^H leaves.

For any $k \geq 0$, before searching the $(k+1)$st child, the α-β algorithm obtains the value of the kth child and possibly uses that value to adjust α or return immediately. This dependency between finishing the kth child and starting the $(k+1)$st child completely serializes the α-β search algorithm.[1]

4.2.3 Scout Search

For a parallel chess program, we need an algorithm that both effectively prunes the tree and can be parallelized. We started with a variant on serial α-β search, called *Scout* search, and modified it to be a parallel algorithm. This section explains the Scout search algorithm.

Figure 4-4 shows the serial Scout search algorithm, which is due to J. Pearl [Pea80]. Procedure **scout** is similar to Procedure **absearch**, except that when considering any child that is not the first child, a *test* is first performed to determine if the child is no better a move than the best move seen so far. If the child is no better, the test is said to *succeed.* If the child is determined to be better than the best move so far, the test is said to *fail,* and the child is searched again *(valued)* to determine its true value.

The Scout algorithm performs tests on positions to see if they are greater than or less than a given value. A test is performed by using an empty-window search on a position. For integer scores one uses the values $(-\alpha - 1)$ and $(-\alpha)$ as the parameters of the recursive

[1]R. Finkel and J. Fishburn showed that if the serialization implied by α-β pruning is ignored by a parallel program, then it will achieve only $\sqrt{P}$ speedup on P processors [FF82].

```
(S1)   Define scout$(n, \alpha, \beta)$ as
(S2)       If $n$ is a leaf then return static_eval$(n)$.
(S3)       Let $\vec{c} \leftarrow$ the children of $n$, and
(S4)           $b \leftarrow -$scout$(c_0, -\beta, -\alpha)$.
(S5)         ;; The first child's valuation may cause this node to fail high.
(S6)         If $b \geq \beta$ then return $b$.
(S7)         If $b > \alpha$ then set $\alpha \leftarrow b$.
(S8)         For $i$ from 1 below $|\vec{c}|$ do:                ;; the rest of the children
(S9)             Let $s \leftarrow -$scout$(\vec{c}_i, -\alpha - 1, -\alpha)$.   ;; Test
(S10)                If $s > b$ then set $b \leftarrow s$.
(S11)                If $s \geq \beta$ then return $s$.           ;; Fail High
(S12)                If $s > \alpha$ then                        ;; Test failed
(S13)                    Set $s \leftarrow -$scout$(\vec{c}_i, -\beta, -\alpha)$.  ;; Research for value
(S14)                    If $s \geq \beta$ then return $s$.       ;; Fail High
(S15)                    If $s > \alpha$ then set $\alpha \leftarrow s$.
(S16)                    If $s > b$ then set $b \leftarrow s$.
(S17)            enddo
(S18)        return $b$.
```

Figure 4-4: Algorithm **scout**.

search, as shown on Line (S9). A child is tested to see if it is worse than the best move so far, and if the test fails on Line (S12) (i.e., the move looks like it might be better than the best move seen so far), then the child is valued, on Line (S13), using a nonempty window to determine its true value.

If it happens to be the case that $\alpha + 1 = \beta$, then Line (S13) never executes because $s > \alpha$ implies $s \geq \beta$, which causes the *return* on Line (S11) to execute. Consequently, the same code for Algorithm **scout** can be used for the testing and for the valuing of a position.

Line S10, which raises the best score seen so far according to the value returned by a test, is necessary to insure that if the test fails low (i.e., if the test succeeds), then the value returned is an upper bound to the score. If a test were to return a score that is not a proper bound to its parent, then the parent might return immediately with the wrong answer when the parent performs the check of the returned score against β on Line S11.

A test is typically cheaper to execute than a valuation because the α-β window is smaller, which means that more of the tree is likely to be pruned. If the test succeeds, then algorithm **scout** has saved some work, because testing a node is cheaper than finding its exact value. If the test fails, then **scout** searches the node twice and has squandered some work. Algorithm **scout** bets that the tests will succeed often enough to outweigh the extra

```
(J1)   Define jamboree($n, \alpha, \beta$) as
(J2)       If $n$ is a leaf then return static_eval($n$).
(J3)       Let $\vec{c} \leftarrow$ the children of $n$, and
(J4)           $b \leftarrow -$jamboree($c_0, -\beta, -\alpha$).
(J5)         If $b \geq \beta$ then return $b$.
(J6)         If $b > \alpha$ then set $\alpha \leftarrow b$.
(J7)         In Parallel: For $i$ from 1 below $|\vec{c}|$ do:
(J8)             Let $s \leftarrow -$jamboree($\vec{c}_i, -\alpha - 1, -\alpha$).
(J9)                 If $s > b$ then set $b \leftarrow s$.
(J10)                If $s \geq \beta$ then abort-and-return $s$.
(J11)                If $s > \alpha$ then
(J12)                    Wait for the completion of all previous iterations
(J13)                         of the parallel loop.
(J14)                    Set $s \leftarrow -$jamboree($\vec{c}_i, -\beta, -\alpha$). ;; Research for value
(J15)                    If $s \geq \beta$ then abort-and-return $s$.
(J16)                    If $s > \alpha$ then set $\alpha \leftarrow s$.
(J17)                    If $s > b$ then set $b \leftarrow s$.
(J18)            Note the completion of the $i$th iteration of the parallel loop.
(J19)          enddo
(J20)        return $b$.
```

Figure 4-5: Algorithm jamboree.

cost of any nodes that must be searched twice, and empirical evidence [Pea80] justify its dominance as the search algorithm of choice in modern serial chess-playing programs.

4.2.4 Jamboree Search

The Jamboree algorithm, shown in Figure 4-5, is a parallelized version of the Scout search algorithm. The idea is that all of the testing of the children is done in parallel, and any tests that fail are sequentially valued. A parallel loop construct, in which all of the iterations of a loop run concurrently, appears on Line (J7). Some synchronization between various iterations of the loop appears on Lines J12 and J18. We sequentialize the full-window searches for values, because, while we are willing to take a chance that an empty window search will be squandered work, we are not willing to take the chance that a full-window search (which does not prune very much) will be squandered work. Such a squandered full-window search could lead us to search the entire tree, which is much larger than the pruned tree we want to search.

The *abort-and-return* statements that appear on Lines J10 and J15 return a value from

Procedure `jamboree` and abort any of the children that are still running. Such an abort is needed when the procedure has found a value that can be returned, in which case there is no advantage to allowing the procedure and its children to continue to run, using up processor and memory resources. The abort causes any children that are running in parallel to abort their children recursively, which has the effect of deallocating the entire subtree.

The actual search algorithm used in ⋆Socrates also includes some *forward pruning* heuristics that prune a deep search based on a shallow preliminary search. The idea is that if the shallow search looks really bad, then most of the time a deep search will not change the outcome. Forward pruning techniques have lately been shown to be extremely powerful, allowing programs running on single processors to beat some of the best humans at chess. The serial Socrates program uses such a scheme, and so does ⋆Socrates. In the ⋆Socrates version of Jamboree search, we first perform the preliminary search, then we search the first child, then we test the remaining children in parallel, and research the failed tests serially.

Parallel search of game-trees is difficult because the most efficient algorithms for game-tree search are inherently serial. We obtain parallelism by performing the tests in parallel, but those tests may not all be necessary in a serial execution order. In order to get any parallelism, we must take the risk of performing extra work that a good serial program would avoid.

Figure 4-6 shows the Jamboree search code transformed into a dataflow graph. This graph clearly shows that all the tests can be executed in parallel, but that only one of the value searches can be executed at a time. This graph was also a useful reference in expressing the Jamboree search code as a Cilk program, since it gives an idea of how the procedure can be broken up into separate threads.

4.3 Using Cilk for Chess Search

In the following two sections we describe the implementation of ⋆Socrates using Cilk-1. These sections are an interesting case study in implementing a large, multithreaded, speculative application. As mentioned in the introduction, ⋆Socrates is a parallelization of a serial chess program. Much of the code, including the static evaluator, is identical in the parallel and the serial versions and is not discussed here. Instead, we focus on the portions of the code which were written specifically for the parallel version.

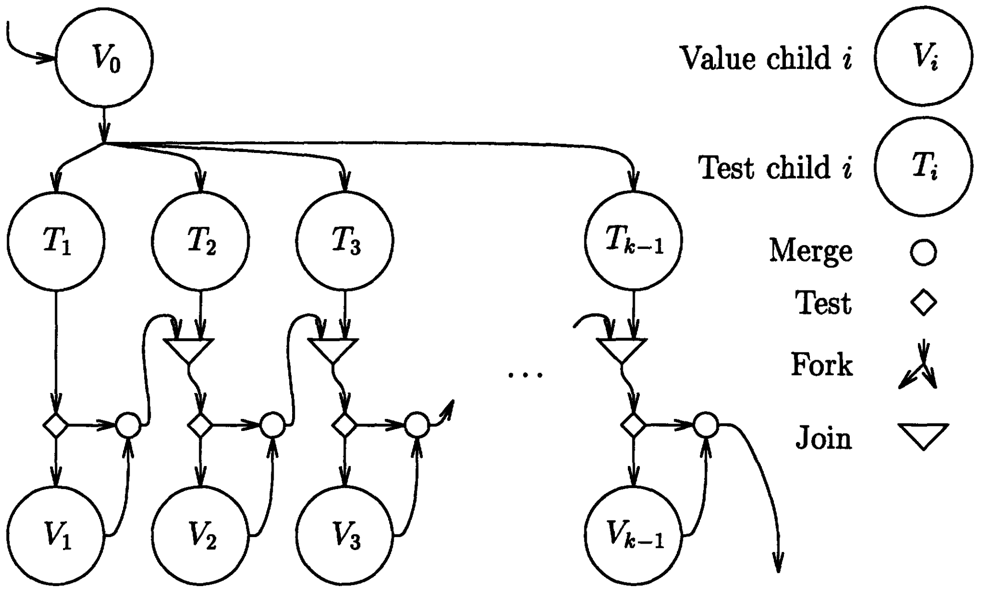

Figure 4-6: The dataflow graph for Jamboree search. First Child 0 is searched to determine its value, then the rest of the children are tested in parallel to try to prove that they are worse choices than Child 0, and then each of the children that fail their respective tests are sequentially researched. Compare this description of the Jamboree algorithm to the textual description in Figure 4-5.

The actual Cilk search code is too large to include here, so instead we show the dag of threads (potentially) created during a search. This dag is shown in Figure 4-7. Note that the ovals labeled $TEST_i$ and VAL_i are recursive calls of the search algorithm which perform either a limited search (a test) or a full search (a value search); so these ovals correspond to entire sub-dags, not just individual threads. Most of this dag is analogous to the dataflow graph for Jamboree search which was shown in Figure 4-6. The threads labeled with the diamond symbol are the threads called `diamond`, so named because they perform the function that diamond nodes did in the dataflow graph of Figure 4-6. The only difference between the algorithm shown here and that implemented by the earlier dataflow graph is that the ⋆Socrates search algorithm first makes a recursive call to the search algorithm to perform a "null move search". This search is a reduced depth search that is part of the forward pruning algorithm which decides if the side to move has an advantage large enough that the search can probably be safely ended here. This recursive call is shown by the oval labeled VAL_{nul} in Figure 4-7. Most of the other differences between the dataflow graph and the Cilk dag are due to one dataflow node being expanded into several Cilk threads. For

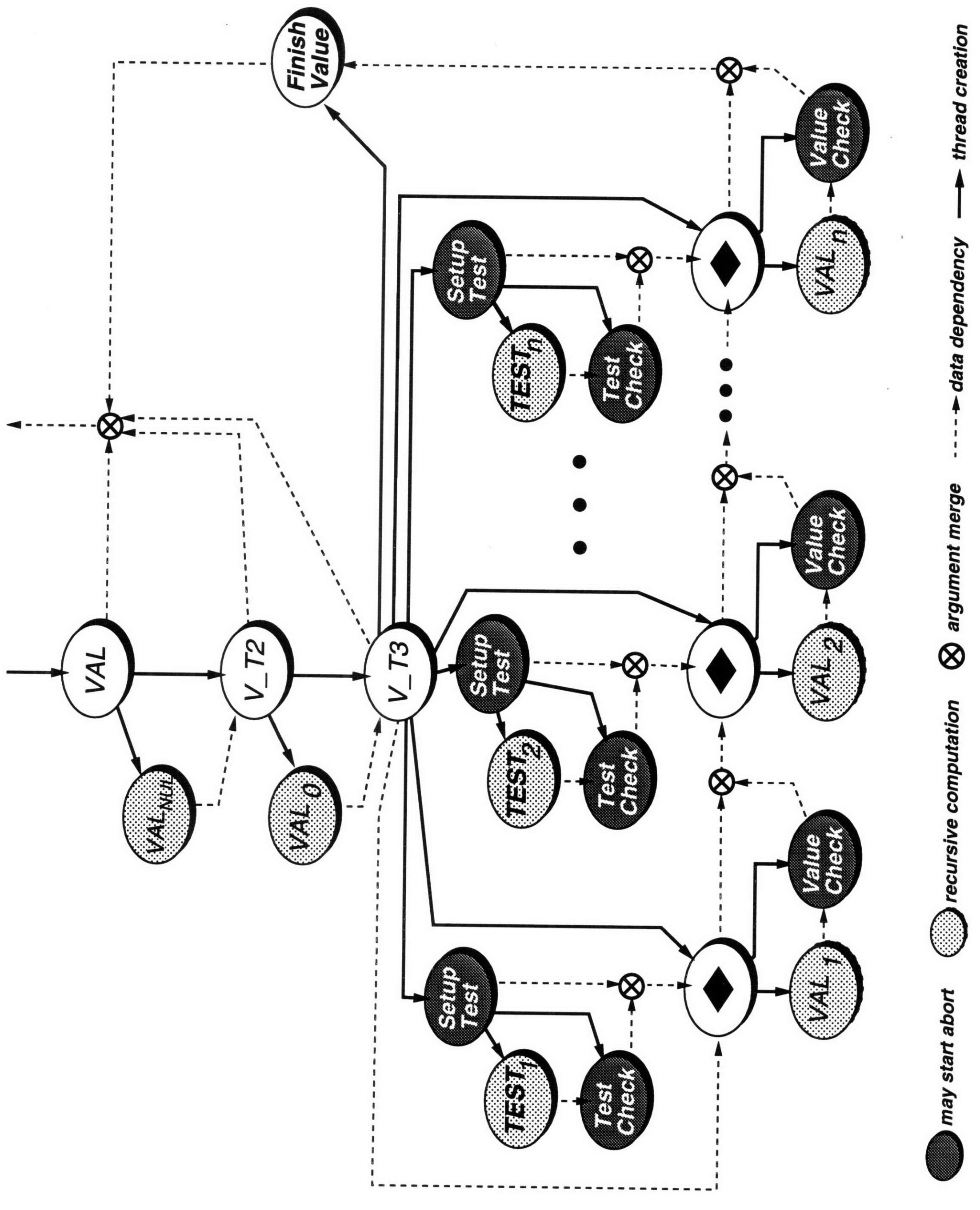

Figure 4-7: This dag shows the dag created by ⋆Socrates when performing a value search. The ovals labeled $TEST_i$ and VAL_i are recursive calls of the search algorithm. Other ovals correspond to Cilk threads.

example, the dataflow graph just has a single node, $TEST_i$, for the test of the ith child. In Cilk this node is broken into several threads so that immediately after the test completes the `test_check` thread can run to check if we are able to abort the rest of the search. These details were ignored in the earlier dataflow graph.

The rest of this section focuses on those aspects of ⋆Socrates related to efficiently implementing the parallel search algorithm. The following section focuses on parallelizing other aspects of ⋆Socrates. Our work on implementing ⋆Socrates's search algorithm lead to many modifications to the scheduler. The changes we made include implementing migration handlers, implementing priority threads, aborting computations that are in progress, changing the order in which threads are stolen, and adding level waiting. The first version of ⋆Socrates was written when we were still using the original PCM system. Many of the improvements to the runtime system that are described in this section were originally added for ⋆Socrates and were later included in the Cilk-1 system.

4.3.1 Migration Threads

We use a large, variable sized data structure (over 200 bytes) to describe the state of a chess board. In the serial code we pass around pointers to this state structure and copy it only when necessary. In the parallel code we cannot just blindly pass pointers between threads, because if the thread is migrated the pointer will no longer be valid. A naive solution is to copy the state structure into every thread, but this adds a significant overhead to the parallel code. This overhead is especially distasteful when you realize that well under 1% of threads are actually migrated, so most of the copying would be wasted effort.

To solve this problem we use migration threads. Any thread can have a migration thread associated with it. When the scheduler tries to migrate a thread that has an associated migration thread, the scheduler first calls the migration thread. This migration thread will return a new closure which is migrated instead.

Using this mechanism we are able to pass pointers to state structures between threads. Any thread that is passed a state pointer is also given a migration thread which can copy the state into the closure if the thread is stolen. When this closure is stolen the migration thread is run and the migration thread creates a copy of the closure with the state copied into it. This closure is them migrated to the stealing processor. Once the closure arrives, the stolen thread can then be called with a pointer to the copied state structure. This

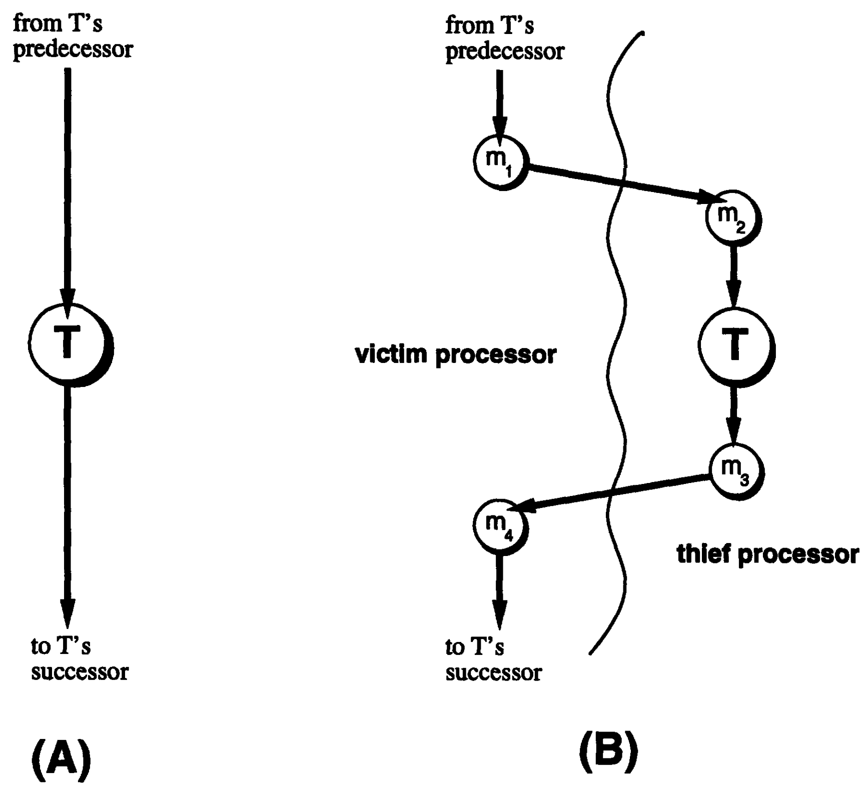

Figure 4-8: This shows how migration threads can be used to insert user code before and after any communication event. Part (A) shows the case where migration does not occur. Part (B) shows the case where migration does occur.

allows the overhead of copying the state to be paid only when it is actually necessary.

Using a migration thread to pass a state structure only works when the stolen thread reads the state structure, but does not write it. Many threads, however, such as the `diamond` and `value_check` threads may update the state structure as well as read it. For these threads, just migrating a copy of the structure when stolen would not be sufficient. Instead, these threads are specified to be *local threads*, which means these threads are not allowed to be stolen. By making these threads local we can guarantee they update the original state structure. For simplicity, many of the threads in ⋆Socrates were made local and only those that start up a test of a child position are stealable.

Migration threads give the user more flexibility than just running a thread on the victim processor before a thread is migrated. Migration threads can be used to run user code before and after every communication, as is shown in Figure 4-8. Part (A) shows the common case: thread T is not stolen and only thread T is run. Part (B) shows the case where thread T is stolen. In this case the system executes the migration thread m_1 on the victim processor. This thread can change the continuation passed to T so that instead of pointing to T's successor, the continuation points to a new closure which will run thread m_4. The closure for thread m_4 is given the original continuation so that thread m_4 can pass the result from T onto T's successor. The migration thread must return a closure to be migrated; this closure can specify that thread m_2, rather than T, be run on the thief processor. When thread m_2 executes it can then run some user code before spawning thread T. As was done on the sending side, m_2 can also splice another thread, m_3, into the path from T to its successor. Although we have described this process as if T were a thread, the same thing can be done where T is an entire subcomputation. This flexibility to run user code at any of these four points was used in ⋆Socrates to implement the abort mechanism described in the next section.

4.3.2 Aborting Computations

In order to implement the Jamboree search algorithm we must be able to abort a computation. When searching a node, if one of its children exceeds β, then the node fails high and the search of the node can be ended without searching the rest of the children. If searches of any other children are already in progress, then those searches should be aborted. The Cilk system has no built-in mechanism for aborting a computation, so this had to be added as user code. Our goal in designing the abort mechanism was to keep it as self contained as possible and to minimize changes to the rest of the code. We wrote the abort mechanism entirely in Cilk code so that it would be able to port easily. Eventually we would like to add support for such a mechanism to Cilk itself.

In order to abort a computation we must first be able to find all of the threads that are working on this computation. To implement this we use ***abort tables*** to link together all the threads working on a computation. When a computation, say A, needs to create several children it first creates an abort table containing an entry for each child of the computation. Each entry in the table keeps track of the status of the children. If a child of A, say B, itself spawns off children, then the entry for B is updated to contain a pointer to the abort table that B creates. Once B and all its children have completed, B's table is deallocated and the entry for B is updated. If a child of A, say C, is stolen, A's entry for C is updated to point to C on its new processor. With this mechanism in place the abort code is able to find all the descendants of any computation. When performing an abort, the abort code does not actually destroy any threads, instead it merely makes a mark in each affected thread's abort table. When a user's thread runs its first action should be to check to see if it has been aborted, and if so skip the rest of its computation. This check allows the user's code to do any cleaning up that may be necessary. (For example, the code may need to free some data structures.)

The abort mechanism provides functions to create, update, and deallocate the abort structures; to check if a thread is aborted; and to start an abort. These mechanisms are implemented independently of the search code. By using these functions and passing around a few pointers to abort tables, the search code was modified to include aborting without too many changes.

One difficulty encountered in implementing the abort tables was in keeping the tables correct when a computation migrates. Our implementation uses migration threads as shown in the previous section to keep track of the state of the computation. When a computation is stolen an abort table is allocated on the stealer's side and the existing abort table is modified to point to it. The difficulty arises because at the time a computation is stolen there is not yet an abort table on the stealer's side to point to. This abort table is not allocated until after the thread begins to run. So instead we create a unique identifier (UID) for each stolen computation, and store that into the abort table. Then on the stealer's side we have a hash table to map the UID into a pointer to the abort table. The protocol for accessing the hash table is quite tricky since there are many cases which require special handling. For example, the network of the CM-5 can reorder messages, therefore we have to handle the case where a message to abort a computation arrives before the thread that will allocate the hash table entry and abort table for that computation. Unfortunately, we did not consider all such possibilities before beginning the design, so getting this mechanism working correctly took longer than anticipated.

Our decision to make the abort mechanism as self contained as possible turned out to be a very good one. The abort code was written back in the early days, when the runtime system was still called PCM. Since becoming Cilk-1, much of the runtime system was completely rewritten, and many low-level data structures were changed. Yet the abort code continued to work. The only changes we had to make were some minor syntax changes when we changed the language. When we ported Cilk-1 and ⋆Socrates to the Paragon, the abort code did not require a single change. Given the difficulty we had getting the original code to work on the CM-5, and given that we did not have access to a debugger on the Paragon, we were quite fortunate to have implemented this code such that it truly was independent of the rest of the system.

4.3.3 Priority Threads

Another change to the Cilk system that was inspired by ⋆Socrates was the addition of priority threads. In the search algorithm there are certain threads that we would like to run as soon as possible, namely the `test_check` and `value_check` threads. These are short threads that receive the result of a search of a child and incorporate that result into the current computation. These threads may update α so we would like this update to occur right away so that future searches can use this updated information. Also, these threads may cause a search to be aborted, and we certainly want this to happen right away. In the original system such a thread T would simply be posted to the bottom of the ready queue, just like every other thread. This thread would then be next in line to execute, but if some other thread was posted before thread T was scheduled, then the execution of T would be indefinitely delayed.

To solve this problem we added priority threads, which are threads that are posted to a single-level priority queue rather than the standard ready queue. Whenever there is a closure in the priority queue the closure at the front of the priority queue will be executed next, before any closure in the standard ready queue. This allows the user to specify threads that should be run as soon as possible.

It is interesting to note that with the provably good scheduler threads like `test_check` and `value_check` must be made priority threads only because ⋆Socrates uses nonstealable threads and nonstealable threads break the guarantees provided by the scheduler. The only time a thread such as `test_check` or `value_check` can be enabled but not executed next is if a `send_argument` from another node is involved. For example, when a test completes the only thread it spawns or enables is its `test_check` thread. When the test is done locally, the `test_check` is placed at the bottom of the ready queue and will normally be executed next. Only if in the meantime a `send_argument` arrives from another node and enables some unrelated thread could the `test_check` not be the next to execute. This situation can only happen when the thread enabled by the `send_argument` is not stealable, because otherwise the thread the **send_argument** enabled would have been stolen away by a provably good steal. Similarly if the test is stolen away, it enables the **test_check** thread via a nonlocal **send_argument**. If the **test_check** thread were not marked nonstealable, it would immediately be stolen back and executed on the remote node.

Priority threads are also used in the implementation of the abort mechanism. As would be expected, they are used to ensure that threads aborting the search are not delayed by threads performing a search.

4.3.4 Steal Ordering

In the original PCM runtime system, the thread queue consists of a single double ended queue. Newly enabled threads are placed at the bottom of the queue, and the local processor takes work out of the bottom as well (i.e. LIFO). When stealing occurs, threads are stolen from the top of the queue (i.e. FIFO). For a tree-shaped computation, the LIFO scheduling allows the computation to proceed locally in a depth-first ordering, thus giving us the same execution order a sequential program would have. When stealing occurs, however, the FIFO steal ordering causes a thread near the top of the tree to be stolen, so a large piece of work is migrated, thus minimizing stealing. Since Jamboree search is a tree-shaped computation, this mechanism works reasonably well.

With this scheduling mechanism, the order in which children are executed depends on whether or not a child is stolen. For most computations this execution order does not matter, but for Jamboree search it does. Execution order has an effect, because if one child causes a search to fail high, the rest of the children do not need to be examined. Our program orders the children such that in the common case where no children are stolen, the children believed to be the best moves are searched first. This order is likely to minimize the total work W, since the best moves are the most likely to beat β, and the once we beat β we fail high and do not need to search any more children. The problem is that when stealing occurs, we steal the child least likely to cause us to fail high.

We would like to steal from the top of the tree, but still steal the child that is most likely to fail high. Such would be the case if the scheduler had the following natural property: If a thread spawns off n children, those children should be executed in the same order regardless of where the children are executed. To add this property we had to modify the scheduler by adding the concept of levels. Each thread in the queue is assigned a level and threads

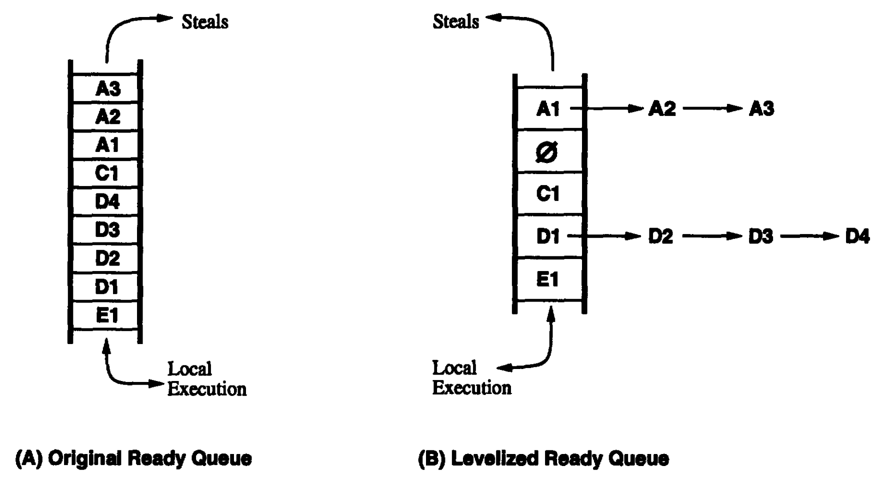

Figure 4-9: This figure shows how the ready queue is implemented. Part (A) shows the original ready queue which was just a double ended queue. Part (B) shows the modified ready queue which is a queue of fifos.

at the same level are executed in a fixed order, regardless of whether they are stolen or executed locally. Between levels, however, scheduling is done as before: We execute locally at the deepest (newest) level and steal from the shallowest (oldest) level. The search code then marks all the children of a computation as being one level deeper than the level at which the computation is currently executing. This strategy gives us exactly the ordering of threads that we want. Figure 4-9 shows a levelized ready queue. Each level is implemented as a first-in-first-out queue, guaranteeing that closures at the same level are executed in the same order regardless of whether the closures are executed locally or are stolen. Adding this to ⋆Socrates reduced the amount of work performed for searching a position and seemed to give a speedup of roughly 20-25%. This idea seemed important enough that we incorporated this mechanism into the Cilk-1 scheduler.

4.3.5 Level Waiting

The final change we made to the scheduler was a further attempt to reduce the extra work being performed by the parallel version. When a processor is searching a board position, P, it spawns off a bunch of children to test. If a processor runs out of children to work on while other children are still being worked on elsewhere, then that processor steals another closure and begins working on the stolen closure.

Consider the case where one (or more) of the children is stolen and the processor finishes the rest of the tests before the test of the stolen child completes. The processor may then be out of work to do[2]. This processor then steals some closure from another processor and begin searching its board position, call it Q. Eventually, the test of the stolen child completes. When this result comes back, it restarts the computation on position P and preempts Q. (Or, depending on the level of each search, Q may preempt P.) We are now in a position where Q, no matter how little work it has, cannot complete until the arbitrarily long computation of P completes. Meanwhile, the computation which spawned Q continues. It may eventually block waiting for Q (and thereby artificially lengthen the critical path) or it may be able to continue, but will search using looser bounds than if Q had completed (and will thereby increase the total work).

To avoid this stalled work we further modified the scheduler. We added "level waiting", a feature which makes uses of the same levels that were described in Section 4.3.4 for optimizing the steal ordering. When a computation spawns children, all the subcomputations are placed at the same level. The level-waiting mechanism simply requires all of these subcomputations to have completed before we may begin any work at a shallower level, or before we can steal. This strategy prevents us from starting, and then preempting, an unrelated search. Implementing this change seemed to give us roughly a 10-15% speedup.

We made this change when we wrote the first version of the chess code running on the CM-5 under PCM. This version did not have the provably good scheduler that is a part of Cilk-1. This modification is actually closely related to the ***busy-leaves*** property described in Section 3.6. At the beginning of this section, we described the situation we were concerned with, namely a case where a search of position Q was preempted in order to continue working on the search of an unrelated position P. In this example the search of Q was effectively a nonbusy leaf. Using the provably good scheduler, such a nonbusy leaf cannot occur: When the search of P is re-enabled, the processor that enables the search of P would steal that search and continued working on it, so P would not interfere with the search of Q. Since the situation that level waiting prevents can not occur with a provably good scheduler, the level-waiting change was removed from the scheduler when we made the scheduler provably good. The level-waiting modification was used during the

[2]The processor is likely to be out of work because none of the children at this level would have been stolen if there were any work earlier in the queue.

1994 tournament, but not afterwards.

Removing this change probably slightly hurt the performance of future versions of ⋆Socrates. In order to obtain the guarantees provided by the provably good scheduler, a program must obey certain constraints. One of these is that the scheduler must be able to migrate any thread to any processor. But as mentioned earlier, many threads are marked nonstealable, which forces them to stay on the same processor on which they were created. Since not all threads can be migrated, the *busy-leaves* property no longer holds, and therefore the situation described above can still occur. Eventually, we expect to rewrite the ⋆Socrates search code such that the performance guarantees of the scheduler do apply. Such a rewrite will put this concern to rest.

4.4 Other Chess Mechanisms

The previous section described issues that arose in getting the search routines to run in Cilk, many of which led to changes in the Cilk system itself. This section describes other aspects of the serial code that had to be modified to run in a parallel system. These aspects include the transposition table, detecting repeated moves, and debugging support.

4.4.1 Transposition Table

Most serial chess programs include a transposition table, which is basically a hash table of previously evaluated positions. After a position is searched, we create (or update) a hash entry for this position. The information stored in this entry includes a depth, a score, a move, and a check key. The depth tells us how deep a search was done, the score tells us the value of the position when searched to that depth, the move tells us what move achieves this score, and the check-key is used for differentiating between two positions which hash to the same entry. Each position has a 64-bit hash key. Part of this key is used to decide what hash entry should be used for the key and part of the key is stored in the hash entry as the check key to distinguish between the many positions which may hash into the same entry.

Before searching a position we first check to see if it is already present in the transposition table with a depth greater than or equal to the depth to which we need to search. If so, then we have a score for this position, and we need not search further. Much of the time when we find a position in the transposition table, the depth is not sufficient for the current search. But even in this case the table is still useful because it tells us the best move found by a shallower search, and often the best move at a shallower depth is still the best move when searched to a deeper depth. By using the move stored in the hash table entry as our predicted best move, we increase our chances of accurately predicting the best move, which, as we saw in Section 4.2, greatly reduces the work and critical path of the computation.

For ⋆Socrates we implemented a distributed transposition table in which entries were hashed across all the processor memories. When a thread begins a search of a position, the first thing it typically does is to lookup that position in the transposition table. We had a choice between implementing a blocking or a nonblocking interface to the table. In a blocking implementation, the thread performing the lookup sends off a lookup request to the appropriate processor and busy-waits until the response arrives, at which point the thread can continue. The obvious disadvantage of blocking is that we waste time busy-waiting. In a nonblocking implementation, we break this thread into several threads. When the time comes to do a lookup, a thread is posted on the processor that holds the entry. This thread performs the lookup and sends the result back to the original processor, which enables a thread that continues the search. This implementation has the advantage that no time is spent busy-waiting during a table lookup. But it has one big disadvantage, namely that it may lead to many searches taking place on the same processor concurrently. Intermixing two or more searches on the same processor can cause both the work and the critical path to increase. To avoid these increases we would have had to modify the scheduler to keep the two computations separate. To avoid the complexity involved in such a modification, we chose to implement a blocking transposition table.

Since there is no way to implement this blocking mechanism using Cilk primitives, we dropped to a lower level and used the Strata active-message library [BB94]. We designed the transposition table such that all accesses are atomic. For example, when a value is to be put into the table, the information about the position is sent to the processor where the entry resides, and that processor updates the entry as required. Alternatively, we could have implemented a nonatomic update by performing a remote read of the entry, modifying the entry, and then doing a remote write. Nonatomic updates would have required more messages and would have required us to lock the entry while the update was in progress, or risk losing some information if two update operations overlapped.

To determine how much the busy-waiting hurts us, we instrumented our code to measure the time spent busy-waiting. Our experiments showed us that the mean time to do a complete lookup was under 1700 cycles, which worked out to about 7% of the execution time. Not all this time is wasted however, while busy-waiting we poll the network so we may spent part of this time responding to arriving messages. But our analysis gives us an upper bound on the cost of busy-waiting.

When we ported the code to the Paragon this lookup time increased, since the Paragon has a larger overhead for using the network. To reduce the impact of doing a global lookup, we do other work while waiting for a lookup to return. In particular, after starting a lookup we perform a static evaluation of the position. This work may be wasted. For example, the lookup may find a valid score. In this case the search is complete and the static evaluation of the position is not needed. But lookups find valid scores less than 10% of the time, and so usually this work is not wasted.

The last aspect of the transposition table we examine is subsumptions. The issue is what, if anything, do we do if two independent searches are concurrently searching the same position (i.e., one search "subsumes" the other). For example, Processor P1 begins a search of Position B, and before it completes and writes its result into the hash table, Processor P2 begins another search of Position B. In this situation, part of the search is being duplicated. In serial code these searches are performed sequentially, so this problem does not occur.

We considered trying to avoid this overhead in the following manner. When a search begins, if the transposition table lookup fails, an entry is created for that position, and it is marked as "search in progress." Then, if another lookup occurs on this position, we know that a search is already being done. We would then have the option of waiting for the earlier search to complete.

We chose not to implement this mechanism, in part because implementing it would have been somewhat complicated. Moreover, it raised several issues of which we had no clear understanding. For example, when we are about to abort a search, is it necessary to first check to see if anyone else is waiting for the results of this search? And if someone is waiting do we still abort the search? Another unanswered question involves how to decide when to wait: If a position is already being searched to depth d, and we want to search it to depth $d - 1$, do we wait for the deeper search? If we don't wait, we are doing extra work, and if we do wait, we may wait much longer than if we had just done it ourselves. In order to estimate how much duplicate work was being performed, we instrumented our program in the following way. Each time we completed a search and were about to write the hash table entry, we first did a hash table lookup to see if we would get a hit if we began the search now. If so, then someone else must have completed a search of this node during the time since we began the search. We found that this occurred less than 1% of the time. This led us to believe that subsumptions were not causing us to waste a significant amount of work. Furthermore, we had implemented a similar mechanism for the earlier StarTech program, and it sometimes sped the program up, and sometimes slowed it down. Consequently, we decided not to implement this mechanism for ⋆Socrates.

4.4.2 Repeated Moves

To fully describe a position in a chess game, we need more than just a description of where each piece is on the board. Some history is needed as well. For example, we need to know whether or not the king has moved. If it has, then we cannot castle, even if the king moves back to its original position. This sort of information can easily be stored in a few bits in the state so maintaining it causes no difficulty.

Other required history can not be stored so easily. In chess if the same position is repeated 3 times, then the game is a draw. Similarly if 50 moves are made by each player without an irreversible move being made, the game is a draw[3]. To handle these cases we need to keep track of all moves since the last irreversible move. (Once an irreversible move is made no earlier position can be repeated.) We keep track of these moves by adding an array of positions to our state structure. This array contains all the positions, represented by their 64-bit hash key, since the last irreversible move.

This array greatly increases the size of the state structure (from about 160 bytes to nearly 1000 bytes). For a serial program the size of the state may not be significant, since the code can just modify and unmodify the same state structure. For parallel code, however, it is often necessary to make copies of the state, and so a large state can slow down the program. To reduce the overhead of copying the state structure, we copy only the meaningful part of the repeated-position array. Since the average length of this portion of

[3]An irreversible move is one which cannot be undone, that is, one which captures a piece or moves a pawn.

the list is quite small (under 2), this copying adds very little overhead.

4.4.3 Debugging

One difficulty in writing a parallel application of this complexity is debugging. Debugging is especially difficult for nondeterministic applications such as ⋆Socrates. In order to make it easier to debug our code, we make liberal use of 'assert' statements. When debugging is turned on, assert statements ensure that conditions the programmer expected to be true, are in fact true. Not only does this methodology cause bugs to be detected sooner, it also helps pinpoint the cause of the bug.

Initially, one of our biggest problems was making sure that the parallel version was working correctly. Most of the code is shared between the parallel and serial versions of the program. Part of the code, in particular the search algorithm, is not. The search algorithm includes not only the basic Jamboree search as described earlier, but also many chess specific heuristics with which we are often experimenting. We were often modifying both the parallel and the serial search algorithms and keeping them consistent was quite error prone. To test if the versions are equivalent, simply running the parallel and serial versions of the code and comparing the results is inadequate, since if the parallel version was only slightly different from the serial version, the two versions would still usually produce the exact same answers. One method we occasionally used in order to test whether both versions were identical was to run the parallel code on one processor and run the serial code and make sure they both searched exactly the same number of positions. Unfortunately, we did not always do this check often enough and at one point so many minor variations had crept in that we wound up spending almost a week trying to make both versions consistent again.

One of the most useful assertions we added was to check at every node of the tree that the results of the parallel code were equivalent to the results of the serial code. In the debugging version of the code, after the search of a position was complete we call the serial code on the same position and compare the results. (We turned the hash table off, since otherwise the serial code simply finds the result in the hash table.) Since the program is nondeterministic, we do not require that the results are identical. Instead, we ensure that both of the returned scores are either both below alpha, both above beta, or identical if between alpha and beta. Due to the large amount of duplicated searching, the resulting code

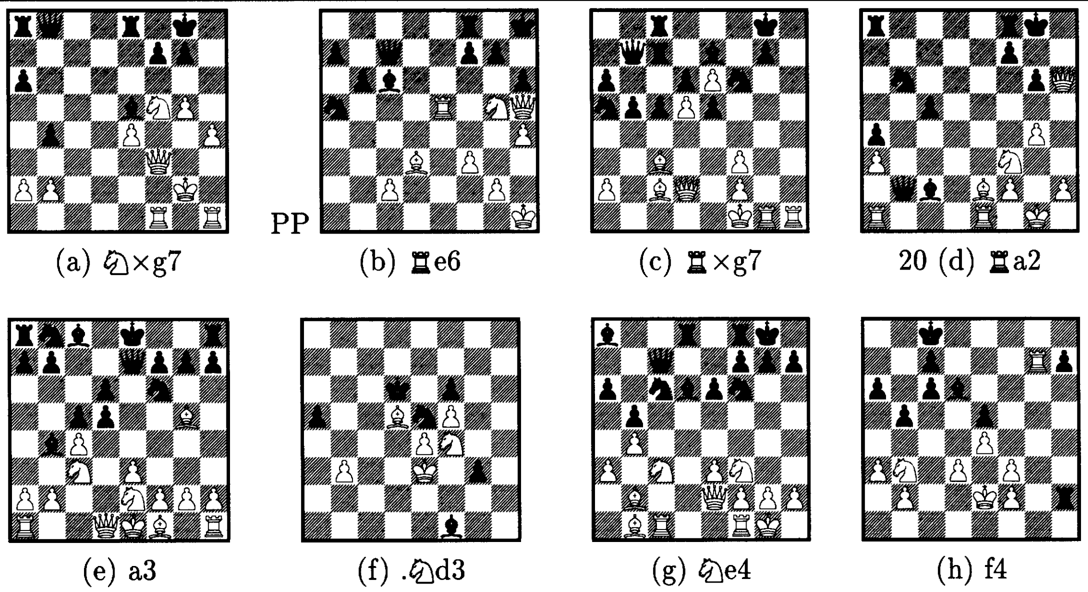

Figure 4-10: The 8 chess positions used in this chapter. Below each position is shown Kaufman's "correct" move for that position. All positions are "White to move", except for Position (f).

was extremely slow. But this version was used only for debugging and was an easy way to detect any differences between the serial and parallel searches and to pinpoint exactly where the differences lay. After we started using this check, keeping both versions identical became much easier. We think this debugging strategy is applicable to many parallel programs, and not just chess.

4.5 Performance of Jamboree Search

Jamboree search is difficult to analyze for arbitrary game trees, because it is difficult to characterize the tree itself, and the tree that is actually searched can depend on how the work is scheduled. Unlike many other applications, the shape of the tree traversed by Jamboree search can be affected by the order of the execution of the work, sometimes increasing the work and sometimes decreasing work. Thus, measurements of "critical path length" and "work" on a particular run may be different than the measurements taken on another run, because the trees themselves are different. Although it is not clear precisely what "critical path" and "work" mean for the speculative Jamboree search, we have found that we can still use the measured critical path length and total work to tune the program.

Our strategy is to measure the critical path and the work on a particular run, and to try to predict the performance from those measurements. We measured the program on the eight problems shown in Figure 4-10. These problems were provided by ⋆Socrates team member L. Kaufman, who is an International Master, For each problem the program was run to various depths up to those that allowed the program to solve the problem by getting the "correct" answer, as identified by Kaufman. We also measured the program running on a variety of different machine sizes. Then we performed a curve fit of the data to a performance model of the form

$$T_{\text{predicted}} = c_1 \cdot \frac{W}{P} + c_\infty \cdot T_\infty.$$

As described in the previous chapter, we found that the performance can be accurately modeled as

$$T \approx (1.067 \pm 0.0141)\frac{W}{P} + (1.042 \pm 0.0467)T_\infty + 0 \tag{4.1}$$

with 95 percent confidence. The R^2 correlation coefficient of the fit is 0.9994, and the mean relative error is 4.05%. To us, these tight error bounds were quite amazing, because chess is a very demanding application. Clearly, there are times during the Jamboree search algorithm when not much parallelism exists. The low coefficients on Equation 4.1 indicate that the program quickly finishes the available work during the times of low parallelism, and when there is much parallelism the program efficiently balances the workload.

We found that the work increases by about a factor of 2 to 3 as the number of processors increases from 1 to 128 processors, and that the critical path length is fairly stable as the number of processors increases. Most of the difficulty of predicting the performance of the chess program comes from the fact that the amount of work is variable. When the program is run on large machines, the processors end up expanding subtrees that are pruned in the serial code. We found that the better the move ordering, the lower the critical path and the less total work is performed. Thus, the move ordering heuristics of a chess program, which are important for serial programs because it reduces the work, are doubly important for our parallel algorithm, because it also decreases the critical path length.

We also found that the critical path does not limit the speedup for our test problems, or for the program running under tournament conditions. By using critical path to understand the parallelism of our algorithm, we are able to make good tradeoffs in our algorithm design. Without such a methodology it can be very difficult to do algorithm design. For example, Feldmann, Monien, and Mysliwietz find themselves changing their Zugzwang chess program to increase the parallelism without really having a good way to measure their changes [FMM93]. They express concern that by serially searching the first child before starting the other children, they may have reduced the available parallelism. Our technique allows us to state that there is sufficient parallelism to keep thousands of processors busy without changing the algorithm. We can conclude that we should try to reduce the total amount of work done by the program, even if it reduces the available parallelism slightly.

Being able to measure the work and critical path of a run was instrumental to our ability to tune the program. We experimented with some techniques to improve the work efficiency, and found several techniques to improve the work efficiency at the expense of increasing the critical path length. For example, on StarTech we considered a algorithm change that would value the first two children before starting the parallel tests of all the remaining children. The idea is that by valuing more children before spawning the parallel tests, it becomes more likely that the we will be able to prune some of the remaining children. When we measured the runtime on a small machine, the program ran faster, but on a big machine the runtime actually got worse. To understand why, we looked at the work and critical path length. We found that this variant of Jamboree search actually does decrease the total work, but it increases the critical path length, so that there is not enough available parallelism to keep a big machine busy. By looking at both the critical path length and the total work, we were able to extrapolate the performance on the big machine from the performance on the little machine. Consequently, we avoided introducing modifications that would hurt us in tournament conditions.

4.6 History of ⋆Socrates

We conclude this chapter by giving a brief history of the ⋆Socrates program.

We began work on this program in May of 1994. Don Dailey and Larry Kaufman of Heuristic Software provided us with a version of Socrates, their serial chess program. During May and June we parallelized the program using Cilk, focusing mainly on the search algorithm and the transposition table. During June, Dailey visited MIT to help tune the program, but we spent most of June simply getting the parallel version of the program to work correctly. In late June, we entered ⋆Socrates in the 1994 ACM International Computer Chess Championship in Cape May, New Jersey. We ran the program on the 512-node CM-5 at the National Center for Supercomputing Applications (NCSA) at the University of Illinois. Despite the fact that we had begun working on the program less than two months earlier, the program ran reliably and finished in third place.

The chess program then sat pretty much untouched for the next 9 months. In March 1995, Don Dailey again joined us to work on the chess portion of the code and we started preparing for the May 1995 tournament. The 1995 tournament was the World Computer Chess Championship held in Hong Kong. The World Computer Chess Championship is held every three years and, unlike the 1994 tournament, which fielded mostly teams from the US, this tournament attracts the best computer chess systems from around the world. We had our work cut out for us.

For this tournament we were able to get access to the 1824 node Intel Paragon [Int94] at Sandia National Labs. We began by porting Cilk to Paragon, with the help of Rolf Riesen of Sandia. The port of Cilk was fairly easy, although in the process we exposed several bugs in the Paragon's SUNMOS operating system. Parts of the original ⋆Socrates code, in particular the transposition table, made direct use of the CM-5 communication hardware. These portions of the code had to be rewritten. However, most of the code, including the abort mechanism, worked without modification, and in short order we had a working chess program.

Our first test of the program came in late March when, at the Maryland Theory Day, we played International Grandmaster Gennady Sagalchik. Grandmaster Sagalchik has a UCSF rating of 2568 and is the 35-th highest ranked player in the US; but we have 1824 Intel i860s. The game was played under standard tournament time controls, 2 hours for the first 40 moves, then 20 moves per hour. Sagalchik drew White and took the early lead in the game. ⋆Socrates played well and came back and eventually gained an advantage. On its 55th move ⋆Socrates promoted its pawn to a queen, and Sagalchik lost shortly thereafter. After this game an informal speed game was played and ⋆Socrates won again.

Although ⋆Socrates won the game, we did have a serious problem during the game. The machine crashed 4 or 5 times during the early part of the match. Fortunately, Sagalchik graciously allowed us to restart each time without losing any time on our clock, so it did not significantly hurt our performance in the game. (Although the stops probably did hurt his concentration.) At least one of the crashes was caused by bad hardware. The rest appeared to be software problems due to incorrect settings of some OS parameters, and changing some of these parameters seemed to solve the problem.

Two months later we competed in the World Computer Chess Championship. In a surprising pairing, we played IBM's Deep Blue Prototype, the heavy favorite, in the first round of this five round tournament. Deep Blue was White and took the early lead. ⋆Socrates held on for a while, but eventually succumbed to Deep Blue. At this point we figured we needed to win all four of our remaining games in order to have a chance for second place. We won the next three games fairly easily defeating the programs Dark Thought, Lchess, and Rebel.

Going into the final round there were four programs left with a chance to win. Deep Blue was the leader, with ⋆Socrates and Fritz a half point behind, and Hitech a further half point behind. Fritz was running on a standard Pentium while the other three had much more powerful hardware. (Hitech uses special-purpose chess hardware, while Deep Blue has parallel special-purpose chess hardware.) In a very surprising upset Fritz quickly defeated Deep Blue. Deep Blue's opening book ended one move to soon, and their first move out of the book was a terrible blunder. Given Deep Blue's loss, a win in our game against Hitech would leave us tied for first. As seemed to happen in many of our games, we got off to a bad start, and Hitech had the early advantage. For a while it looked pretty grim. But we were outsearching Hitech by several ply and eventually this advantage began to show as ⋆Socrates's evaluation of our position started to improve slightly on each move. Eventually we gained a big advantage and Hitech resigned.

This left us in a tie for first place. At 9pm, shortly after our Hitech game ended, we began a tiebreak game against Fritz which lasted til 3am. We drew White for this playoff game but again got off to a poor start and after the opening "Fritz had a distinct advantage."[4] For a large number of moves Fritz' advantage stayed fairly constant, with an unusually large number of each sides moves being predicted by the opponent. However, as the endgame approached, Fritz began to take advantage of its edge and was able to start pushing its pawn towards promotion. After it was apparent to both programs that the pawn could not be stopped, we resigned.

Although we lost the playoff, we did finish a respectable second. Our program ran

[4]According to D. Beal in a description of the tournament in [Bea95].

reliably throughout the tournament, with the only crashes being due to memory ECC errors. One area for improvement to ⋆Socrates that this tournament pointed out is our opening, as we fell behind early in several of our games.

The ⋆Socrates program has been an exciting program to work on, and it has met the goals we had in mind when we first decided to work on a computer chess program. ⋆Socrates has allowed us to showcase the performance, stability, expressibility, and portability of the Cilk system. In addition ⋆Socrates has helped us improve the Cilk system by pointing out several useful modifications to the system. The ⋆Socrates program also pointed out to us that programming in continuation passing style was not as easy as we had first thought. The pseudocode for the Jamboree search algorithm takes one procedure and only twenty lines. The Cilk code, however, utilizes over a dozen different threads to implement the search code, increasing to almost two dozen if the code for aborting computations is included as well. Our experience with ⋆Socrates helped bring the issue of programmability to the forefront. The next chapter, as well as Chapter 7, deals with this issue directly.

Chapter 5

Cilk-2: Programming at a Higher Level

In this brief chapter we look at some modifications to the Cilk language and runtime system intended to make Cilk programs easier to write.

The PCM and Cilk-1 systems were successful, in part, because they took the base message-passing system and allowed the user to program at a higher level. The user can write his program in terms of threads, and the system takes care of all the details and protocols needed to execute a multithreaded program on the underlying machine architecture. The Cilk-1 style of coding, where the user explicitly creates and wires together threads, offers the programmer much flexibility. But this flexibility comes at a price, namely the difficulty of writing such codes. Users must write their codes in an explicit continuation passing style, and a single sequential C procedure may need to be broken into many separate threads to express it in Cilk-1. Although some find this a natural way to program, many find it confusing. Even when the code is straightforward, it is still often tedious to write and read such codes.

This chapter describes Cilk-2 which extends Cilk-1 by adding simple language extensions that allow users to write codes at a higher level. To make writing Cilk programs easier, we wanted to relieve the programmer of the task of writing a program in continuation-passing style. Just as PCM raises the level of programming and hides the details of the lower-level message-passing system, we wanted to raise the level further and hide the details of the thread-based runtime system. Of course, we also wanted to do this without destroying the performance guarantees that Cilk provides. We attempted to choose a set of higher-level primitives that would allow a wide range of programs to be expressed. We appear to have been successful, as these language extensions have been sufficient for us to rewrite most of our Cilk-1 applications in the Cilk-2 style. The only exception is the chess program which, due to its speculative nature, cannot be expressed in Cilk-2 style without resorting to the lower-level Cilk-1 mechanisms. The Cilk-2 language includes all the Cilk-1 mechanisms, so chess, as well as any other Cilk-1 code, can still be expressed in the Cilk-2 system. However, when we talk about a "Cilk-2 style" code, we mean those codes written using only the new, higher-level, primitives. So we say that chess cannot be written in Cilk-2 style, even though it can be written in the Cilk-2 system. In Chapter 7 we examine further enhancements that allow speculative applications, such as chess, to be written at a higher level.

Although we wanted to raise the level of programming so that users need not have to deal with threads, we still wanted to have an "explicitly parallel" language where the user must explicitly specify what can be done in parallel. We did not want to design an "implicitly parallel" language where the system or compiler would try to deduce what can be run in parallel. The Cilk system may be a good target language for a parallelizing compiler, but doing a good job at building such a compiler is a much more difficult task than we were planning to undertake. Another part of the reason we did not want an implicitly parallel language was because we believe that to write an efficient parallel program the user must think about parallel algorithms. Having to explicitly specifying what can be done in parallel is one way to force a user think this way. Even when using an implicitly parallel system, a programmer must still have a good understanding of which portions of his code will run in parallel. If we had tried to build an implicitly parallel system, and did only a halfway decent job, then it might often be unclear to the programmer what would be executed in parallel. This would result in a system much harder to program than an explicitly parallel system.

In this chapter we first look at a simple algorithm expressed in Cilk-1 in order to highlight some of the difficulties of programming in the Cilk-1 language. Then, we describe the Cilk-2 language and its implementation. We then revisit the example algorithm and see how Cilk-2 makes the example algorithm much easier to express.

The Cilk-2 system described in this chapter represents joint work with other members of the Cilk team: Robert Blumofe, Matteo Frigo, Bradley Kuszmaul, Charles Leiserson, Rob Miller, Keith Randall, and Yuli Zhou. I was involved in making the design decisions described in this chapter, but much of the implementation was done by others: Rob Miller implemented the Cilk-to-C preprocessor, and most of the runtime system modifications were made by Matteo Frigo and Keith Randall.

5.1 A Cilk-1 Example: Knary

To highlight some of the difficulties in programming in Cilk-1, consider the knary program that was introduced in Chapter 3. Knary was introduced as a synthetic benchmark whose parameters could be set to produce a variety of values for work and critical path. **knary(n,k,r)** generates a tree of branching factor k and depth n in which the first r children at every level are executed sequentially and the remainder are executed in parallel. At each node of the tree, the program runs an empty "for" loop for 400 iterations, simulating the work that would be done in an actual program. The knary program counts the number of leaves of this tree, so the program is in effect a complicated way to compute the value of k^{n-1}. There are faster ways to compute k^{n-1}, of course, but we are interested in this program because of its control structure. At a high level, the control structure of this program is reminiscent of the chess program, where, when searching a position, we first do some recursive searches one at a time, and then spawn off a bunch of searches in parallel.

The description of this program is quite simple, and we would hope that its implementation would be also. This program could be written in many ways, but we have chosen to present the code as it was originally written for the performance tests shown in Table 3.1 of Chapter 3, and first presented in [BJK+95]. As such, this code does not necessarily represent the "best" way of implementing knary, but shows how an experienced Cilk programmer quickly coded up this application.

Figure 5-1, which is split across two pages, shows the knary code. Four different threads were used in this version of knary. The first is the **knary** thread which performs some imitation work and then spawns off **knary_serial**, which performs **NumSerial** serial sub-calls, and **knary_parallel** which performs **NumParallel** parallel sub-calls. Notice that, as is common in continuation-passing style code, **knary** performs its two spawns in the reverse of the order in which the threads will execute. The routine **knary_parallel**, which is the second to be executed, must be spawned first, so that when **knary_serial** is spawned, it can be told where to pass its results. The thread **knary_serial** calls itself repeatedly in order to spawn off the next serialized call and to sum the results from the previous call. The thread **knary_parallel** enters a loop to spawn off the parallel calls of **knary**, and chains the results of these calls through **sum** in order to sum the results.

It is not important for the reader to understand all the details of this implementation. What is important to notice is that although the description of knary is quite simple, the implementation is clearly not. We should further point out that this implementation cheats slightly in that it takes advantage of the fact that knary is called as a stand alone program and not as part of a larger program. This implementation sets the value of **NumSerial** and **NumParallel** in the main routine (not shown), which executes on all processors when the application begins. The program should really be written so that these values are passed as arguments to the knary thread. Although passing **NumSerial** and **NumParallel** could easily be done, these values would also need to be passed through the **knary_serial** and **knary_parallel** threads, further increasing the argument count of these threads and making the code even harder to read.

Part of the reason that Cilk-1 code can appear so convoluted is that threads are not easily composable. Consider what happens to the knary code if we want to perform real work at the beginning of each **knary** thread, instead of just having imitation work. Assume there is some function **work()** that performs some work and returns a value indicating whether this call of knary should return immediately or should continue as usual. Again, this control is similar to part of the chess search algorithm. If **work** is a standard C procedure, this change is easy to make. We just replace the empty loop in **knary** with a call to **work** followed by a test to see if we should perform a **send_argument** and return. If **work** is not a C procedure, but is a Cilk thread, then this change is more complex. The **knary** thread must now be broken into two threads, call them **knary-1** and **knary-2**. The **knary-1** thread first does a **spawn_next** of **knary-2** and then spawns the **work** thread. The **knary-2** thread receives the result of the **work** thread, and depending on this result, either returns a value or continues as the original **knary** thread does.

```
/* knary(n, k, r)
 * This code is a recursive program similar to fib that spawns a tree
 * of depth 'n and branching factor 'k' and counts all the leaves.
 *   Thus computing (k)^(depth-1) .
 *
 * The first 'r' of the sub-trees are spawned serially.
 * The rest of the sub-trees are spawned in parallel.
 */


/* These two values are set on all processors by the main() routine (not
 * shown) according to the program inputs.
 */
int NumSerial;                      /* set to r */
int NumParallel;                    /* set to n-r */

/*** Computes knary of depth 'depth' ***/
thread knary(cont k, int depth){
  cont serial_result;

  /* Do some work in each thread. */
  int i;
  dummy = (int)&i;
  for(i=0;i<400;i++){}

  if (depth<2) {
    SendWordArgument (k, 1);
  }
  else{
    spawn_next knary_parallel(k,depth,?serial_result, NumParallel);
    spawn_next knary_serial(serial_result,depth,0,0,NumSerial);
  }
}


thread sum (cont k, int x, int y)
{
  SendWordArgument (k, x+y);
}

  /* ... continued on next page ... */
```

Figure 5-1: Cilk-1 code for knary.

```
/*  Calls knary 'num_serial_left' times.
 *  Each subcall is spawned serially.
 *  Returns the sum of all the calls.
 */
thread knary_serial (cont k, int depth, int result1, int result2, int num_serial_left)
{
  int result = result1 + result2;
 if (num_serial_left==0){
    SendWordArgument(k,result);
  }
  else{
    cont child_result;
    spawn_next knary_serial(k, depth, ?child_result, result, num_serial_left-1);
    spawn knary(child_result,depth-1);
  }
}

/* Calls knary 'num_left' times.
 * All calls are spawned in parallel.
 * Returns the sum of all the calls.
 *
 * Rather than spawning off num_left sums, we could use a single thread
 * to sum all the results.  Instead this method was chosen because
 * it performs the same number of spawns as knary_serial.
 */
thread knary_parallel (cont k,int depth, int result_from_ser, int num_left)
{
  int i;
  cont c_left,c_right;
  if(num_left==0){
    SendWordArgument(k,result_from_ser);
  }
  else{
    for (i=0;i<num_left-1;i++){
      spawn_next sum(k, ?c_left, ?c_right);
      spawn knary(c_left,depth-1);
      k = c_right;
    }
    spawn_next sum(k, ?c_left, result_from_ser);
    spawn knary(c_left,depth-1);
  }
}
```

Figure 5-1 continued: Cilk-1 code for knary

5.2 The Cilk 2 System

The goal when designing Cilk-2 was to allow the user to program in the traditional call-return style while still using the same efficient runtime system of threads wired together in a continuation-passing style. The natural way to implement such a system is to specify a set of higher-level parallel constructs which a preprocessor can translate into Cilk-1 style threaded code. These higher-level constructs need to be powerful enough to express most existing programs without resorting to writing threaded code. At the same time these constructs need to have a relatively straightforward transformation into threaded code.

In order to perform such a translation, we needed a more sophisticated preprocessor than the simple macro preprocessor originally used with Cilk-1. What we implemented was a new type-checking preprocessor for Cilk. This preprocessor was implemented by Rob Miller and is described in more detail in [Mil95]. This preprocessor is based on C-to-C, a tool which parses a C program and turns it into an abstract syntax tree (AST). C-to-C then performs type checking and dataflow analysis on this AST, and then uses the AST to regenerate a C program. C-to-C was extended to create Cilk-to-C. Cilk-to-C parses a program written in Cilk and creates an extended "Cilk AST" which describes the program. A Cilk AST can include constructs not allowed in a C AST. Cilk-to-C then performs type checking and other analysis on this Cilk AST, annotating the nodes of this tree with extra information. Cilk-to-C then transforms the Cilk AST into a pure C AST, removing any Cilk-specific constructs by replacing them with calls to Cilk runtime system primitives. After this phase the AST can be used to generate a C program.

An alternative to building a preprocessor was to implement a full blown Cilk compiler. We chose the former option because building our own compiler would have been a much larger task, and would have resulted in a system that was less portable. The Cilk-to-C preprocessor allows us to do much of what we would want a compiler to do, without the drawbacks of building a compiler.

The first version of Cilk-to-C was written for the Cilk-1 language, and it replaced the original macro preprocessor. This new type-checking preprocessor benefits the Cilk-1 programmer in two ways. First, Cilk-to-C detects errors, particularly type errors, that the original preprocessor cannot detect. With the original preprocessor, some of these errors are detected by the C compiler, so the error messages refer to the processed code, instead of the user's source code. Since the user may be unfamiliar with the processed code, the error may be difficult for the user to understand. Other type errors were not detected at all by the original Cilk-1 system, and would lead to an incorrect program. Since Cilk-to-C performs type checking, not only is it able to detect these errors, but it can point to the location in the original source code where the errors occurred, thus making the errors much easier to fix. The other benefit of a type-checking preprocessor for Cilk-1 is that it makes the language somewhat simpler. The runtime system has several primitives (**SendWordArgument()**, **SendCharArgument()**, etc.) for sending arguments to closures. Previously the user had to choose one of several such functions depending on the type of argument that was being sent. The Cilk-1 language now has only one such function, namely **send_argument()**, which is used for sending arguments of any type to other closures. The type-checking preprocessor knows the type of the argument being sent and automatically generates a call to the correct runtime primitive.

Once we had a preprocessor powerful enough to deal with higher-level constructs, we had to decide just what higher-level constructs we wanted. While writing applications, we noticed common paradigms that appeared in many Cilk codes. The most common paradigm was spawning off several child threads and creating a successor thread to receive the values computed by these children. It was this paradigm that we chose to support in Cilk-2. An example of the Cilk-2 language can be seen in Figure 5-2, which shows the code for fib, which recursively computes the nth Fibonacci number. The Cilk-2 code for fib is very similar to the C code for fib, containing only a few additional constructs. The first addition is the keyword `cilk` before the procedure definition. This keyword indicates that the following procedure is a Cilk procedure, not a standard C procedure. The next addition is that the two subcalls to the fib procedure are preceded by the keyword **spawn** which indicates that these procedures can be executed in parallel. The final addition is the **sync** statement. The **sync** statement indicates that the procedure is to be suspended at the **sync** point until all spawned children have completed. Consequently, the code following the **sync** can safely use the variables x and y, which receive their values from the spawned children.

We considered several choices for how to express synchronization in Cilk-2 before deciding on this one. The other options we considered incorporated mechanisms which gave the user more control over the synchronization process. In particular we considered a mechanism essentially the same as join variables in Cid[Nik94]. In this proposal each spawn would

```
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);
   }
}
```

Figure 5-2: A Cilk-2 procedure to compute the nth Fibonacci number.

have a *join counter* associated with it. At a sync point the user would need to specify a join counter to wait on. Waiting on a join counter would indicate that the procedure should suspend until all children spawned with that join counter have completed. Another proposal required the user to explicitly specify at each sync point precisely which variables he wanted to wait for.

All these other proposals had the advantage that they gave the user more control over synchronization, but in the end we decided we did not want to give the user that control. In most cases this extra control is unneeded and just makes the code messier. With the exception of chess, all the Cilk programs we have written can be expressed in Cilk-2 style. If we had chosen one of these other proposals, all of our programs would have had to deal with the extra details exposed by that proposal, but only the chess program would have benefited from it. (And even these proposals were not sufficient to completely remove the need for the chess program to resort to the lower-level Cilk-1 mechanisms.)

Also, with the mechanism we chose, all programs written in Cilk-2 meet the criteria for Cilk's performance guarantees as described in Chapter 3. Some of the other proposed mechanisms did not. The last advantage of the chosen mechanisms is that since they give the programmer less control, they are simpler to implement. Simplicity of implementation is not why we chose this path, but it is a nice feature anyway.

Before concluding this section, we give a brief overview of the implementation of Cilk-2. We focus on what constructs Cilk-2 code is translated into. See [Mil95] for details on how this transformation is performed.

The idea for transforming Cilk-2 style code into threaded code is fairly straightforward. Since the runtime system deals with threads, not procedures, a Cilk-2 procedure must be broken up into a set of threads. The sync points form the boundaries for these implicitly defined threads. In effect we treat a `sync` like a `spawn_next`, where the thread being spawned is the rest of the procedure after the `sync`.

One could try to translate Cilk-2 code directly into Cilk-1 style code. Using this translation, the fib code in Figure 5-2 would then be transformed into code almost identical to the fib code shown in the Cilk-1 chapter. Although this translation would work fine for fib, in general it would be more difficult. In Cilk-1 code, when spawning a child, that child must be passed a continuation which points to the "rest of the procedure", therefore the `spawn_next` of the rest of the procedure must occur before any children are spawned. When translating Cilk-2 code, having to perform the `spawn_next` first is troublesome because a `sync` may be nested within a conditional or a loop. Therefore when executing a procedure, until we actually reach a `sync`, we may not know which `sync` we will reach. When a child is spawned, not knowing which `sync` will be reached makes it difficult to create the continuation that must be passed to that child.

To solve the problem of not being able to create a continuation to pass to a child, Cilk-2 is translated such that all the threads that form a procedure share the same closure. This closure is sometimes called a *frame*. When a thread is spawned, the current frame is used as the frame to which the thread should send its result. When a `sync` is reached, all the variables that are needed after the `sync` are stored into the frame. When the next thread of the procedure is executed, the first part of that thread has code to read the needed variables back out of the frame. Using only one frame has efficiency advantages as well. Only one frame needs to be allocated and initialized per procedure, rather than one frame for every thread of a procedure. Also, there may be variables which are written into the frame once, and then used by several threads of that procedure. If separate closures were used for every thread, then these variables would have to be copied from closure to closure.

5.3 Knary Revisited

Let us now go back to the knary example and see how it can be expressed in Cilk-2. Figure 5-3 shows the knary code as written in Cilk-2. Instead of the four threads used by

```
/* knary(n, k, r) */
cilk int knary(int depth, int num_serial, int num_parallel){

  /* Do some work in each thread. */
  int i;
  int dummy = (int)&i;
  for(i=0;i<ExtraWork;i++){}

  if (depth<2) {
    return 1;
  }
  else{
    int results[MAX_PARALLEL];
    int answer = 0;

    /* Perform the serial recursive calls, summing in the result */
    for (i=0;i<num_serial;i++){
      child_result= spawn knary(depth-1);
      sync;
      answer += child_result;
    }

    /* Spawn off the parallel calls */
    for (i=0;i<num_parallel;i++){
      results[i] = spawn knary(depth-1);
    }
    sync;

    /* sum the results */
    for (i=0;i<num_parallel;i++){
      answer += results[i];
    }
    return answer;
  }
}
```

Figure 5-3: Cilk-2 code for knary.

the Cilk-1 code, the Cilk-2 code requires just one procedure. This procedure contains a loop to spawn off the first `num_serial` sequential children, followed by a loop to spawn of the `num_parallel` parallel children. The main difference between these two loops is that the loop for the sequential children has a `sync` statement inside the loop while the loop for the parallel children has a `sync` statement only after the loop.

Composing Cilk procedures in Cilk-2 is much easier than composing threads in Cilk-1. Earlier, we gave the example of making the following modification to knary. Instead of performing make-work, knary was supposed to call a function, `work()`, that performs some work and may return a value, STOP, indicating that this call of knary should return immediately instead of continuing as usual. In Cilk-1 this change is easy to implement if `work()` is a C function, but it requires more effort if `work()` is written as a Cilk thread. In Cilk-2, this code is easy to write even if `work()` is a Cilk procedure. We just add the following code to the beginning of knary:

```
i = spawn work();
sync;
if (i==STOP) return(0);
```

This code is almost the same as what would be required if `work()` were a C procedure. The only difference is the addition of `spawn` and `sync`.

5.4 Conclusion

The Cilk-2 language allows the programmer to write his application at a higher level using the familiar call/return semantics. Our type-checking preprocessor then automatically breaks up the user's code into a multithreaded program which can be executed with our work-stealing scheduler, thereby providing the user with the time, space, and communication performance guarantees described in Chapter 3. Although the Cilk-2 language is more restrictive than Cilk-1, we were able to rewrite almost all existing Cilk-1 programs using the new, simpler, Cilk-2 syntax.

The only application that cannot be expressed using the Cilk-2 syntax was ⋆Socrates. We cannot express ⋆Socrates because it performs speculative computations, and because some of these speculative computations are aborted. For most algorithms, the order in which the user's threads are executed does not effect the amount of work performed by the program. But for speculative algorithms, the amount of work the program performs can depend greatly on the order in which threads are executed. Cilk-1 contains features such as priority threads, which are instrumental in implementing speculative searches efficiently. These features are not reflected in the higher-level Cilk-2 constructs, so the algorithms that used theses features cannot be efficiently written using only the higher-level Cilk-2 constructs. Another reason ⋆Socrates cannot be written entirely with the higher-level

Cilk-2 constructs is the ⋆Socrates sometimes kills off previously spawned threads. The ⋆Socrates code is able to abort existing threads only because Cilk-1 contains sufficient low-level primitives, such as migration and nonstealable threads, to allow the user's code to keep track of all the spawned threads. Again, these low-level features are not available via the higher-level Cilk-2 constructs. In Chapter 7 we will describe some further high-level extensions to Cilk which will allow the chess program to be written without resorting to the lower level Cilk-1 primitives.

Chapter 6

Cilk-3: Shared Memory for Cilk

One of the biggest shortcomings of the Cilk systems described so far is that it is difficult to write applications where there are significant amounts of data shared throughout the computation. The improvements to the Cilk system described in Chapter 5 make it much easier to express algorithms in Cilk, but they do nothing to increase the range of applications that can be expressed. In Cilk all data that is shared by several threads must be explicitly passed among those threads. Consequently, writing efficient programs that deal with large structures is difficult. The Cilk applications we have described so far mostly consist of tree-like algorithms where very little data needs to be shared between different subtrees. Those applications that do need to share data throughout the computation do so only by going outside of Cilk. For example, the global transposition table in the $\star$Socrates program was implemented directly with active messages.

The shared-memory system described in this chapter is an attempt to alleviate this problem. With this system we can implement data-intensive applications such as matrix multiply, LU-decomposition, and Barnes-Hut. Rather than attempting to build a shared memory system that can solve all problems, we focused on building one that would be sufficient for the types of problems that are naturally expressed in a multithreaded programming environment such as Cilk. Instead of using one of the consistency models derived from sequential consistency, we use our own, relaxed, consistency model. Our model, which we call "dag consistency," is a lock-free consistency model which, rather than forcing a total order on global memory operations, instead ensures only that the constraints specified by

Some of the work described in this chapter will be reported on in a paper by Robert Blumofe, Matteo Frigo, Charles Leiserson, Keith Randall and myself [BFJ$^+$96].

the computation dag are enforced. Because dag consistency is a relaxed consistency model, we have been able to implement coherence efficiently in software for Cilk. We dubbed the Cilk-2 system with dag-consistent shared memory "Cilk-3."

This Cilk shared memory system was designed by members of the Cilk team: Robert Blumofe, Matteo Frigo, Charles Leiserson, Keith Randall and myself. Matteo Frigo, Keith Randall and I implemented the CM-5 version of the system. Robert Blumofe is implementing a version of this system for networks of workstations. Charles Leiserson and I devised the correctness proof given in Section 6.3.

6.1 Introduction

Architects of shared memory for parallel computers have traditionally attempted to support Lamport's model of sequential consistency [Lam79] which states:

> A system is sequentially consistent if 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 of each individual processor appear in this sequence in the order specified by its program.

Unfortunately, designers have found Lamport's model difficult to implement efficiently, and hence relaxed models of shared-memory consistency have been developed [DSB86, GS93, GLL$^+$90] 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 performed by physical processors. In this chapter, we introduce "dag consistency", a lock-free, processor-independent consistency model which can be implemented efficiently for multithreaded programming environments such as Cilk.

To illustrate the concepts behind dag consistency, consider the problem of parallel matrix multiplication. One way to program matrix multiplication is to use the recursive divide-and-conquer algorithm shown in Figure 6-1. 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.

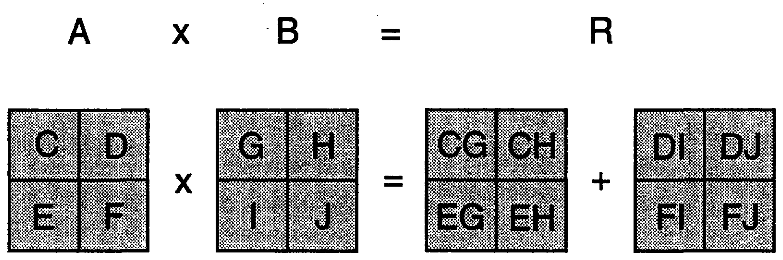

Figure 6-1: Recursive decomposition of matrix multiplication. The multiplication of $n \times n$ matrices requires eight multiplications of $n/2 \times n/2$ matrices, followed by one addition of $n \times n$ matrices.

Figure 6-2 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*. All three matrices are stored in shared memory. The Cilk procedure **`matrixmul`** takes pointers to the first block in each matrix as input, 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 quickly computed. As **`matrixmul`** executes, values are stored into `R`, as well as into a temporary matrix **`tmp`**.

The procedure **`matrixmul`** 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 page-aligned 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, storing the results in the 8 disjoint submatrices of `R` and **`tmp`**. The recursion is made to execute in parallel by using the **`spawn`** directive, which is similar to a C function call except that the caller can continue to execute even if the callee has not yet returned. The **`sync`** statement in line 21 causes the procedure to suspend until all the spawned procedures 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 **`matrixmul`** returns.

```
cilk void matrixmul(long nb, shared block *A, shared block *B,
                             shared block *R)
 {
  if (nb == 1)
    multiply_block(A, B, R);
  else {
    shared block *C,*D,*E,*F,*G,*H,*I,*J;
    shared block *CG,*CH,*EG,*EH,*DI,*DJ,*FI,*FJ;
    shared page_aligned block tmp[nb*nb];

    /* get pointers to parts of original matricies */
    partition_matrix(nb, A, &C, &D, &E, &F);
    partition_matrix(nb, B, &G, &H, &I, &J);

    /* get pointers to places to put results */
    partition_matrix(nb, R, &CG, &CH, &EG, &EH);
    partition_matrix(nb, tmp, &DI, &DJ, &FI, &FJ);

    /* do multiplication subproblems */
    spawn matrixmul(nb/2, C, G, CG);
    spawn matrixmul(nb/2, C, H, CH);
    spawn matrixmul(nb/2, E, H, EH);
    spawn matrixmul(nb/2, E, G, EG);

    spawn matrixmul(nb/2, D, I, DI);
    spawn matrixmul(nb/2, D, J, DJ);
    spawn matrixmul(nb/2, F, J, FJ);
    spawn matrixmul(nb/2, F, I, FI);
    sync;

    /* add results together into R */
    spawn matrixadd(nb, tmp, R);
    sync;
  }
 return;
 }
```

Figure 6-2: Cilk code for recursive blocked matrix multiplication. The call `multiply_block(A,B,R)` performs a serial multiplication of blocks `A` and `B` and places the result in block `R`. The side length of each of these three matrices is `nb` blocks. The procedure `matrixadd` is implemented by a straightforward parallel divide-and-conquer algorithm.

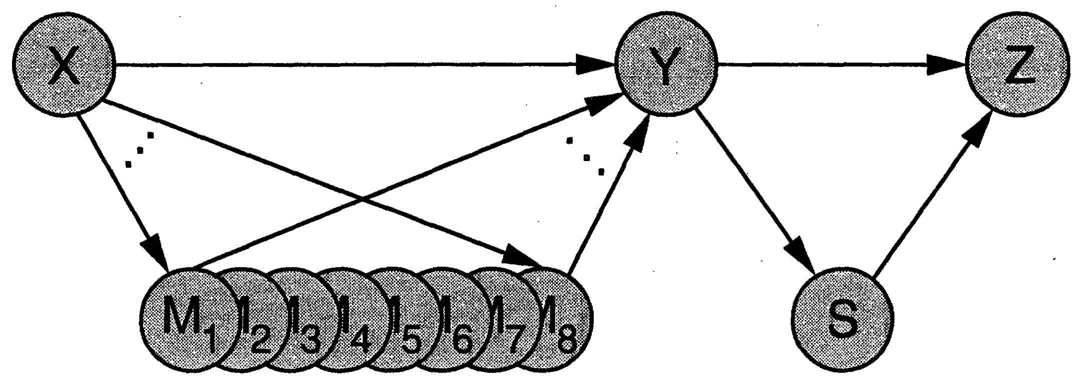

Figure 6-3: Dag of blocked matrix multiplication. Each circle represents a thread of the computation. Threads are linked by downward spawn edges, horizontal continue edges, and upward return edges. Some edges were omitted for clarity.

Like any Cilk multithreaded computation, the parallel instruction stream of `matrixmul` can be viewed as a directed acyclic graph (dag) of threads organized into a tree of procedures. Figure 6-3 illustrates the structure of the dag. Each vertex corresponds to a *thread* of the computation, which in Cilk is a nonblocking sequence of instructions. A procedure is a sequence of threads that share the same *frame* or *activation record*. For example, the syncs in lines 21 and 23 break the procedure `matrixmul` into three threads X, Y, and Z, which correspond respectively to the partitioning and spawning of subproblems $M_1, M_2, \ldots, M_8$ in lines 2–20, the spawning of the addition S in line 22, and the return in line 25. The dag of a Cilk computation contains three kinds of edges. A *spawn* edge connects a thread with its spawned child. A *continue* edge connects a thread with its successor in the same procedure. A *return* edge reflects the synchronization that occurs when a child completes and notifies the thread in its parent procedure that is waiting for its return. Thus, a Cilk computation unfolds as a *spawn tree* composed of procedures and the spawn edges that connect them to their children, but the execution is constrained to follow the precedence relation determined by the dag of threads.

What kind of memory consistency is necessary to support a shared-memory program such as `matrixmul`? 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 basic idea 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 `matrixmul`, 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. Dag consistency is similar to location consistency [GS93], but it is defined in terms of the dag of a user's multithreaded computation rather than in terms of processors and synchronization points. We shall present a formal model of dag consistency in Section 6.2.

The current Cilk mechanisms to support dag-consistent distributed shared memory on the Connection Machine CM-5 are implemented in software. Nevertheless, codes such as `matrixmul` run with good efficiency, as we shall shortly document. Like many software distributed shared memory implementations, the Cilk implementation is page based to amortize the cost of remote reads and writes over many references. Whenever a thread accesses a page that is not resident in the local page cache of a processor, a page fault occurs, and a protocol ensues that brings the required page into the page cache. Although memory locations are grouped into pages, Cilk maintains dag consistency at the granularity of individual 32-bit words.

Cilk also supports stack allocation of distributed shared memory. The declaration of `tmp` in line 8 of `matrixmul` causes the shared-memory stack pointer to be incremented by `nb*nb` blocks. When running in parallel, however, a simple serial stack is insufficient. Instead, Cilk provides a distributed "cactus stack" [HD68, Mos70, Ste88] that mimics a serial stack in such a way that during execution, every thread can access all the variables allocated by its parents and can address a variable directly by its depth in the stack. Despite the fact that the stack is distributed, allocation can be performed locally with no interprocessor communication. Cactus stack memory is deallocated automatically by the Cilk runtime system when a spawned procedure returns.

Before discussing how maintenance of dag consistency affects the performance of Cilk programs, let us first review the performance of Cilk programs that do not use shared memory. Any multithreaded program can be measured in terms of the "work" and "critical path length" of its computation dag. The *work*, denoted T_1, is the time used by a one-processor execution of the program, which corresponds to the sum of the execution times of all the threads. The *critical path length*, denoted T_∞, is the total amount of time required by an infinite-processor execution, which corresponds to the largest sum of thread execution times along any path. With P processors, the execution time cannot be less than T_1/P or less than T_∞, and Cilk's "work-stealing" scheduler provably achieves $O(T_1/P + T_\infty)$ time with high probability on fully strict multithreaded computations. For example, the work to multiply two $n \times n$ matrices using `matrixmul` is $\Theta(n^3)$ and the critical path of this algorithm is $\Theta(\lg^2 n)$. If no shared-memory page faults were taken, therefore, the entire algorithm would run in $\Theta(n^3/P + \lg^2 n)$ time with high probability.

In order to model performance accurately, however, we must account for the effects of shared-memory page faults. We show that if F_1 is the number of page faults for a multithreaded program running on 1 processor with a page cache of size C, then the total number F_P of page faults taken by P processors, each with a page cache of size C, is $F_P \leq F_1 + 2Cs$, where s is the number of steals during the execution.

A graph of the performance of `matrixmul` on the Connection Machine CM-5 is shown in Figure 6-4. Three curves are shown, the lower curve is the performance of the `matrixmul` code shown in Figure 6-2 on a 1024×1024 multiply, and the upper two curves are the performance of a variant that uses no temporary storage and has a longer critical path. The dag-consistent shared-memory code performs at 5 megaflops per processor as long as the work per processor is large. This performance compares reasonably well with the other matrix multiplication codes on the CM-5. For example, an implementation coded in Split-C [CDG+93] 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 current implementation can be reduced further, but that in any case, it is a reasonable price to pay for ease of programming and dynamic scheduling.

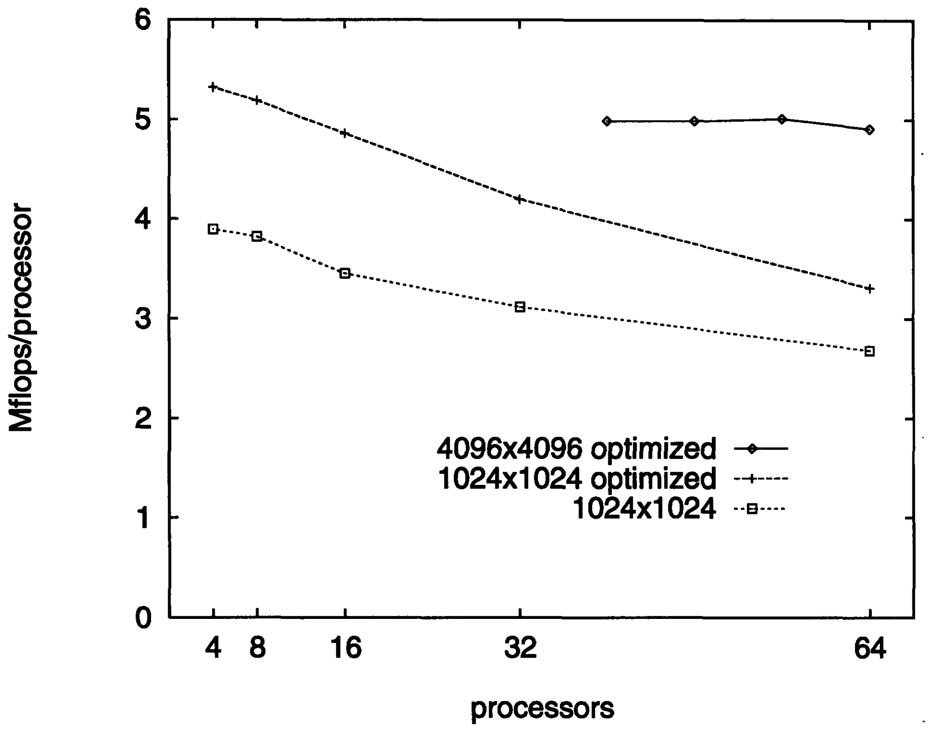

Figure 6-4: Megaflops per processor versus the number of processors for several matrix multiply runs on the Connection Machine CM-5. The shared-memory cache on each processor is set to 2MB. The lower curve is for the `matrixmul` code in Figure 6-2 and the upper two curves are for an optimized version that uses no temporary storage.

We have implemented irregular applications that employ Cilk's dag-consistent shared memory, including a port of a Barnes-Hut N-body simulation [BH86] and an implementation of Strassen's algorithm [Str69] for matrix multiplication. These irregular applications provide a good test of Cilk's ability to schedule computations dynamically. We achieve a speedup of 9 on an 8192-particle N-body simulation using 32 processors, which is competitive with other software implementations of distributed shared memory [JKW95]. Strassen's algorithm runs as fast as `matrixmul` for 2048×2048 matrices, and we coded it in Cilk in a few hours.

The remainder of this chapter is organized as follows. Section 6.2 gives a formal definition of dag consistency, and Section 6.3 describes an abstract algorithm for maintaining dag consistency and then gives a proof of its correctness. Section 6.4 describes an implementation of this algorithm for Cilk on the Connection Machine CM-5, and also describes our cactus-stack memory allocator. Next, Section 6.5 analyzes the number of faults taken by multithreaded programs, both theoretically and empirically. Section 6.6 compares dag-consistency with some related consistency models and offers some ideas for the future.

6.2 Dag Consistency

In this section, we formally define dag consistency in terms of the dag that represents a multithreaded computation. We give conditions under which dag-consistent multithreaded programs are deterministic, and we discuss how nondeterminism can arise. Finally, we investigate 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.

We first introduce some terminology. Let $G = (V, E)$ be the dag of a multithreaded computation. For $i, j \in V$, if a path of nonzero length from thread i to thread j exists in G, we say that i (strictly) *precedes* j, which we write $i \prec j$. For any thread $i \in V$, the set $\{j \in V : j \prec i\}$ is the set of *predecessors* of i, and the set $\{j \in V : i \prec j\}$ is the set of *successors* of i. We say that two threads $i, j \in V$ with $i \neq j$ are *incomparable* if we have $i \not\prec j$ and $j \not\prec i$.

Shared memory consists of a set M of *objects* containing a *value* field that threads can read and write. To track which thread is responsible for an object's value, we imagine that each value has a tag which the write operation sets to the name of the thread performing the write. When a thread performs a read on an object, it receives some value, but the particular value it receives depends upon the consistency model. We assume without loss of generality that each thread performs at most one read or write. We also make the simplifying assumption that all objects contain some initial value tagged with a "fictitious" thread that precedes all other threads. This assumption saves us the trouble of specifying what happens if a thread reads an object not written by any "real" predecessor.

Informally, we want to define dag consistency such that a read can "see" a write only if there is some serial execution order in which the read sees that write. Unlike sequential consistency, however, dag consistency allows different reads to return values that are based on different serial orders, as long as the values returned are consistent with the precedence relations given by the dag.

In addition to stating what values might be seen by a read, the following formal definition of dag consistency focuses on what values a read cannot see.

Definition 8 *The shared memory M of a multithreaded computation $G = (V, E)$ is* **dag consistent** *if the following conditions hold:*

1. *If any thread $i \in V$ reads any object $m \in M$, it receives a value v tagged with some thread $j \in V$ such that j writes v to m and we have $i \not\prec j$.*

2. *For any three threads $i, j, k \in V$, such that $i \prec j \prec k$ holds, if j writes some object $m \in M$ and k reads m, then the value received by k is not tagged with i.*

The first part of this definition says that if a thread i reads an object, it receives the value written to that object by some thread j, where j must not be a successor of i. Since most computer systems are not prescient, this part of the definition is easy to implement. The second part of the definition is more subtle, because it says what the value cannot be, rather than what the value can be. It ensures that a writer masks writes by any of its predecessors from all of its successors. This property is implicit in ordinary serial execution: whenever a write to an object occurs, previous values of the object are thenceforth forever hidden.

The definition of dag consistency allows nondeterminism, which we view as a generally undesirable property of a parallel program, but it is relatively easy to write programs that are guaranteed to be deterministic. Nondeterminism arises when a write to an object occurs that is incomparable with another read or write to the same object. For example, if a read and a write to the same object are incomparable, then the read may or may not receive the value of the write. Similarly, if two writes are incomparable and a read exists that succeeds them both with no other intervening writes, the read may receive the value of either write. To avoid nondeterminism, we require that no write to an object occurs that is incomparable with another read or write to the same object. If no two writes to the same object are incomparable, then 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 the definition of dag consistency, every read of an object sees exactly one write to that object. Since a write of one object has no bearing on a read of a different object, the execution is deterministic.

Nondeterminism is not the only problem that can arise in dag-consistent programs. As with most consistency models, dag consistency can suffer from atomicity anomalies when the concrete objects supported by the shared-memory system are larger than the abstract objects that the programmer is reading and writing. For example, suppose that system objects are 4 bytes long, but the programmer is treating the system object as 4 1-byte abstract objects. Two incomparable threads may each perform an update on a different abstract object, expecting both writes to be visible to a common successor. But, if these 1-byte values are packed into the same 4-byte concrete object, then these writes are really incomparable writes to the same 4-byte object. Consequently, one of the writes may nondeterministically mask the other, and the update to one of the bytes may be lost. Fortunately, this problem can easily be avoided by not packing together abstract objects that might be updated by incomparable threads.

Atomicity anomalies can also occur when the programmer's abstract object is larger than the system's concrete object. For example, suppose the system supports 4-byte concrete objects, but the programmer needs an 8-byte 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, which we advocate as good parallel programming practice, the programmer need not worry about this atomicity problem.

6.3 Maintaining Dag Consistency

In this section we show how dag consistency can be maintained during the execution of a multithreaded computation. We focus on the class of "well-structured" computations that we showed in Section 3.6 can be scheduled efficiently. We give an algorithm that maintains dag-consistent shared memory for well-structured computations executing in a distributed environment, and we prove that it is correct. Section 6.4 describes our implementation of the algorithm.

Our dag-consistency algorithm depends on properties of multithreaded dags. Recall that a procedure consists of a sequence of threads connected by continue edges. Spawn edges go from a thread in one procedure to the first thread in a child procedure, thereby structuring the procedures into a spawn tree. Alternatively, we can view the spawn and continue edges as structuring the threads into a ***spawn-continue tree.*** If the computation is ***fully strict***, every data-dependency edge entering a procedure comes from the final thread of a child procedure. Fully strict computations can be scheduled efficiently, because they are well structured. We shall exploit this property to design an efficient dag-consistency algorithm.

We shall present the dag-consistency algorithm using the nomenclature of Cilk's work-stealing scheduler in which an idle processor obtains work by "stealing" a thread from a busy processor. After stealing a thread, a processor executes the thread, which may cause other threads to be created. The processor executes the created threads in depth-first order, mimicking ordinary, serial, depth-first execution. If another processor requests work from the processor, the thread that is closest to the root and ready to execute is stolen away. We call the subtree of the spawn-continue tree that is rooted at a stolen thread a ***subcomputation.*** We shall be particularly interested in the ***kernel of a subcomputation***, which we define to be the (stolen) root thread of the subcomputation together with all threads reachable by spawn and continue edges without passing through another stolen thread. Thus, as shown in Figure 6-5, the scheduling algorithm partitions the spawn-continue tree into a tree of kernels, each of which consists of all threads that execute on the same processor from the time that a subcomputation's root thread is stolen to the time that the processor goes idle.

In order to make our dag-consistency algorithm simple, we rely on a property of Cilk's runtime system. During the execution of a fully strict computation, a thread can be stolen only if it belongs to a procedure whose previously spawned children have all completed. Since a procedure with outstanding children cannot be moved, Cilk's bookkeeping of the relationship between the procedure and its children is straightforward, because a child's parent never moves. The dag-consistency algorithm also exploits this property, but for a different reason. Specifically, it relies on the fact that a data-dependency edge leaving a thread always goes to another thread in the same subcomputation kernel or to a thread in the parent kernel. If it were possible to steal a thread belonging to a procedure with spawned children outstanding, then the parent of a thread might belong to a kernel which is far away in the kernel tree.

The dag-consistency algorithm takes advantage of locality in the kernel tree by maintaining coherence information on a kernel basis, rather than on a thread basis. Whenever

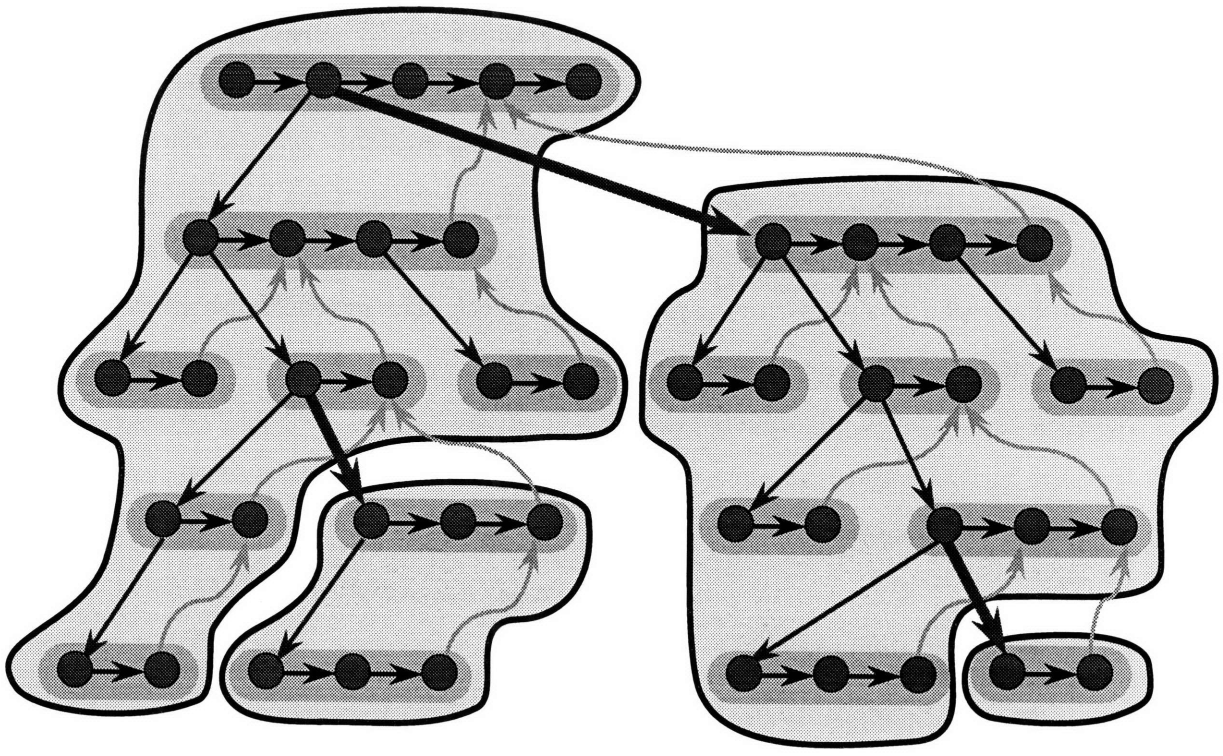

Figure 6-5: This shows a spawn-continue tree partitioned into a kernel tree. The data-dependency edges of the dag are also shown faintly. The spawn-continue tree is partitioned into four kernels, and the spawn edges beginning each new kernel are highlighted.

a processor steals a thread, it creates a *cache* on that processor to store shared-memory objects used by the threads that will be part of the newly created kernel. All reads and writes of an object performed by a thread are performed on the version of the object in the cache of that thread's kernel. In order to propagate changed values correctly, the cache maintains a *dirty bit* for each object that tells whether the object has been modified since it was brought into the cache. An object is *clean* if it has not been modified and *dirty* otherwise. An object may reside in any number of caches at a time, and the value of the object may differ from cache to cache. For simplicity, we assume that there is a fictitious initial thread that precedes all other threads. This initial thread is the only thread in its kernel, and it maintains a cache that always contains every object.

Two operations are used by the dag-consistency algorithm to move objects between caches: fetch and reconcile. If a cache A for a kernel contains an object, a read or write to the object by a thread in the kernel operates directly on the cached object without any object movement. If cache A does not contain the object when the read or write occurs, the dag-consistency algorithm must perform a *fetch* to bring the object into A from another cache B. The fetch operation simply copies the object from cache B into cache A and marks the new version in cache A as clean. The *reconcile* operation is used to remove an object from a cache, typically when a subcomputation completes or when room must be made in the cache for a different object. If cache A contains an object, it can reconcile the object to another cache B only if B also contains the object. The reconcile operation first checks to see whether A's version of the object is clean or dirty. If A's version is clean, then the reconcile operation simply removes the object from cache A. If A's version is dirty, however, then the reconcile operation marks B's version as dirty, updates it to have the same value as A, and then removes the object from cache A.

The dag-consistency algorithm DAGGER operates on the shared memory M of a multithreaded computation as follows:

- Whenever a thread is stolen, its newly created kernel is given an empty cache.
- Whenever a thread i in kernel X accesses an object $m \in M$ that is not resident in X's cache, then m is fetched into X's cache from kernel Y's cache, where Y is the least ancestor of X in the kernel tree such that Y's cache contains m.
- Whenever a kernel X enables a data-dependency edge to a different kernel (its parent in the kernel tree, since the computation is fully strict), every object m in X's cache is first reconciled to kernel Y's cache, where Y is the least ancestor of X in the kernel tree such that Y's cache contains m.

Proof of Correctness

Before we can prove that DAGGER maintains dag consistency, we need to formalize the effects that DAGGER has on the user computation. It suffices to focus on an arbitrary object m, since dag-consistency is defined for each object in M separately. Our strategy is to embed the actions of DAGGER on m into the dag containing the user actions. As in Section 6.2, we assume for simplicity that each thread performs only one action. We specify the important state variables of the system and show how each action affects them.

The important actions performed by the user's code are **read**, **write**, and **sync**. When a thread i performs a read, the value returned by the read is the value of m currently in the cache of thread i's kernel. A write thread updates the value of m in the cache. A sync thread is one which has more than one incoming edge, namely one continue edge and one or more data dependency edges. A sync thread that is part of procedure P executes only after all the outstanding children of procedure P have completed.

In addition to the actions performed by the user's code, we shall also be concerned with the actions **fetch**, **reconcile**, and **steal** performed by DAGGER, which we can view as special threads that are inserted by DAGGER into the computation dag. These threads are inserted to satisfy certain semantic properties. Before a user thread can read or write m when it is not in the cache, a fetch thread is inserted prior to the access. Whenever a data-dependency edge goes from a child kernel to an ancestor, a reconcile thread is added immediately preceding the data-dependency edge. Since the dag is fully strict, this reconcile thread is the final thread executed by the child kernel. Lastly, a steal thread is inserted as the first thread of a kernel whenever a steal occurs. In most cases, when a thread is stolen the steal thread is inserted just before the thread that was stolen. However, when a sync thread is stolen, the steal thread is inserted immediately after the sync thread.

We can view each kernel X as keeping track of the following state variables:

- $\mathit{val}(X) \equiv$ value of m in X's cache, or NIL if m is not in X's cache.
- $\mathit{tag}(X) \equiv$ the tag associated with $\mathit{val}(X)$, or NIL if m is not in X's cache.
- $\mathit{dirty}(X) \equiv$ TRUE if m is dirty and FALSE otherwise.
- $\mathcal{H}(X) \equiv$ the set of "hidden" threads.

The variables $\mathit{tag}(X)$ and $\mathcal{H}(X)$ are for purposes of the proof only. In practice, only the value and dirty bit actually need to be maintained. The ***hidden set*** $\mathcal{H}(X)$ is maintained as a set of threads whose writes to m can no longer be "seen" by threads in X (whether m is in X's cache or not), but $\mathcal{H}(X)$ may also contain threads which do not write to m. The variable $\mathit{tag}(X)$ is used to identify which thread performed the write that stored the value $\mathit{val}(X)$.

We have now embedded the operation of DAGGER on a user computation into a multithreaded computation $G = (V, E)$ in which each thread $i \in V$ performs at most one simple action. We shall next examine how the dag affects the caches of the various subcomputation kernels as it is executed. First, however, we define some helpful notations. For any thread

$i \in V$, we define $K(i)$ to be the kernel to which i belongs. The state variables dynamically change, and so when we need to be precise about the value of a state variable at different times, we superscript the state variable with a parenthesized thread name to indicate the state variable's value immediately after the specified thread executes. For example, $\mathcal{H}^{(i)}(X)$ denotes $\mathcal{H}(X)$ immediately after thread i executes. We will also use $\mathcal{H}^{(i-)}(X)$ to denote $\mathcal{H}(X)$ immediately before thread i executes. At any point in time, we define $la(X)$ to be the least ancestor of kernel X for which $val(X) \neq$ NIL. If X must fetch or reconcile, $la(X)$ is the kernel that DAGGER causes X to fetch from or reconcile to.

We are now ready to describe how the actions performed by a thread affect the state variables of a multithreaded computation $G = (V, E)$. For each action performed by a thread $i \in V$, we shall show how the value, tag, dirty bit, and hidden set are updated in i's kernel as well as in any other kernels affected. All variables not explicitly mentioned in the following pseudocode remain unchanged. Although in practice, many threads execute concurrently, we shall assume that only one action occurs at a time.

The following pseudocode describes the effects of some thread $i \in V$, where $X = K(i)$ is i's kernel. Depending on the action performed by i, we execute one of the following cases:

```
read:
    No change to state variables.

write(v):
    $val(X) \leftarrow v$
    $tag(X) \leftarrow i$
    $dirty(X) \leftarrow$TRUE
    $\mathcal{H}(X) \leftarrow \mathcal{H}(X) \cup \{j \in V : j \prec i\}$

sync:
    $\mathcal{H}(X) \leftarrow \mathcal{H}(X) \cup \mathcal{H}(K(j))$, for all $j \in V$ such that $(j, i) \in E$

fetch:
    $val(X) \leftarrow val(la(X))$
    $tag(X) \leftarrow tag(la(X))$
    $dirty(X) \leftarrow$FALSE
    $\mathcal{H}(X) \leftarrow \mathcal{H}(X) \cup \mathcal{H}(la(X))$
```

reconcile:

```
    If $dirty(X) =$FALSE, do nothing, else:
    $val(la(X)) \leftarrow val(X)$
    $tag(la(X)) \leftarrow tag(X)$
    $dirty(la(X)) \leftarrow$TRUE
    $\mathcal{H}(la(X)) \leftarrow \mathcal{H}(la(X)) \cup \mathcal{H}(X)$
    $val(X) \leftarrow$NIL
    $tag(X) \leftarrow$NIL
    $dirty(X) \leftarrow$FALSE
```

steal:

```
    $\mathcal{H}(X) \leftarrow \mathcal{H}(K(j))$, where $(j,i) \in E$ is the unique incoming edge to $i$
    $val(X) \leftarrow$NIL
    $tag(X) \leftarrow$NIL
    $dirty(X) \leftarrow$FALSE
```

In order to prove that DAGGER maintains dag consistency, we must first understand the ramifications of a dirty bit. The next lemma proves that during the time that a given tag i appears dirty in a kernel Y, all successors of thread i belong to the subtree rooted at Y.

Lemma 9 *Suppose* DAGGER *maintains the shared-memory object m of a multithreaded computation $G = (V, E)$. Then, at any time after the execution of a thread $j \in V$, if $tag(Y) = i$, $dirty(Y) =$* TRUE, *and $i \prec j$, then Y is a (not necessarily proper) ancestor of $K(j)$ in the kernel tree.*

Proof: To prove this lemma we first examine the constrained manner in which dirty objects are manipulated. We shall say that a kernel Y has a *dirty tag i* if $tag(Y) = i$ and $dirty(Y) =$ TRUE. A dirty tag is created by a write by the thread i. If a kernel Y with a dirty tag i reconciles to an ancestor kernel Z, then the dirty tag i moves from Y to Z. When a reconcile to a kernel with dirty tag i is performed, the dirty tag i disappears. In short, a dirty tag i can be created by a write thread i, it can move up the kernel tree, and it can disappear.

To prove the lemma, it suffices to prove the following claim: if dirty tag i exists immediately after executing a thread j, where $i \prec j$, then j belongs to the subtree rooted at the kernel Y containing dirty tag i. The claim implies the lemma, because dirty tags can only move up the kernel tree. Consequently, if immediately after j executes, it belongs to the subtree rooted at a kernel Y with dirty tag i, then j must belong to the subtree rooted at whatever kernel contains the dirty tag i for as long as the dirty tag i exists, which proves the lemma.

We prove the claim using induction on the length of the longest path from i to j. The base case is a path of zero length, in which case $i = j$. The base case is true, since after i executes the dirty tag i is in the kernel $Y = K(i)$. For the induction step, we shall show that if the claim holds for all threads n edges away from i, then it holds for all threads $n+1$ edges away.

To prove the induction step, suppose that thread j is $n+1$ edges away from i. Then, an edge $(k, j) \in E$ must exist, where k is exactly n edges away from i. If $K(k) = K(j)$, then the claim holds trivially, because k and j belong to the same kernel. Thus, we can assume that $K(k) \neq K(j)$ and consider two cases based on the type of the edge from k to j. For the first case, suppose that (k, j) is a spawn or continue edge. Then j must be a steal thread, and thus, $K(j)$ is a child of $K(k)$. Consequently, since k belongs to the subtree rooted at the kernel Y containing dirty tag i, it follows that j must also belong to the subtree rooted at Y. For the second case, suppose (k, j) is a data-dependency edge. Then, since any data-dependency edge between threads in different kernels always goes to a thread in the parent kernel, $K(j)$ must be a parent of $K(k)$, which implies that k is the last thread to execute in $K(k)$ and k is a reconcile thread. Consequently, $K(k) \neq Y$, because Y's dirty bit is TRUE, and the reconcile sets $K(k)$'s dirty bit to FALSE. Since by induction $K(k)$ is in the subtree rooted at Y and $K(k) \neq Y$, the parent of $K(k)$, namely $K(j)$, must be in the subtree rooted at Y, which proves the claim and the lemma. ■

The next lemma establishes two monotonicity properties of hidden sets. The first property says that the hidden set of any kernel monotonically increases with time as actions are performed. The second property says that the hidden set of a thread increases monotonically along any path in the dag, where by hidden set of a thread, we mean the hidden set of that thread's kernel immediately after the thread executes.

Lemma 10 *Suppose* DAGGER *maintains the shared-memory object m of a multithreaded computation $G = (V, E)$. Then, for all threads $i, j \in V$ such that $i \prec j$ and kernel Y, we* *have*

$$\mathcal{H}^{(i)}(Y) \subseteq \mathcal{H}^{(j)}(Y) \tag{6.1}$$

$$\mathcal{H}^{(i)}(K(i)) \subseteq \mathcal{H}^{(j)}(K(j)) \tag{6.2}$$

Proof: Property 6.1 follows directly by induction on the actions in G, because no action removes elements from a kernel's hidden set. We prove Property 6.2 by showing that it holds for every edge $(i, j) \in E$, which implies that it holds whenever $i \prec j$. If i and j are in the same kernel, Property 6.1 implies the property. Otherwise, i and j are in different kernels, and we examine two cases depending on the type of edge (i, j). If (i, j) is a continue or spawn, then j performs a steal action, and thus $\mathcal{H}(j) = \mathcal{H}(i)$, and the property holds. If (i, j) is a data-dependency edge, then j performs a join actions, and thus $\mathcal{H}(j) \supseteq \mathcal{H}(i)$, and the property holds. ∎

We now prove three invariants on hidden sets. The first says that for any kernel Y, the hidden set of the kernel that Y would fetch from includes everything in Y's hidden set. This invariant ensures that when Y fetches the object m, the value received can be seen by Y. The second invariant says that if m is dirty in a cache, only descendent kernels can have the thread that wrote the value in their hidden sets. The third invariant says that the value in a kernel's cache was not written by a thread in its hidden set. In other words, the hidden set of a kernel does indeed represent those threads whose writes the kernel cannot see.

Lemma 11 *Suppose* DAGGER *maintains the shared-memory object m of a multithreaded computation $G = (V, E)$. Then, for all kernels X and Y, the following statements are invariant during the execution of G:*

1. $tag(Y) = \text{NIL} \Longrightarrow \mathcal{H}(Y) \subseteq \mathcal{H}(la(Y))$.

2. $dirty(Y) = \text{TRUE}$ *and* $tag(Y) \in \mathcal{H}(Z) \Longrightarrow Z$ *is a descendant of* Y.

3. $tag(Y) \notin \mathcal{H}(Y)$.

Proof: We will proof these invariants by induction on the actions.

To prove Invariant 1, observe that all hidden sets are initially empty. Therefore Invariant 1 holds initially, providing the base case. We now examine each action and show that if the invariant is true before the action, then it is true afterwards. Note that the invariant we are proving mentions two kernels, Y and $la(Y)$. For each kernel, X, that has its state variables changed by an action, we will have to show that the invariant holds in two cases: (1) the invariant must holds for X and its least ancestor $la(X)$, and (2) the invariant must hold for any descendants of X, Z, where, either before or after the operation, $X = la(Z)$.

For a **read** thread, the invariant holds trivially, since no state variables change.

For a **write** performed by thread i in kernel X, the state variables of kernel X are modified, so we must show the invariant are maintained on kernel X for both cases mentioned above. For case (1) the invariant holds trivially since $tag(X) \neq \text{NIL}$. For case (2), we note that the write operation can not affect which kernel is the least ancestor of another. So X is the least ancestor of a node, Z, after a sync if and only if X was the least ancestor of Z before the sync. We must show that if a kernel Z exists such that X is the least ancestor of Z and $tag(Z) = \text{NIL}$, then $\mathcal{H}(Z) \subseteq \mathcal{H}(X)$. If such a kernel Z exists then by induction the invariant held before the operation, namely $\mathcal{H}(Z) \subseteq \mathcal{H}^{(i-)}(X)$. Since the write operation can only increase $\mathcal{H}(X)$, the invariant holds after the write operation as well.

For a **fetch** performed by thread i in kernel X, we must show that the invariant holds for the modified kernel X. For case (1) the invariant holds trivially since $tag(X) \neq \text{NIL}$. For case (2) we must show that if there exists some kernel Z such that X is the least ancestor of Z and $tag(Z) = \text{NIL}$, then $\mathcal{H}(Z) \subseteq \mathcal{H}(X)$. If such a kernel Z exists then before the fetch operation $la(X)$ was the least ancestor of Z. By induction $\mathcal{H}(Z) \subseteq \mathcal{H}^{i-}(la(Z) = la(X))$, and by the definition of the fetch action $\mathcal{H}^{i-}(la(X)) \subseteq \mathcal{H}(X)$; therefore, $\mathcal{H}(Z) \subseteq \mathcal{H}(X)$ which maintains the invariant.

For a **reconcile** performed by thread i in kernel X to kernel $Y = la(X)$, we must show that the invariant holds both for kernel X and for kernel $Y = la(X)$. We will deal with kernel X first. For case (1), by the definition of the reconcile action $\mathcal{H}(Y) \leftarrow \mathcal{H}(Y) \cup \mathcal{H}(X)$, so $\mathcal{H}(X) \subseteq \mathcal{H}(Y)$ and the invariant holds. For case (2) we must show that if there were some kernel Z with $tag(Z) = \text{NIL}$ which had X as its least ancestor, then after the reconcile $\mathcal{H}(Z) \subseteq \mathcal{H}(la(Z))$. Before the reconcile X was the least ancestor of Z so by induction $\mathcal{H}(Z) \subseteq \mathcal{H}(X)$. After the reconcile $Y = la(Z)$ and from the definition of reconcile it follows that $\mathcal{H}(Y) \supseteq \mathcal{H}(X)$. Therefore $\mathcal{H}(Z) \subseteq \mathcal{H}(Y)$ and the invariant holds.

Now we will show the invariant is maintained for kernel $Y = la(X)$. For case (1) the invariant holds trivially since $tag(X) \neq \text{NIL}$. For case (2) we must show that if there exists some kernel Z such that Y is the least ancestor of Z and $tag(Z) = \text{NIL}$, then $\mathcal{H}(Z) \subseteq \mathcal{H}(X)$. We have already dealt with the case where before the reconcile the least ancestor of Z was X, all that remains is the case where before the reconcile the least ancestor of Z was Y. By induction $\mathcal{H}(Z) \subseteq \mathcal{H}(Y)$ before the reconcile, and since the reconcile can only increase $\mathcal{H}(Y)$ this invariant is still true after the reconcile.

For a **steal** performed as the first thread of kernel X, we must show that the invariant holds for kernel X. For case (1), since $val(X) = \text{NIL}$ we must show that $\mathcal{H}(X) \subseteq \mathcal{H}(la(X))$. The definition of the steal action states that $\mathcal{H}(X) \leftarrow \mathcal{H}(K(j))$, where $(j,i) \in E$ is the unique incoming edge to i. If $val(K(j)) \neq \text{NIL}$ then $K(j)$ is the least ancestor of X, and the invariant holds since $\mathcal{H}(X) = \mathcal{H}(la(X))$. If $val(K(j)) = \text{NIL}$ then $la(X) = la(K(j))$, and by induction $\mathcal{H}(K(j)) \subseteq \mathcal{H}(la(K(j)))$. Therefore $\mathcal{H}(X) = \mathcal{H}(K(j)) \subseteq \mathcal{H}(la(X))$ and the invariant holds. For case (2) the proof is trivial since X has no descendants.

For a **sync** performed by thread i in kernel X, we must show that the invariant holds for kernel X.

The sync thread i has one incoming continue edge from some thread k and one or more incoming data-dependency edges. We have $X = K(i) = K(k)$, because only steal threads can have an incoming continue edge from another kernel. Also, for the data-dependency edges either they come from threads in the same kernel, or they come from threads in kernels, whose *val* is NIL. For case (1), we must show that if $val(X) = \text{NIL}$ then $\mathcal{H}(X) \subseteq \mathcal{H}(la(X))$. We will assume $val(X) = \text{NIL}$, otherwise the proof is trivial. The definition of sync states that $\mathcal{H}(X) \leftarrow \mathcal{H}(X) \cup \mathcal{H}(K(j))$, for all $j \in V$ such that $(j,i) \in E$. Lets look at each set unioned to form $\mathcal{H}(X)$ and show that all of these sets are subsets of $\mathcal{H}(la(X))$. For $\mathcal{H}(X)$ we have by induction $\mathcal{H}^{(i-)}(X) \subseteq \mathcal{H}(la(X))$, The rest of the sets unioned in each correspond to an incoming edge. For the incoming continue edge (k,i), we have $K(k) = X$, so this just unions in $\mathcal{H}(X)$ again. For each incoming data-dependency edge (j,i) we have either $K(j) = K(i) = X$, in which case we just union in $\mathcal{H}(X)$ yet again, or we have $K(j) \neq X$. In this second case we have $val(K(j)) = \text{NIL}$ and $la(K(j)) = la(X)$ and, by induction, $\mathcal{H}(K(j)) \subseteq \mathcal{H}(la(K(j))$. Therefore $\mathcal{H}(K(j)) \subseteq \mathcal{H}(la(X))$ which shows that $\mathcal{H}(X) \subseteq \mathcal{H}(la(X))$. Finally, for case (2) we note that the sync operation only changes the hidden set, so X is the least ancestor of a node, Z, after a sync iff X was the least ancestor of Z before the sync. By induction the invariant holds before the operation, Since the sync operation can only increase $\mathcal{H}(X)$, the invariant holds after the sync operation as well.

To prove Invariant 2, we will again use induction on the actions. Informally, Invariant 2 states that if kernel X has a dirty tag i then i can only appear in hidden sets which are in the subtree rooted at X. All hidden sets are initially empty so the base case of the proof is trivial. We now need to examine each action and show that if Invariant 2 is true before the action, then it is true afterwards. As with the previous example, there are two kernels in the invariant, so there are two cases we must consider. Case (1) occurs when we modify the variables of a kernel X, and we set $\mathit{dirty}(X)$ to TRUE or modify $\mathit{tag}(X)$. In this case we must show that for all Z such that $\mathit{tag}(X) \in \mathcal{H}(Z)$, Z is a descendant of X. Case (2) occurs when we modify the hidden set of a kernel X. In this case we must show that the threads added to $\mathcal{H}(X)$ do not cause the invariant to be invalidated. In particular, we must show that the threads added to $\mathcal{H}(X)$ do not appear dirty in a kernel which is a proper descendant of X.

For **read**, the invariant holds trivially since there are no changes.

For a **write** performed by a thread i in kernel X, we modify both the hidden set and the tag, so we must consider both cases. For case (1), since $\mathit{tag}(X) =$ TRUE we must show that for all Z such that $\mathit{tag}(X) \in \mathcal{H}(Z)$, Z is a descendant of X. First notice that only the write operation adds threads to a hidden set that are not already in some other hidden set, and the write operation only adds threads that are predecessors of the write thread. This implies that a thread j can not appear in any hidden set before thread j executes. Therefore when write thread i executes i does not appear in any hidden set. Therefore there can be no Z such that $i = \mathit{tag}(X) \in \mathcal{H}(Z)$.

For case (2), since we add to $\mathcal{H}(X)$ we must show that the threads added to $\mathcal{H}(X)$ do not appear dirty in a kernel which is a proper descendant of X. We prove this by contradiction. Assume that added thread j does appear dirty in a kernel Y which is a proper descendant of X. Then we have $\mathit{dirty}(Y) =$ TRUE and $\mathit{tag}(Y) = j$ and $j \prec i$. By Lemma 9, Y is an ancestor of X, but by our assumption Y is a proper descendant of X. This contradiction completes the proof for case (2).

For a **fetch** by a thread in kernel K, we need not consider case (1) since $\mathit{dirty}(X)$ is set to FALSE. For case (2) we will first make the observation that adding a thread j to the hidden set of any kernel Y can not invalidate the invariant if j is already in the hidden set of an ancestor of Y. This observation is true either because no dirty tag j exists, in which case j can be added to any hidden set without breaking the invariant, or because dirty tag

j exists in some kernel Z and Y is a descendant of Z, in which case j can be added to any descendant of Y because such a descendant is also a descendant of Z. Since every thread added to $\mathcal{H}(X)$ by the fetch operation is already in the hidden set of an ancestor of X, adding these threads preserves the invariant.

For a **reconcile** performed by thread i in kernel X to kernel $Y = la(X)$, we need consider cases (1) and (2) for kernel Y. For kernel X there are no cases to consider since $\mathcal{H}(X)$ is unmodified and $tag(X)$ is set to NIL. For case (1) of Y we must show that $tag(Y)$ only appears in hidden sets that are descendants of Y. By induction, before the reconcile $tag(Y)$ only appeared in hidden sets that were descendants of X. Since all descendants of X are also descendants of Y, $tag(Y)$ only appears in hidden sets that are descendants of Y. For case (2) of Y we must show that the threads added to $\mathcal{H}(Y)$ do not appear as a dirty tag in a kernel which is a proper descendant of Y. The only threads added to $\mathcal{H}(Y)$ are those in $\mathcal{H}(X)$. Consider the state before the reconcile. We know by induction that if one of these added threads, j, appeared as a dirty tag, then X was a descendant of $Z = K(j)$. Since Y is the least ancestor of X, Y must also be a (not necessarily proper) descendant of Z. Therefore, the threads added to $\mathcal{H}(Y)$ can only appear as a dirty tag in a kernel which is a proper ancestor of Y; thus the invariant is maintained.

For a **steal** performed as the first thread of kernel X, we need not consider case (1) since $tag(X)$ is set to NIL. For case (2) we notice that all threads added to the $\mathcal{H}(X)$ are taken from an ancestor, therefore, as shown for the fetch operation, the invariant is maintained.

For a **sync** performed by thread i in kernel X, we must show that the invariant holds for case (2). All threads added to $\mathcal{H}(X)$ are from the hidden set of a child of X. To show that these added threads do not break the invariant we will use a simplified version of the argument made for reconcile. By induction, if Y, a child of X, has a thread j in its hidden set that appears as a dirty tag in kernel Z, then Z must be a ancestor of Y. Since $tag(Y) =$ NIL, Z must be a proper ancestor of Y. Since X is the parent of Y, Z must also be an ancestor of X. Therefore, the threads added to $\mathcal{H}(X)$ can only appear as a dirty tag in a kernel which is an ancestor of X; thus the invariant is maintained.

To prove Invariant 3, the final invariant, observe that all hidden sets are initially empty. Consequently, $tag(Y) \notin \mathcal{H}(Y)$ before the computation begins, which provides the base case of the induction. We now examine each action and show that if the invariant is true before the action, then it is true afterwards.

For **read**, the invariant holds trivially, since no state variables change.

For a **write** performed by thread i in kernel X, thread i is not added to the hidden set $\mathcal{H}(X)$, and so we need only to show that i is not in $\mathcal{H}(X)$ before the write. Observe that the only time when a thread j that not in any hidden set is added to some hidden set is when a successor of j performs a write. Before i executes, none of its successors have executed, and thus, i belongs to no hidden set. In particular, we have $i \notin \mathcal{H}(X)$.

For a **fetch** performed by a thread in kernel X, to show that the invariant holds after the thread executes, we must show $tag(X) \notin \mathcal{H}^{(i-)}(X)$ and $tag(X) \notin \mathcal{H}^{(i-)}(la(X))$. The second is true by induction, which implies that the first is true, because Invariant 1 guarantees $\mathcal{H}(X) \subseteq \mathcal{H}(la(X))$.

For a **reconcile** performed by a thread in kernel X, we must show the invariant is maintained for both the modified kernels X and $Y = la(X)$. The invariant holds trivially for kernel X, since $tag^{(i)}(X) = \text{NIL}$. To show it is true for kernel Y, we must show that $tag(X) \notin \mathcal{H}(X)$ and $tag(X) \notin \mathcal{H}(Y)$. The first holds by induction, and the second follows from Invariant 2.

For a **steal** of a thread now in kernel X, the invariant holds trivially, since $tag(X)$ is set to be NIL.

For a **sync** by thread i in kernel X, the sync thread i has one incoming continue edge from some thread k and one or more incoming data-dependency edges. We have $X = K(i) = K(k)$, because only steal threads can have an incoming continue edge from another kernel. If $tag(X) = \text{NIL}$, then the invariant is trivially maintained. Otherwise, we must show that none of the hidden sets that are unioned to form $\mathcal{H}(X)$ contain $tag(X)$. Initially, by induction $tag(X) \notin \mathcal{H}(X)$, and so we must show that for each incoming data-dependency edge $(j, i) \in E$, we have $tag(X) \notin \mathcal{H}(K(j))$. If thread k also belongs to kernel X, then trivially $tag(X) \notin \mathcal{H}(K(j)) = \mathcal{H}(X)$. If $K(k) = Y \neq X$, however, then because any data-dependency edge between threads in different kernels always goes to a thread in the parent kernel, Y is a child of X in the kernel tree. Moreover, Y has completed, and the last thread of Y was a reconcile. Therefore, we have $tag(Y) = \text{NIL}$, which implies that $X = la(Y)$. Consequently, by Invariant 1, we have $\mathcal{H}(Y) \subseteq \mathcal{H}(X)$. By induction, we know that $tag(X) \notin \mathcal{H}(X)$ before i executes, and therefore $tag(X) \notin \mathcal{H}(Y)$. ■

We are now ready to prove the correctness of DAGGER.

Theorem 12 *If the shared memory M of a multithreaded computation $G = (V, E)$ is maintained using* DAGGER, *then M is dag consistent.*

Proof: We must show that both parts of Definition 8 hold. The first part holds trivially. To prove that the second part holds, consider three threads $i, j, k \in V$, where $i \prec j \prec k$. Since j performs a write and $i \prec j$, by the pseudocode for the write action, it follows that $i \in \mathcal{H}^{(j)}(K(j))$. Moreover, since $j \prec k$, Lemma 10 implies that $\mathcal{H}^{(j)}(K(j)) \subseteq \mathcal{H}^{(k)}(K(k))$. Thus, we have $i \in \mathcal{H}^{(k)}(K(k))$, which with Invariant 3 implies that $tag^{(k)}(K(k)) \neq i$. ∎

The algorithm DAGGER depends heavily on the fact that computations are fully strict, but the basic ideas in DAGGER can be extended to nonstrict computations. The idea is that whenever a data dependency edge goes from a sending kernel X to a receiving kernel Y, we first find the least common ancestor Z of X and Y in the kernel tree. Then, we walk up the tree from X reconciling at each kernel along the way until Z is reached. In effect, this action unions X's hidden set into Z's. Lastly, we walk up the tree from Y to Z, reconciling at each kernel along the way. At this point, $la(X)$ is either Z or an ancestor of Z, and thus, when Y next fetches, it obtains a hidden set that includes X's, thus ensuring that the second part of Definition 8 is met. In all other respects, this algorithm is the same as the DAGGER algorithm, and when the dag is fully strict, this procedure reduces to the basic DAGGER algorithm, since $Y = Z$. The DAGGER algorithm is simpler to implement, however.

In this section we have proven that the DAGGER algorithm maintains dag consistency. See [BFJ+96] for another algorithm we have implemented which also maintains dag consistency.

6.4 Implementation

This section describes our implementation of dag-consistent shared memory for the Cilk multithreaded runtime system running on the Connection Machine CM-5 parallel supercomputer [LAD+92]. We first describe the Cilk language extensions for supporting shared-memory objects and the "diff" mechanism [KCDZ94] for managing dirty bits. We then describe the distributed "cactus-stack" [HD68, Mos70, Ste88] memory allocator which the system uses to allocate shared-memory objects. Finally, we describe the mechanisms used by the runtime system to maintain dag-consistency.

The Cilk system on the CM-5 supports concrete shared-memory objects of 32-bit words. All consistency operations are logically performed on a per-word basis. If we were to allow every word to be fetched and reconciled independently, however, the system would be terribly inefficient. Since extra fetches and reconciles do not adversely affect the consistency algorithm, we implemented the familiar strategy of grouping objects into *pages* [HP90, 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 fetch/reconcile overhead.

Unfortunately, the CM-5 operating system does not support handling of page faults by the user, and so we were forced to implement shared memory in a relatively expensive fashion. Specifically, in our CM-5 implementation, shared memory is kept separate from the other user memory, and special operations are required to operate on it. Most painfully, testing for page faults occurs explicitly in software, rather than implicitly in hardware. Our Cilk-to-C type-checking preprocessor [Mil95] alleviates some of the discomfort, but a transparent solution that uses hardware support for paging would be much preferable. A minor advantage to the software approach we use, however, is that we can support full 64-bit addressing of shared memory on the 32-bit Sparc processors of the CM-5 system.

Cilk's language support makes it easy to express operations on shared memory. The user can declare `shared` pointers and can operate on these pointers with normal C operations, such as pointer arithmetic and dereferencing. The type-checking preprocessor automatically generates code to perform these operations. The user can also declare `shared` arrays which are allocated and deallocated automatically by the system. As an optimization, we also provide `register shared` pointers, which are a version of `shared` pointers that are optimized for multiple accesses to the same page. In our CM-5 system, a `register shared` pointer dereference is about 5 cycles slower than an ordinary C pointer dereference when it performs multiple accesses to within a single page. Finally, Cilk provides a loop-hole mechanism to convert `shared` pointers to C pointers, which allows direct, fast operations on pages, but requires the user to keep the pointer within a single page. We hope to port Cilk in the near future to an architecture and operating system that allow user-level code to handle page faults. In such a system, no difference need exist between `shared` objects and their C equivalents, and operations on shared memory can be implemented transparently with no per-access overhead.

An important implementation issue that we faced with the software implementation of dag-consistent shared memory on the CM-5 was how to keep track of which objects on a page have been written. Rather than using dirty bits explicitly, as the DAGGER algorithm from Section 6.3 would suggest, Cilk uses a *diff* mechanism as is used in the Treadmarks system [KCDZ94]. The diff mechanism computes the dirty bit for an object by comparing that object's value with its value in a copy made at fetch time. Our implementation makes this 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 allows the user to manipulate a page using an ordinary C pointer that incurs no run-time system overhead [ZSB94].

We needed to support the detection of writes in software, because the CM-5 provides no direct hardware support to maintain dirty bits explicitly at the granularity of words. We rejected out of hand the unpleasant alternative of requiring the user to maintain his own dirty bits. Since we have limited compiler support and we wish to call existing C code from Cilk procedures, we determined that it would be too difficult to modify our Cilk-to-C type-checking preprocessor to automate the maintenance of explicit dirty bits. Likewise, we felt that the strategy of using binary rewriting to detect writes in software [BZS93] would entail too much effort. We finally settled on the diff mechanism for its simplicity.

Some means of allocating memory must be provided in any useful implementation of shared memory. We considered implementing general heap storage in the style of C's `malloc` and `free`, but most of our immediate applications only required stack-like allocation for temporary variables and the like. Since Cilk procedures operate in a parallel tree-like fashion, however, we needed some kind of parallel stack. We settled on implementing a *cactus-stack* [HD68, Mos70, Ste88] allocator.

From the point of view of a single Cilk procedure, a cactus-stack behaves much like an ordinary stack. The procedure can allocate and free memory by incrementing and decrementing a stack pointer. The procedure views the stack as a linearly addressed space extending back from its own stack frame to the frame of its parent and continuing to more distant ancestors.

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 6-6. In

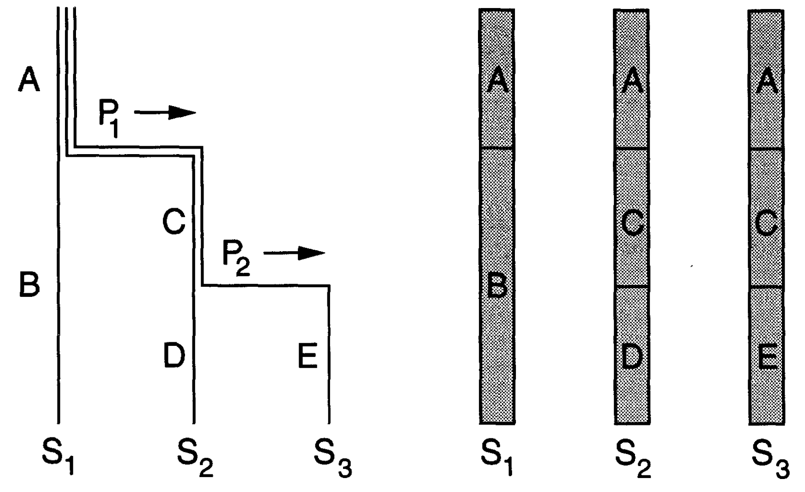

Figure 6-6: A cactus-stack. Procedure P_1 is stolen from subcomputation S_1 to start subcomputation S_2, and then procedure P_2 is stolen from S_2 to start subcomputation S_3. Each subcomputation sees its own stack allocations and the stack allocated by its ancestors. The stack grows downwards. The left side of the picture shows how the stack grows like a tree, resembling a cactus. The right side shows the stack as seen by the three subcomputations. In this example, the stack segment A is shared by all subcomputations, stack segment C is shared by subcomputations S_2 and S_3, and the other segments are private.

the figure, subcomputation S_1 allocates some memory A before procedure P_1 is spawned. Subcomputation S_1 then continues to allocate more memory B. When procedure P_1 is stolen and becomes the root of subcomputation S_2, a new branch of the stack is started so that subsequent allocations performed by S_2 do not interfere with the stack being used by S_1. The stacks as seen by S_1 and S_2 are independent below the steal point, but they are identical above the steal point. Similarly, when procedure P_2 is stolen from S_2 to start subcomputation S_3, the cactus stack branches again.

Cactus-stack allocation mirrors the advantages of an ordinary procedure stack. Any object on the stack that is viewable by a procedure has a simple address: its offset from the base of the stack. Procedure local variables and arrays can be allocated and deallocated automatically by the runtime system in a natural fashion, as was shown in the matrix multiplication example in Figure 6-2. Allocation can be performed completely locally without communication by simply incrementing a local pointer, although communication may be required when an out-of-cache stack page is actually referenced. Separate branches of the cactus stack are insulated from each other, allowing two subcomputations to allocate and free objects independently, even though objects may be allocated with the same address.

Procedures can reference common data through the shared portion of their stack address space.

Cactus stacks have many of the same limitations as ordinary procedure stacks [Mos70]. For instance, a child thread cannot return to its parent a pointer to an object that it has created. Similarly, sibling procedures cannot share storage that they create on the stack. Just as with a procedure stack, pointers to objects allocated on the cactus-stack can only be safely passed to procedures below the allocation point in the call tree. Heap storage offers a way of alleviating some of these limitations (and we intend to provide a heap allocator in a future version of Cilk), but the cactus stack provides simple and efficient support for allocation of procedure local variables and arrays.

The CM-5 implementation of Cilk supports shared memory by combining the DAGGER algorithm for maintaining consistency with a cactus-stack allocator. Two regions are allocated within the primary memory of each processor. The first is the *page cache*, which contains local copies of pages for the threads running on the processor, and the second is the *backing store*, which is a distributed repository for pages that for one reason or another were forced out of the processor caches. In addition, various data structures for memory management are kept in each processor.

In Section 6.3, we assumed that an initial thread exists whose cache contains every object. In the CM-5 implementation of Cilk, the backing store serves as the cache of this fictitious initial thread. Our implementation of DAGGER accesses the backing store as if it were an ordinary cache, but unlike an ordinary cache which can discard objects when it runs out of room, the backing store never discards an object. Consequently, the size of the backing store determines how large a shared-memory application one can run. On the CM-5, the backing store is implemented in a distributed fashion by allocating a large fraction of each processor's memory to this function. When space for a new backing store page is needed, it is requested from a processor uniformly at random. This policy ensures that backing store is spread evenly across the processors' memory. Consequently, we can run applications that use up to about half the total available primary memory on all processors. In other systems, it might be reasonable to place the backing store on disk *à la* traditional virtual memory.

The page manager for the shared-memory system keeps information on each active and suspended kernel. Specifically, it maintains the base and limit of the portion of cactus stack that has been allocated by the threads of the kernel. It also uses a hash table to keep track of which pages are currently in the local page cache, which pages it has allocated but which do not yet exist, and the backing-store addresses for any pages that it has allocated and for which a copy exists in the backing store. The page manager also keeps track of user references to each page in the local cache so that it can perform LRU page replacement when the cache becomes full.

When a page is allocated on the stack, the stack limit is increased, but no storage is assigned for the page until it is actually used. Thus, allocation is an extremely cheap operation, as it is in sequential stack-based languages such as C. When the page is accessed for the first time, storage is allocated for it in the cache of the accessing processor. When a subcomputation completes, all of the pages in the cache whose addresses lie beyond the local stack limit are discarded, since those were allocated by completed procedures in the kernel of the terminating subcomputation. In addition, if any of these pages have been written to the backing store, they are removed and the space for them in the backing store is freed. The pages in the cache whose addresses indicate that they were allocated by ancestors are reconciled with their least ancestor in accordance with the DAGGER algorithm. If no ancestor has a copy of a given page, then the page is written to a random location in the backing store, and the ancestor that allocated the page keeps track of the backing-store location.

Most of the actions on shared-memory described in the DAGGER algorithm from Section 6.3 can be implemented straightforwardly. The only action of significant complexity occurs when a subcomputation kernel needs to find its least ancestor holding a particular page. On the CM-5, we call the process of finding this ancestor *tree climbing*, because we climb up the tree of kernels until we find the page in question. We also considered a directory-based algorithm, but it would have been more complex to implement, and so for our first implementation, we opted for the simpler strategy.

Tree climbing for fetching and reconciling are similar, and so we shall describe here only the steps taken by our implementation when a thread fetches an object on a page on its stack. When a page reference occurs, the page manager within the processor takes one of four actions:

1. If the page is in the cache, the user's action is performed on the cached copy directly.

2. If the page is not in the cache and its stack address is beyond the current stack limit, then the page manager signals an error, since the page is not allocated.

3. If the page is not in the cache and its stack address indicates that this thread's kernel was responsible for allocating the page, then the page manager goes directly to the backing store to fetch it, since none of the kernel's ancestors hold a copy. This action may cause another page to be ejected from the cache, which may itself cause tree climbing for reconciliation.

4. If the page is not in the cache and its stack address indicates that one of the ancestors of the thread's kernel allocated the page, then the page manager sends a message to the kernel's parent to fetch the page for the faulting thread recursively.

This last action keeps climbing the kernel tree until one of two events occur. If the page is found in an ancestor's cache, then we fetch the page out of that cache. If we reach the ancestor that allocated the page and its cache does not currently have the page, however, we fetch the page directly from backing store. Once the page has been obtained, we add it to the kernel's cache and return control to the user thread to make use of it.

6.5 An Analysis of Page Faults

In this section, we analyze the number F_P of page faults that a (well-structured) computation incurs when run on P processors using Cilk's randomized work-stealing scheduler and the implementation of dag-consistent shared memory described in Section 6.4. We prove that F_P can be related to the number F_1 of page faults taken by a 1-processor execution by the formula $F_P \leq F_1 + 2Cs$, where C is the size of each processor's cache in pages and s is the total number of steals executed by Cilk's provably good work-stealing 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 theorem assumes that the application is well-structured, in the sense described in Section 6.3. The proof takes advantage of properties of the least-recently used (LRU) page replacement scheme used by Cilk.

Theorem 13 *Let F_P be the number of page faults of a well-structured Cilk computation when run on P processors, and let C be the size of of each processor's cache in pages. Then, we have $F_P \le F_1 + 2Cs$, 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 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, and hence forms a leaf in the kernel tree. 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 $F_1 + 2C(s-1)$ 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 that 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 may be, after one of the two executions takes C page faults, the states of the two caches must be identical. Indeed, at the point when one execution has just taken its Cth page fault, each cache contains exactly the last C distinct pages referenced [JD73].

We now use this property of LRU to count the number of page faults of the execution E. The fault behavior of E is the same as the fault behavior of E' except for the subcomputation T and its parent, call it U, in the kernel tree. The only difference between the two executions is that the starting cache state of T and the starting cache state of the section 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 $F_1 + 2C(s-1) + 2C = F_1 + 2Cs$ page faults. ■

Theorem 13 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 its victim 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

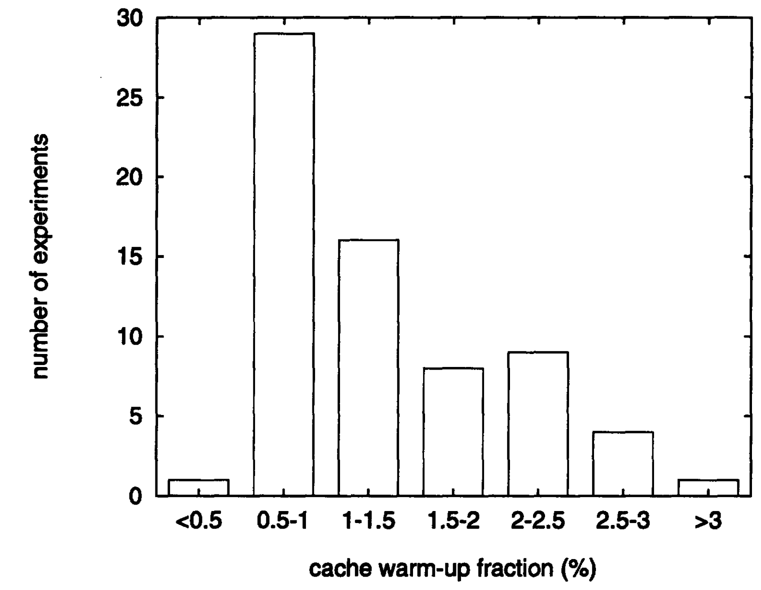

Figure 6-7: Histogram of the cache warm-up fraction $(F_P - F_1)/2Cs$ for a variety of applications, cache sizes, processor counts, and problem sizes. The vertical axis shows the number of experiments with a cache warm-up fraction in the shown range.

execution.

To measure the warm-up overhead, we counted the number of page faults taken by several applications—including `matrixmul`, a parallel version of Strassen's algorithm [Str69], and a parallel version of a Barnes-Hut N-body code [BH86]—for various choices of cache, processor, and problem size. For each run we measured the *cache warm-up fraction* $(F_P - F_1)/2Cs$, which represents the fraction of the cache that needs to be warmed up on each steal. We know from Theorem 13 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 6-7 showing the cache warm-up fraction for 72 experimental runs of the above applications, with processor counts ranging from 2 to 64 and cache sizes from 256KB to 2MB. Figure 6-8 shows a particular example for the page faults during the multiplication of 512×512 matrices using a 1-megabyte cache and 4-kilobyte pages, from which it can be seen that the constant overhead multiplying the number s of steals is closer to 9 than to $2C = 512$.

The reason why the cache warm-up costs are so low can be explained by examining the distribution of stolen problem sizes. We performed an experiment that recorded the size of each subproblem stolen, and we noticed that most of the tasks stolen during an execution

Processors		1	2	4	8	16	32	64
Steals		0	16	31	144	424	1053	1982
Page Faults	F_P	9552	9557	9760	10216	14151	19508	24634
	$F_1 + 2Cs$	9552	13648	17488	46416	118096	279120	516944
	$F_1 + 9s$	9552	9696	9831	10848	13368	19029	27391

Figure 6-8: Page faults versus the number of processors for a 512×512 multiply with a 1 Megabyte cache. We show the number of successful steals s, the number of page faults F_P, our upper bound $F_1 + 2Cs$ on the number of page faults, and our approximation $F_1 + 9s$.

were quite small. In fact, only 5–10% of the stolen problems were "large," where a large subproblem is defined to be one which takes C or more pages to execute. The other 90–95% of the tasks are small. Therefore, most of the stolen subcomputations never perform C page faults before terminating. Thus, the bound $F_P \leq F_1 + 2Cs$ derived in Theorem 13 is a very conservative bound, and in practice we see less than 3% of the extra $2Cs$ steals.

6.6 Conclusion

Many other researchers have investigated distributed shared memory. To conclude, we briefly discuss related work and offer some ideas for the future.

The notion that independent tasks may have incoherent views of each others' memory is not new to Cilk. The BLAZE [MR87] language incorporated a memory semantics similar to that of dag-consistency into a PASCAL-like language. The Myrias [BBZ88] computer was designed to support a relaxed memory semantics similar to dag-consistency, with many of the mechanisms implemented in hardware. Loosely-Coherent Memory [LRV94] 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 [LH89] and others [CBZ91, FLA94, KCDZ94], Cilk's implementation uses fixed-sized pages to cut down on the overhead of managing shared objects. In contrast, systems that use cache lines [CA94, KOH$^+$94, RLW94] require some degree of hardware support [SFL$^+$94] to manage shared memory efficiently at the granularity of cache lines. As another alternative, systems that use arbitrary-sized objects or regions [CAL+89, JKW95, TBK93] require either an object-oriented programming model or explicit user management of objects.

As we have gained experience programming with dag consistency, we have encountered some deficiencies of dag consistency that tend to make certain programming idioms inefficient. For example, consider a serial program that calls two procedures, each of which increments a variable by a certain amount. To parallelize this program using dag consistency, one cannot merely spawn the two procedures in parallel, because the update of one may be lost. Instead, a copy of the variable must be made for one of the procedures and when they both return, the parent must add the value of the copy into the original variable. This extra copying can be very expensive when it occurs in the inner loop of a program. A big advantage of direct hardware support for Lamport's model of sequential consistency [Lam79] is that no copying of temporaries need occur. We are currently investigating how this kind of problem can be solved efficiently in Cilk without direct hardware support.

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. We are currently investigating how to incorporate file I/O in our system.

We are also currently working on porting dag-consistent shared memory to our Cilk-NOW [Blu95] adaptively parallel, fault-tolerant, network-of-workstations system. We are using operating system hooks to make the use of shared memory be transparent to the user. We expect that the well-structured nature of Cilk computations will allow subcomputations to maintain coherent views of shared memory efficiently, even in the presence of processor faults.

Chapter 7

Cilk-4: Supporting Speculative Computations

The Cilk-4 system, which is still in the process of being implemented, is intended to remedy a deficiency in the Cilk-2 language. When we designed Cilk-2 and added support for procedures with call/return semantics, we were able to rewrite almost all existing programs using the new, simpler, Cilk-2 syntax. The only existing application which can not be expressed in the Cilk-2 style is ⋆Socrates. We can not express ⋆Socrates using the higher-level Cilk-2 constructs in large part because the control structure of the parallel search algorithm used in ⋆Socrates is fairly complex, and includes speculative computations which may be killed off. For most algorithms, the order in which the user's threads are executed affects neither the number of threads created nor the work performed by those threads. But for speculative algorithms, like the Jamboree search algorithm at the heart of ⋆Socrates, the number of threads executed, and the amount of work those threads perform, can depend greatly on the order in which threads are executed. The high-level Cilk-2 constructs do not give the user enough control over the execution of his code to write an efficient speculative algorithm, so the chess code continues to use the lower-level Cilk-1 syntax.

Although we focus on ⋆Socrates, chess is not the only speculative algorithm one might want to write in Cilk; there are many others. Any sort of search algorithm where only some solutions are wanted can be naturally cast as a speculative algorithm. For example, consider the protein folding code of Section 2.4. A useful modification to the algorithm would be rather than finding all possible foldings of a polymer, instead finding just one folding that has an energy value less than some threshold.

In this chapter we first describe some proposed extensions that will allow speculative algorithms to be written. We then show how these extensions could be used to implement ⋆Socrates.

This chapter represents ongoing joint work by the members of the Cilk team: Robert Blumofe, Feng Ming Dong, Matteo Frigo, Bradley Kuszmaul, Charles Leiserson, Richard Tauriello, Keith Randall, and myself. Feng Ming Dong has modified the Cilk-to-C preprocessor to accept the Cilk-4 language and Richard Tauriello has begun implementation of the runtime system changes needed for Cilk-4.

7.1 The Cilk-4 Language

We have proposed two extensions to the Cilk-2 language that will enable us to express ⋆Socrates as well as other speculative algorithms without resorting to the lower-level Cilk-1 syntax. There are two extensions that are needed. The first extension is to allow the user to specify a restricted piece of code, called an *inlet* [CSS+91], that is to be executed as soon as a spawned child returns. In chess, an inlet can be used to check the result of a test of a position and perform some action based on that result. The second extension is to allow the user to abort all the children spawned by a procedure. In a speculative program, computations are spawned off whose results may not be needed. When it is determined that certain results are not needed, the computation computing those results should be aborted. Currently, the chess program contains user level code for aborting children. This code was fairly difficult to implement correctly, required detailed knowledge of the runtime system, and it worked only with the Cilk-1 syntax. Adding an abort primitive to the runtime system greatly simplifies the writing of speculative computations.

The Cilk-4 language allows the user to specify inlets to receive results of child procedures. We want inlets to run shortly after the child completes and incorporate returned results into the parent computation. Therefore inlets must be able to read and write variables local to the parent procedure. In order to be able to name the variables in the parent procedure, the user defines inlets within the parent procedure. Since the parent procedure, as well as the inlets, may read and write the same variables, the system guarantees that inlets from the same parent procedure do not execute concurrently with each other or with the parent procedure. Inlets can contain arbitrary Cilk code with just one restriction, namely that inlets are not allowed to execute a **sync**. Inlets can spawn off additional procedures, however. These spawns are treated just like spawns by the parent procedure itself: The parent procedure does not proceed beyond a **sync** while any of these spawns are outstanding.

The syntax of spawning an inlet is:

inlet_spawn I (*args* ...);

where I is the name of an inlet and each argument can be either a standard argument, or a spawn expression. When a spawn expression is used as the ith argument of an inlet, the spawned child is spawned off in the usual manner. The type of the inlet's ith argument must match the type of the value returned by the spawned child. When the spawned child completes, its return value is passed to the inlet as its ith argument. Once an inlet has all its arguments the inlet code can be executed.

Inlets can be implemented without any major changes to our runtime system. Each time a procedure P spawns an inlet I, which receives the result of zero or more child procedures, a new closure is created for the inlet thread I. This inlet thread is spliced into the computation dag between P and the children. The children send their results to I rather than the parent P. This inlet thread I is treated as a high-priority thread, so that once the result from all the children arrive, the inlet I executes shortly thereafter. The result of the inlet thread is then sent to P, where it is treated the same as any arriving argument.

Ensuring that two inlets from the same parent procedure cannot execute concurrently is not difficult to do. Since inlets can access the variables in the frame of a procedure, the system is constrained to execute an inlet on the same processor that contains the frame of the parent procedure. Since frames are never moved when there are outstanding spawns, all inlets are therefore executed on the same processor that executed their parent procedure. This property allows us to easily make the guarantee that no two inlets can execute concurrently.

When implementing inlets we must also be careful not to break the performance guarantees provided by our scheduler. The subtlety occurs in regards to the provably good steals described in Section 3.3. Remember that when a stolen thread sends a value that enables a second thread, the enabled thread is posted to the ready queue of the sending processor, not to the processor on which it originally resided. This policy is necessary for the scheduler to be provably good. What happens when a stolen thread on processor P sends a value which enables an inlet on processor Q? According to the above policy, the inlet must be stolen and executed on processor Q. But as we have seen in the preceding paragraph, inlets cannot be stolen. To solve this problem we treat the inlet as if it were executing on processor Q. Therefore any computation enabled by the execution of the inlet must be migrated to processor Q. Computations which may be enabled by the inlet include any procedures that the inlet spawns, as well as the parent procedure itself if the value being returned is the last value the parent is waiting for. By migrating these computations to the sending processor, and by having the sending processor not begin work-stealing until it finds out if any computations will be migrated, we maintain the performance guarantee.

The change to the Cilk-4 language for the second extension is to provide two new primitives: **abort_and_continue()** and **abort_and_return()**. When one of these primitives is called, either from within a procedure P or, more commonly, from within an inlet created by procedure P, the system terminates any outstanding children that procedure P may have. This implies terminating not just the children of P, but also all descendants of those children. Just how "graceful" this termination will be is still to be determined. Currently we expect that we will not halt any executing threads, but will simply prevent new threads from beginning. We also considered allowing the user to specify a piece of code, similar to an inlet, that would be executed when a procedure is aborted. This would allow the user to "clean up", perhaps, for example, releasing some piece of storage the procedure had allocated. We decided not to implement this option, since we saw no immediate needed for it, but we may decide to add it if a need arises. The two new primitives differ in where execution continues after the abort completes. When the **abort_and_continue()** primitive is called, the procedure continues at the next **sync** statement once all children have been aborted, while when the **abort_and_return()** primitive is called, the procedure returns once the abort completes.

The abort mechanism will be implemented similarly to the abort mechanism in the chess code (as described in Section 4.3.2). To implement aborts, we will augment the runtime system to keep track of the status of all spawned children. The status must contain enough information to find all children, even those that have been stolen by another processor. In essence, this information creates a tree of all existing procedures. When an abort occurs inside procedure P we can walk the subtree rooted at P to find all descendants of P. As we walk this tree we set a flag in each procedure indicating that the procedure is to be aborted. At the beginning of each procedure, the preprocessor adds a check of the abort flag, and if this flag is set the procedure returns without executing the user's code. Similar checks will be performed each time a procedure restarts after a **sync**. Since a spawned, but aborted, procedure still executes, the dag of the aborted computation is cleaned up automatically. This method of aborting also allows the runtime system to wait until the abort is complete before returning from the abort primitive.

One unanswered question about the implementation of aborts is how much overhead will the abort mechanism add to the execution of a program. Inlets have the nice property that they cause no overhead except where they are used, but this property is not true of aborts. Even if a procedure P does not itself perform any aborts, one of P's ancestors could do an abort, in which case P and all its descendants need to be aborted. Therefore, the system must keep track of the information needed to do an abort for all procedures. Although we do not expect the overhead for keeping this information to be large, this overhead seems wasteful, especially since most programs do not use aborts. To eliminate this overhead, the current proposal is to to have a compile-time flag that informs the preprocessor whether abort information should be kept for all procedures, or none of them. In this way programs that do not use the abort mechanism will not pay any overhead.

As mentioned earlier, the modifications discussed here have not yet been implemented. The current status is that most of the design decisions have been made, we have decided how to implement the changes, and implementation has begun. Although we have settled on a syntax, the syntax always seems open to change, so the final syntax will likely differ from what is presented here. However, even if the details of the syntax change, I do not expect the power and expressibility of the final version to differ significantly from the system presented here. Currently Feng Ming Dong is working on modifying the type-checking preprocessor to support the new syntax, and Richard Tauriello is making the modifications to the runtime system to support inlets and aborts. Richard Tauriello will report on the runtime system modifications in his masters thesis.

7.2 A Cilk-4 Example: Chess

We now have the primitives needed to write the chess code in Cilk-4, but before examining this code, let us first briefly review the requirements of the search algorithm used in ⋆Socrates. The inputs to the search algorithm includes a chess position, a depth d, and a range of interest specified by the bounds α and β. If the exact value v of the position when searched to depth d is in the range (α, β), then the exact value of the position should be returned. But if $v < \alpha$, then the search need only return some $v' \leq \alpha$ where v' is an upper bound on v. Similarly, if $v > \beta$, then the search need only return some $v' \geq \beta$ where v' is a lower bound on v. In the case where $\alpha + 1 = \beta$ the search reduces to a test of whether $v \geq \beta$ or $v < \beta$. We call a search a *test search* if $\alpha + 1 = \beta$, and otherwise, it is called a *full value search*. Figure 7-1 replicates the Cilk-1 dag from Figure 4-7 which describes the control flow of a full value search.

The search code is broken in two parts. Figure 7-2 shows the Cilk-4 code for a test search, and Figure 7-3 shows the code for a full value search. In this code we focus on the control flow needed to perform a search and ignore details that do not effect the control flow. In the code shown, each search routine is passed two items: (1) a state structure which completely describes the current position and the search to be done, and (2) a move specifying which move is to be applied to the current position. The code performs the search and returns a score for the new position.

The code for the test search is the simpler of the two. Initially, we define the inlet `check_test_result`, which is described later. Then, we begin by applying the move to the state structure and if we have searched deep enough we immediately return a value for the position. Otherwise, we determine if a null-move search should be performed, and if so, we spawn it off and wait for the result. We then check the returned score, and if the score is greater than β, we return immediately. Returning with a score greater than β is called *failing high*. Otherwise, we search the first child, wait for it to complete, and again check to see if we can fail high. Up to this point the search code can be written entirely in Cilk-2 style.

It is the next part of the code where the new constructs are used. Typically, if a search is going to fail high, it will do so either in the null-move search or during the search of the first child. Therefore, after doing these two searches, it is reasonable to do all the

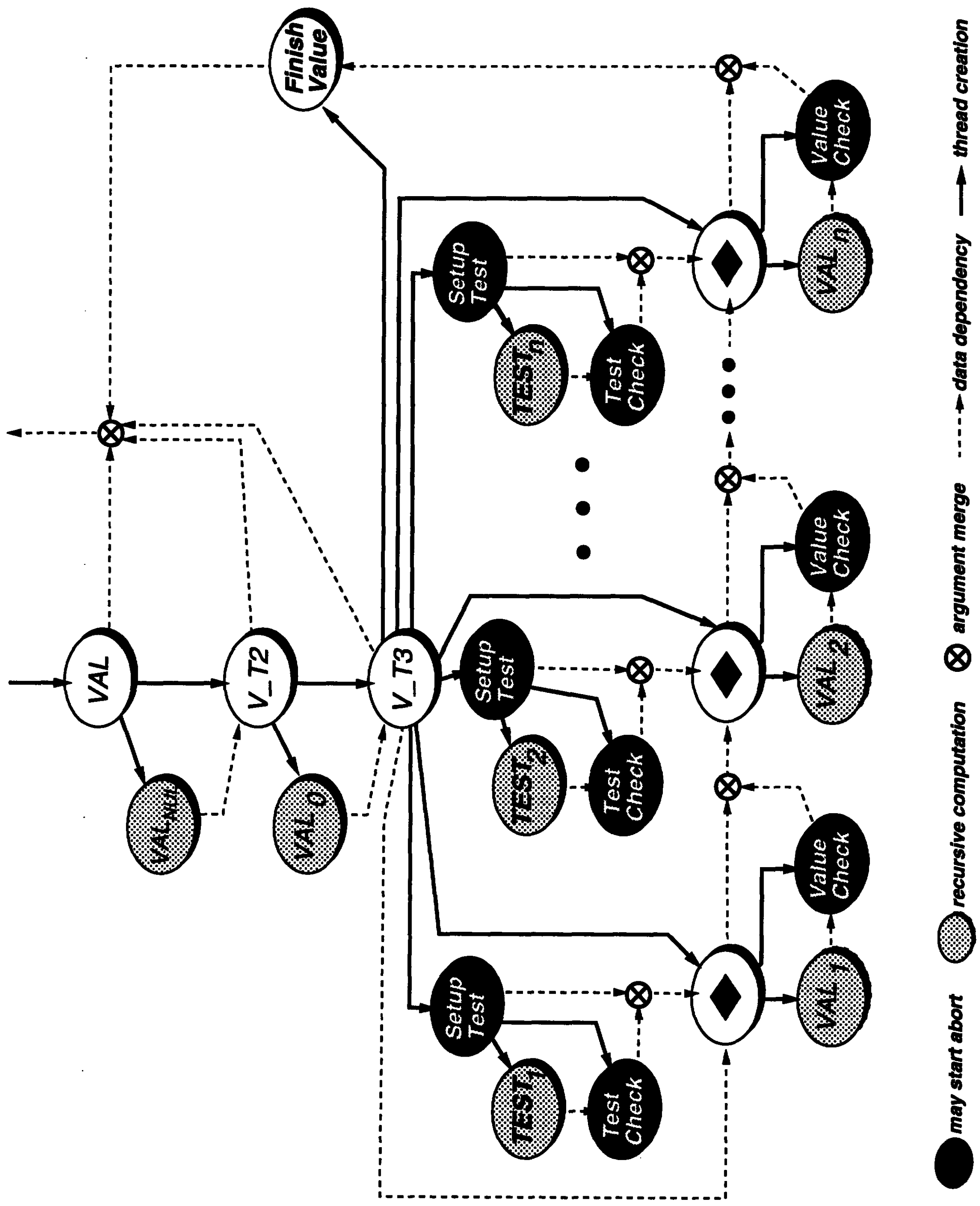

Figure 7-1: This dag shows the dag created by ⋆Socrates when performing a value search. The circles labeled $TEST_i$ and VAL_i are recursive calls of the search algorithm. Other circles correspond to Cilk threads.

```
cilk int test(STATE s, MOVE move){
  int child_score;              /* result from search of a child */
  int bestscore;                /* best score found so far */

  /* define the inlet to be run when a test completes */
  inlet void check_test_result(int score){
    if (score>=s.beta){
      abort_and_return(score);
    }
    bestscore = MAX(bestscore, score);
  }

  apply_move(s,move);           /* make the move */

  if(s.depth==0) {              /* If we reached the bottom of the tree */
    return(evaluate(s));        /* compute and return the score */
  }


  /* Try a null move search. */
  if (try_null_move_search_p(s)){
    child_score = spawn test(s, NUL_MOVE);
    sync;
    /* If score from null move beats beta we are done. */
    if (child_score>=s.beta)  return(child_score);
  }

  generate_moves(&move_list);    /* Generate the moves to be tried */

  /* Search the first child. */
  child_score = spawn test(s,move_list[0]);
  sync;
  /* If score from first move beats beta we are done. */
  if (child_score>=s.beta)   return(child_score);
  bestscore = MAX(bestscore, child_score);

  /* spawn a test for each remaining move */
  for(j=1;j<s.num_children;j++)
    inlet_call check_test_result(spawn test(s,move_list[j]));
  sync;
  return(bestscore);
}
```

Figure 7-2: Cilk-4 code for test search.

remaining searches in parallel. If any one of these does fail high, we can return without completing any remaining searches. In order to return immediately, inlets and aborts are needed. We define a simple inlet, called **`check_test_result`**, to receive the result of the search of a child. This inlet checks to see if the child's score beats β, and if so, it calls the new primitive **`abort_and_return()`**, which kills off any computations spawned by this search procedure and returns. This inlet also updates **`bestscore`**, which keeps track of the best score found so far. The remainder of the code for a test search spawns off searches of the rest of the possible moves, while specifying that the result of each search should be passed to a **`check_test_result`** inlet. After spawning all the children the code performs a **`sync`**. If the **`sync`** is reached, then no test failed high, so we simply return **`bestscore`**, the value of the best move that we found.

Figure 7-3, which is split into two parts, shows the code for a full value search. The code begins by defining the needed inlet, which will be described later. After defining the inlet, the rest of the code, which begins on the second page of Figure 7-3, is very similar to the code for a test search. Initially it is identical: we return if the position has been searched deep enough, if a null move search fails high, or if the search of the first child fails high. At this point the Jamboree search algorithm requires us to perform test searches on all the remaining moves in parallel to see if they could possibly beat the best score. If the test of any move fails high, then we need to do a full value search for that move. These full value searches should be done one at a time and in order. To perform the search in this way, the code spawns off all the tests in parallel, just as in the code for a test search. The difference is in the inlet that is run when the tests return. This inlet needs to make sure the full value searches are spawned off only if needed and only in serial order.

In order to control the spawning of the value searches, we store information about the status of the child searches in the parent's frame. We create an array **`status[i]`**, whose ith element gives the search status for the ith move. Initially, all the entries are set to `DOING_TEST` indicating that a test search has been spawned off. Other possible states are `DO_VAL`, which indicates that a full value search is needed, `DOING_VAL`, which indicates that a full value search has been spawned off, and `DONE`, which indicates that no more searches are needed for this move. As an optimization, we also keep track of a variable **`next_child`**, which indicates the next move for which a full value search could be started. A move can be the next to be fully searched only if for all of the earlier moves, the full value searches of those moves have either been completed, or were not necessary at all.

To implement the Jamboree search algorithm, the code uses the inlet **`child_done`** which receives the result of the search of a child. The same inlet is used to receive results from test searches and from value searches.

This inlet first checks to see if the returned result beat β. If so, the search should fail high, and the **`abort_and_return()`** primitive is used to end the search immediately. Otherwise, the inlet continues. The inlet next updates the status information for this child. When a test search has just completed, performing this update requires checking to see if the returned result was greater than α. If so, a value search is needed and so the status is set to `DO_VAL`. Otherwise no value search is needed and so the status is set to `DONE`. When a value search has just completed, no further searching is need for this child so the status of the child is set to `DONE`. The last action of the inlet is to spawn off a value search if appropriate. This action is performed by walking through the moves, beginning with **`next_child`**, until either a value search is spawned off, or until we discover no value search should be started. If the status of the move being considered is `DOING_TEST`, then no value search is spawned off, because we must wait until the test of that move completes. If the status of the move being considered is `DO_VAL`, then we spawn off the value search for that move, specifying, of course, **`child_done`** as the inlet to be run when the search completes. If the status of the move being considered is `DOING_VAL`, then no value search is spawned off, because there is already a value search in progress. Otherwise the status of the move being considered is `DONE`, so no value search is needed for it. In this case we set **`next_child`** to be the next move and loop.

There is one detail that may affect performance that was handled more efficiently in the Cilk-1 version than in the Cilk-4 version shown above. When we perform a test search, this version tests against the value of α that the parent had when the search was spawned. Occasionally, between the time the test search was spawned, and the time execution of that test begins, the parent's value of α may increase. The search algorithm is more efficient if we test against the parent's current value of α rather than the earlier value. Since the Cilk-1 version passes around pointers to state structures, test searches in Cilk-1 have access to the parent's current version of α, and the current version is in fact used by the test search. The Cilk-4 version shown above does not use the latest version of α because it has no access to that value. If we modified the above code to store the state structures in shared memory,

```
cilk int value(STATE s, MOVE move){
/* Remember that only one value search should be in progress at a time,
 * and that all value searches must be done in order.
 * The variables:
 * status[i] tracks the status of move 'i'.  It is one of:
 *    DOING_TEST -- initial test not complete
 *    DO_VAL     -- full value search needed, not yet started
 *    DOING_VAL  -- full value search in progress
 *    DONE       -- all testing of this child complete
 * next_child:  the next child for which a value search could be begun.
 */
  int status[MAX_NUM_MOVES], next_child;
  int bestscore;

  inlet void child_done(int child_score, int child_index, int srch_type){
    if (child_score>=s.beta) abort_and_return(child_score);
    bestscore = MAX(bestscore,child_score);
    s.alpha = MAX(child_score,s.alpha);

    if (srch_type==TEST_SRCH){
      /* A test search completed: Set status based on test result. */
      if (child_score>test_alpha)
        status[child_index]=DO_VAL;
      else
        status[child_index]=DONE;
    } else{
      /* A value search completed:  This child is finished. */
      status[child_index]=DONE;
    }

    /* See if we need to start a value search */
    while(;next_child<s.num_children;next_child++){
      if (status[next_child]==DOING_TEST) break;
      else if (status[next_child]==DOING_VAL) break;
      else if (status[next_child]==DO_VAL){
        inlet_call child_done(spawn value(s, move_list[next_child]),
                              next_child, FALSE);
        break;
      }
    }
  }

  /*** ... continued on next page ... ***/
```

Figure 7-3: Cilk-4 code for value search.

```
  /*** Full Value Search Continued ***/

  apply_move(s,move);            /* make the move */

  if(s.depth==0) {               /* If we reached the bottom of the tree */
   bestscore=evaluate(s);        /* compute and return the score */
   return(s);
  }

  /* Try a null move search. */
  if (try_null_move_search_p(s)){
    child_score = spawn value(s, NUL_MOVE);
    sync;
    /* If score from null move beats beta we are done. */
    if (child_score>=s.beta)  return(child_score);
  }

  generate_moves(&move_list);    /* Generate the moves to be tried */
  /* Search the first child. */
  child_score = spawn value(s,move_list[0]);
  sync;
  /* If score from first move beats beta we are done. */
  if (child_score>=s.beta)   return(child_score);
  s.alpha = MAX(s.alpha,child_score);
  bestscore = MAX(bestscore,child_score);

  next_child=1;
  for(j=1;j<s.num_children;j++)status[j]=DOING_TEST;

  /* Spawn off the tests */
  test_alpha = s.alpha;
  for (i=1;i<s.num_children;i++)
    inlet_call child_done(spawn test(s,move_list[i]), i, TEST_SRCH);
  sync;

  /* We reach here only if no child beat beta */
  return(bestscore);
}
```

Figure 7-3 continued: Cilk-4 code for value search and then passed around pointers to the state structure, then the Cilk-4 version could use the latest value of α as well.

7.3 Conclusions

Using the Cilk-4 primitives we are now able to implement the Jamboree search algorithm in under three pages of code. This is much simpler than the two dozen threads which are needed to implement the search code in the lower level Cilk-1 syntax. It will be interesting to see how the performance of the Cilk-4 implementation compares with the performance of the Cilk-1 implementation. We expect that the performance of the two will be similar. However, even though the Cilk-4 version is written at a higher level, it is possible it will perform better. The Cilk-1 version used low-level features, such as nonstealable threads, that interfere with the operation of the provably-good scheduler by causing the busy-leaves property not to hold. By eliminating the use of these features, the Cilk-4 version may actually improve the efficiency of the search.

The additions described in this chapter are useful for more than just chess. One natural application of Cilk-4 is for use with backtrack searches where only one solution is needed. In Cilk-2 it is easy to write a backtrack search routine that finds all possible solutions to a problem. But often only one solution is needed, and Cilk-2 style codes have no way to stop the search once the first good solution is found. With the Cilk-4 additions, it is easy to modify a search program to stop after finding one solution: just associate with each spawn an inlet that performs an `abort_and_return` if the spawned child has found a solution.

Chapter 8

Conclusions

This chapter summarizes some of the features of the Cilk system, and then describes some areas for future work.

8.1 Summary

We think the Cilk system has achieved its goal of allowing a programmer to easily and efficiently implement a wide range of asynchronous, dynamic, parallel algorithms. At the beginning of this document, we listed a number of characteristics that a good parallel programming system should have. This section revisits this list and examines how Cilk stands up.

- **Minimize the gap between applications and languages:** The Cilk system allows a programmer to focus on his application, not on the low-level protocols needed to implement a parallel algorithm. The Cilk system minimizes this gap by raising the level of programming. The Cilk system hides from the programmer most low-level details such as thread encapsulation, thread scheduling, and load balancing. The Cilk-2 system goes further and hides the continuation-passing nature of the runtime system from the user, allowing the user to write code in a style similar to serial code. Cilk-4 further minimizes this gap by allowing the user to easily implement efficient speculative computations.

- **Provide predictable performance:** In Chapter 3 we showed that with P processors, the expected execution time of a Cilk computation, including scheduling overhead, is bounded by $T_P = O(T_1/P + T_\infty)$. With this knowledge a programmer is able to predict the performance of his program before it is even executed by estimating T_1 and T_∞. Alternatively, a programmer can run his program once and use the reported values of T_1 and T_∞ to accurately predict how his program will perform on other machine sizes.

- **Execute efficiently:** There is not a large overhead for executing a program with the Cilk system. We have shown empirically (Section 3.4) that for most applications the execution time of a Cilk program on one processor is comparable to the execution time of a serial code when run on the same processor.

- **Scale well:** We have shown empirically (Section 3.5) that we can accurately model the execution time of a program as $T_P \approx T_1/P + c_\infty T_\infty$, where c_∞ is a small value ($c_\infty = 1.5$ for the knary example). Since the constant in front of the T_1/P term is one, we obtain nearly perfect linear speedup when the available parallelism is large compared to the number of processors.

- **Portable:** Cilk has been ported to a wide range of systems. It runs on various serial machines (under Unix and Linux), Symmetric MultiProcessors (e.g. Sun, SGI), and Massively Parallel Processors (e.g. CM-5, Paragon). In addition, Blumofe has implemented a version of Cilk which runs on Networks of Workstations [Blu95].

- **Leverage existing codes:** Since Cilk can call standard C functions, much of the existing serial code can often be used when porting an application to Cilk. This was especially important in porting the Socrates chess program and the POV-Ray ray tracer to Cilk. Both of these are large applications, and when they were ported to Cilk most of the code for these applications was able to be reused without modification.

- **Be expressive:** Although there are many applications that cannot be easily expressed in Cilk, there are a wide range of applications which can. And many of the applications that can be expressed in Cilk cannot be easily expressed in other parallel languages. The $\star$Socrates program is an example of one such complex program. Although this program was time consuming to write and debug in Cilk-1, the Cilk-4 additions described in Chapter 7 help make this program easier to express in Cilk. In addition, the shared memory system described in Chapter 6 significantly increases the expressibility of the language by allowing large amounts of data to be easily shared throughout the computation.

8.2 Future Work

The Cilk system described in this thesis is quite useful, as it allows a wide range of programs to be easily expressed, while still achieving good performance. But, as we said earlier, the story of Cilk is one of incremental improvement. There are still areas we think we can improve, and so the story of Cilk is not yet over. We conclude this thesis by describing some of the improvements to the system that we have considered.

One improvement we would like to make is to build a shared memory system which does not destroy the performance guarantees of Chapter 3. With the current shared memory system we are able to bound the number of page faults a program makes, but we have been unable to provide a tight theoretical bound on the execution time of a shared memory Cilk program. Experiments are currently under way with a new, and simpler, implementation of dag-consistent shared memory, about which we hope to be able to prove tighter bounds. This dag-consistent shared memory implementation, does not perform the tree-walk operation described in Chapter 6. Instead a kernel always go to the backing store to access a page. This change makes theoretical analysis easier since it eliminates the need to analyze the execution time of performing the tree walk. In practice, on machines such as the CM-5, where sending a short message is inexpensive, we expect the performance for this new system to be similar to the performance of the tree walk algorithm. For machines where the message overhead is larger, we expect this implementation to be more efficient since eliminating the tree walk reduces the number of protocol messages that are needed.

Another improvement we would like to make is to allow certain shared memory applications to run more efficiently by reducing the amount of data movement necessary. Shared memory lets us move data to the computation, but, as is well known, moving the computation to the data is often more efficient. Currently the Cilk system has no way of moving the computation to the data, so a shared memory Cilk program often needs to perform more data movement than other systems need to perform. As an example, consider performing many iterations of array relaxation. In a data parallel program the array would be distributed across the machine, with each processor having its own section, and on each iteration only the elements at the edge of a processors section would need to be communicated. In a naive Cilk program, on each iteration a new random tree is built to spread the the computation among the processors. Each processor would typically get a different portion of the array each iteration. The entire array would be communicated twice per cycle: first from the backing store to the processor performing the computation, and then back again to the backing store.

We have considered two methods by which we could more closely associate a computation to its data, thereby reducing the amount of communication a shared memory program requires.

The simpler of the two methods is based on augmenting the existing shared memory system to try to increase reuse of data. The idea is to try to keep track of how the computation was spread out on the previous iteration, and, where possible, try to repeat that computation tree on the next iteration, so that a processor would tend to work on the same data from iteration to iteration. To implement this we would have to figure out how to regrow the computation tree the same way from iteration to iteration. Presumably the user would specify when the system should try to do this. Also, the current shared memory system cannot take advantage of potential data reuse between iterations: When a new kernel begins (which happens every iteration) a processor will not use any of the data currently in its cache because it does not know if that data is up-to-date. We would need to modify the shared-memory system so that when an old copy of a needed page is in a processor's cache, the processor is able to check to see if the page is current before it requests a new copy. To take full advantage of this method we would also need to modify the system so that updated pages do not need to be sent to the backing store after every iteration. Instead, we would want to tell the backing store where the updated page is, and leave the updated page in the cache, even after the kernel that updated the page finishes.

A second method to decrease communication in programs using lots of data is to use what we call *persistent closures*. The idea here is to try to mimic the way that the data-parallel model works. We would bind to a processor a thread which performs part of the computation, and allow that thread to be executed repeatedly. For the array relaxation example, we would bind to each processor a thread to perform the relaxation on part of the array, and that thread would be executed once each iteration. The portion of the array used by that thread would be bound to the processor as well, and we provide the user with a way to specify what data needs to be communicated on each iteration. We have performed some simple experiments using this idea and have seen some promising results. We implemented an array relaxation example using low-level Cilk-1 features, and this program out-performed a similar array relaxation program we wrote in a data parallel language. Although we can implement this mechanism using low-level Cilk features, we do not now how to add such a mechanism to Cilk at a reasonably high level. Also we have no idea how to integrate such a feature into Cilk in a way that gives us any performance guarantees.

We have no current plants to add either of these methods to Cilk since we have some questions about the implementation of each of them. We do expect to eventually address the issue of reducing data movement in Cilk, but whether it will be via one of these two methods, or via something completely different, is not yet known.

A final improvement that we are considering is to implement a stack-based execution model using lazy task creation [MKH91]. This modification is one we think we understand, and we expect to implement it in the near future. Switching Cilk to a stack-based execution model would provide two benefits. First, it would lower the overhead of spawning new tasks. Second, under a stack-based model, the execution order of Cilk programs would more closely mimic the execution order of serial programs.

A stack-based model differs from the current model in what happens when a spawn is encountered. In the current system when a **spawn** is reached the state needed to execute the spawned child is packaged up and put aside, and the parent procedure immediately continues execution after the **spawn**. This is the opposite of the execution order of function calls in most languages. Typically, when a function call is made the parent function suspends until the called function completes. Under a stack based model when a **spawn** is reached, the execution order mimics the execution order of a serial program by suspending the parent and beginning execution of the child. However, enough state about the parent is kept around so that if a steal request arrives the parent can be packaged up and stolen.

This stack-based technique makes the overhead of a **spawn** comparable to the overhead of a procedure call. The only difference is that when spawning, some extra information may be kept around so that the parent is able to be stolen. When a parent is stolen, then the stack-based model has the additional cost of packaging up the parent. This cost should be similar to, and probably slightly greater than, the cost of performing a spawn in the current system. Since steals are rare compared to **spawns**, this technique significantly reduces the overhead of a Cilk program.

8.3 Concluding Remarks

This thesis began by pointing out that recently parallel hardware has been advancing faster than parallel software. Parallel machines are becoming commonplace, but they are typically used for executing many independent jobs, since writing a true parallel program is still a difficult task. Cilk alone is not the solution to the "parallel software problem." Probably no one system is. But Cilk has the potential to become part of the solution, and to help spread the use of parallel programming. By allowing a programmer to easily implement efficient, asynchronous, parallel algorithms, Cilk can be become one more entry in a programmer's arsenal of tools for attacking parallel programming problems.

Appendix A

Protein Folding Optimizations

This appendix describes the algorithmic optimizations that we made to the original protein folding code. These changes provided a speedup of 1 to 2 orders of magnitude on various problem sizes. This appendix assumes the reader is familiar with the application as described in Section 2.4.

In Section 2.4 we described the changes made to the protein folding code to express it in Cilk. In addition to these changes, we made other changes to the original serial code which significantly improved the performance of the protein folding application. These changes make use of simple checks to see if we can end the search down a branch early. When searching through the cube creating a partial path, it is easy to create a partial path from which no Hamiltonian path can be created. As a simple example, let us consider paths on a two dimensional grid. A short path which includes the three points nearest the corner, but not the corner itself, can never be extended into a Hamiltonian path. But since much of the grid remains unvisited, a naive algorithm would perform a significant amount of searching before giving up on this partial path. By adding some simple checks to the search algorithm we can reduce much of this wasted search. We do this as follows: For each point in the cube we keep track of how many unvisited neighbors it has. A point with no unvisited neighbors, such as the corner point in the above example, can never be reached. Since a Hamiltonian path must reach every path once, if a point with no unvisited neighbors exists, then no Hamiltonian path is possible. Path (a) of Figure A-1 shows a 2-D grid in which point 0 has no unvisited neighbors. If at some point in the search a point with no unvisited neighbors is created, then there is no way to produce a Hamiltonian path from the current partial path, so the search of the current partial path can be ended. Also, notice that if a point with only one unvisited neighbor exists, then no path can be created that continues through that point. Therefore any possible Hamiltonian paths must end at that point. Therefore, if two such points ever exist then no Hamiltonian path is possible. Path (b) of Figure A-1 shows an example where two points, namely 0 and 2, both have one unvisited neighbor, and so no path is possible. So at each step the algorithm checks to see if a two such points have been created, and if so the search of the current partial path is ended. These checks speed up the program by approximately a factor of 10 on the $3 \times 3 \times 3$ cube and 40 on the $4 \times 3 \times 3$ cube. We could not compute the exact speedup for the $4 \times 4 \times 3$ cube because it would take too long to run this size without the improvements. However, we estimate that the improvements provide a speedup of over 100 on this problem size.

A further improvement can be made by noting that the set of points at which a Hamiltonian path can end is partially determined by the point at which the path begins. To take advantage of this we give each point a *parity*. A point at position (i, j, k) is given a parity $(i + j + k)$ mod 2. Note that each time a point is added to the end of the partial path, the parity of the point at the end of the path changes. This change occurs because exactly one of the indices of the new end point differs by exactly 1 from the previous end point. Therefore if we know the starting point of a path, and we know how long the path is, we can compute what the parity of the endpoint is. So for any partial path, since we know the parity of its starting point, we can easily compute what the parity must be for an endpoint of any Hamiltonian path beginning with that partial path. We have seen that when the search creates a point with only one unvisited neighbor, that point must be the end of any Hamiltonian path. As described earlier, our search code detects when a point with only one unvisited neighbor is created. When we create such a point we also check to see if its parity is the predicted parity for the endpoint. If not we stop the search of the current partial path. Path (c) of Figure A-1 shows an example of this check for a 5×5 2-D grid. In this figure each point is labeled with its parity. A Hamiltonian path is of an odd length (25), so it must begin and end on the same parity, in this case odd. In this example, point 6 must be the final point, but since it has even parity no Hamiltonian path is possible.

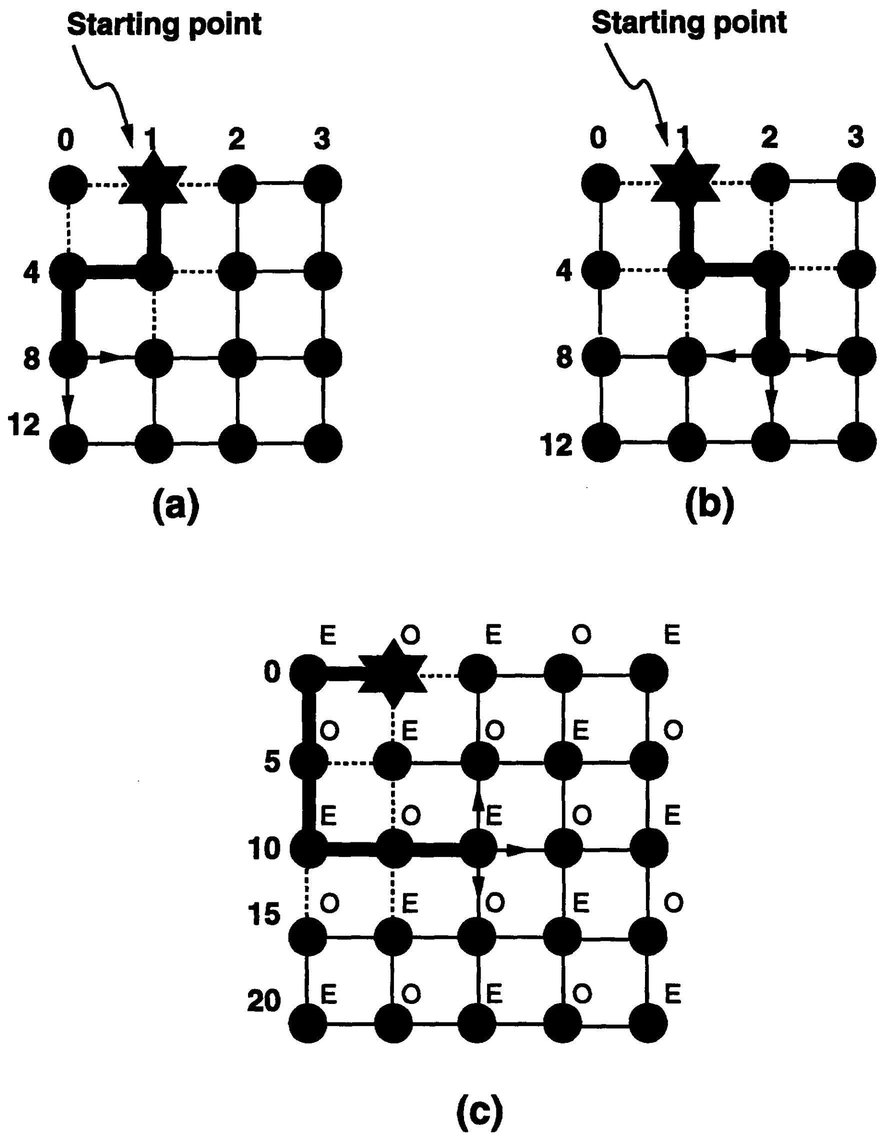

Figure A-1: Each figure shows a partial path on a 2-D lattice. None of these partial paths can result in a Hamiltonian path: Path (a) because it has a point (0) with no unvisited neighbors, and Path (b) because it has two points (0,2) with only one unvisited neighbor, and Path (c) because a Hamiltonian path would have to end at a point (6) with the wrong parity.

Bibliography

[AAC$^+$92] Gail Alverson, Robert Alverson, David Callahan, Brian Koblenz, Allan Porterfield, and Burton Smith. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, pages 188–197, Washington, D.C., July 1992.

[ABLL91] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 95–109, Pacific Grove, California, October 1991.

[ACP95] Thomas E. Anderson, David E. Culler, and David A. Patterson. A case for NOW (networks of workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[BB94] Eric A. Brewer and Robert Blumofe. Strata: A multi-layer communications library. In *Proceedings of the 1994 MIT Student Workshop on Scalable Computing*, July 1994.

[BBB$^+$94] D. Bailey, E. Barszcz, J. Barton, D. Browning, et al. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[BBZ88] 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.

[BCK$^+$89] M. Berry, D. Chen, P. Koss, D. Kuck, et al. The Perfect club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, 1989.

[BE89] Hans Berliner and Carl Ebeling. Pattern knowledge and search: The SUPREM architecture. *Artificial Intelligence*, 38(2):161–198, March 1989.

[Bea95] D. Beal. Round-by-round. *ICCA Journal*, 18(2), 1995.

[BFJ$^+$95] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Rob Miller, Keith H. Randall, and Yuli Zhou. *Cilk 2.0 Reference Manual.* MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, Massachusetts 02139, June 1995. Available via `ftp://theory.lcs.mit.edu/pub/cilk/manual2.0.ps.Z`.

[BFJ$^+$96] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. Dag-consistent distributed shared memory. In *Proceedings of the 10th International Parallel Processing Symposium*, Honolulu, Hawaii, April 1996.

[BH86] J. E. Barnes and P. Hut. A hierarchical $O(N \log N)$ force calculation algorithm. *Nature*, 324:446, 1986.

[BJK$^+$95] 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.

[BL94] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, Santa Fe, New Mexico, November 1994.

[Ble92] Guy E. Blelloch. Programming parallel algorithms. In *Proceedings of the 1992 Dartmouth Institute for Advanced Graduate Studies (DAGS) Symposium on Parallel Computation*, pages 11–18, Hanover, New Hampshire, June 1992.

[Ble93] Guy E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-93-129, School of Computer Science, Carnegie-Mellon University, April 1993.

[Blu95] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995.

[BP94] Robert D. Blumofe and David S. Park. Scheduling large-scale parallel computations on networks of workstations. In *Proceedings of the Third International Symposium on High Performance Distributed Computing*, pages 96–105, San Francisco, California, August 1994.

[Bre74] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, April 1974.

[BS81] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 187–194, Portsmouth, New Hampshire, October 1981.

[BZS93] 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.

[CA94] David Chaiken and Anant Agarwal. Software-extended coherent shared memory: Performance and cost. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 314–324, Chicago, Illinois, April 1994.

[CAL$^+$89] 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*, pages 147–158, Litchfield Park, Arizona, December 1989.

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 152–164, Pacific Grove, California, October 1991.

[CD88] Eric C. Cooper and Richard P. Draves. C threads. Technical Report CMU-CS-88-154, School of Computer Science, Carnegie-Mellon University, June 1988.

[CDG+93] 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.

[CGH94] Rohit Chandra, Anoop Gupta, and John L. Hennessy. COOL: An object-based language for parallel programming. *IEEE Computer*, 27(8):13–26, August 1994.

[CRRH93] Martin C. Carlisle, Anne Rogers, John H. Reppy, and Laurie J. Hendren. Early experiences with Olden. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, August 1993.

[CSS+91] 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*, pages 164–175, Santa Clara, California, April 1991.

[DMBS79] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK Users' Guide.* Siam, Philadelphia, 1979.

[DSB86] 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.

[EAL93] Dawson R. Engler, Gregory R. Andrews, and David K. Lowenthal. Filaments: Efficient support for fine-grain parallelism. Technical Report TR 93-13a, The University of Arizona, 1993.

[EL94] Natalie Engler and David Linthicum. Not just a PC on steroids. *Open Computing*, pages 43–47, April 1994.

[FF82] Raphael A. Finkel and John P. Fishburn. Parallelism in alpha-beta search. *Artificial Intellgence*, 19(1):89–106, September 1982.

[FLA94] 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.

[FM87] Raphael Finkel and Udi Manber. DIB—a distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, April 1987.

[FMM93] R. Feldmann, P. Mysliwietz, and B. Monien. Game tree search on a massively parallel system. In *Advances in Computer Chess 7*, pages 203–219, 1993.

[FMM94] Rainer Feldmann, Peter Mysliwietz, and Burkhard Monien. Studying overheads in massively parallel min/max-tree evaluation. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 94–103, Cape May, New Jersey, June 1994.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability.* W.H. Freeman and Company, 1979.

[GLL$^+$90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, Seattle, Washington, June 1990.

[Gra66] R. L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 45:1563–1581, November 1966.

[Gra69] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[GS93] Guang R. Gao and Vivek Sarkar. Location consistency: Stepping beyond the barriers of memory coherence and serializability. Technical Report 78, McGill University, School of Computer Science, Advanced Compilers, Architectures, and Parallel Systems (ACAPS) Laboratory, December 1993.

[Gwe94] Linley Gwennap. Intel extends 486, Pentium families. *Microprocessor Report*, 8(3):1–11, March 1994.

[Hal84] Robert H. Halstead, Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 9–17, Austin, Texas, August 1984.

[Hal85] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[HD68] E. A. Hauck and B. A. Dent. Burroughs' B6500/B7500 stack mechanism. *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 245–251, 1968.

[HKT93] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Preliminary experiences with the Fortran D compiler. In *Supercomputing '93*, pages 338–349, Portland, Oregon, November 1993.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: a Quantitative Approach.* Morgan Kaufmann, San Mateo, CA, 1990.

[HS86] W. Hillis and G. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[HWW93] Wilson C. Hsieh, Paul Wang, and William E. Weihl. Computation migration: Enhancing locality for distributed-memory parallel systems. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 239–248, San Diego, California, May 1993.

[HZJ94] Michael Halbherr, Yuli Zhou, and Chris F. Joerg. MIMD-style parallel programming with continuation-passing threads. In *Proceedings of the 2nd Inter-*

national Workshop on Massive Parallelism: Hardware, Software, and Applications, Capri, Italy, September 1994. A longer version appeared as : MIT Laboratory for Computer Science, Computation Structures Group Memo 355.

[Int94] Intel Supercomputer Systems Division, Beaverton, Oregon. *Paragon User's Guide*, June 1994.

[JD73] Edward G. Coffman Jr. and Peter J. Denning. *Operating Systems Theory.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1973.

[JK94] Chris Joerg and Bradley C. Kuszmaul. Massively parallel chess. In *Proceedings of the Third DIMACS Parallel Implementation Challenge*, Rutgers University, New Jersey, October 1994. Available as `ftp://theory.lcs.mit.edu/pub/cilk/dimacs94.ps.Z`.

[JKW95] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 213–228, Copper Mountain Resort, Colorado, December 1995.

[JP92] Suresh Jagannathan and Jim Philbin. A customizable substrate for concurrent languages. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 55–67, San Francisco, California, June 1992.

[Kal90] L. V. Kalé. The Chare kernel parallel programming system. In *Proceedings of the 1990 International Conference on Parallel Processing, Volume II: Software*, pages 17–25, August 1990.

[KC93] Vijay Karamcheti and Andrew Chien. Concert—efficient runtime support for concurrent object-oriented programming languages on stock hardware. In *Supercomputing '93*, pages 598–607, Portland, Oregon, November 1993.

[KCDZ94] 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.

[KEW+85] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 276–283, 1985.

[KHM89] David A. Kranz, Robert H. Halstead, Jr., and Eric Mohr. Mul-T: A high-performance parallel Lisp. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 81–90, Portland, Oregon, June 1989.

[KM75] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, Winter 1975.

[KOH+94] 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*, pages 302–313, Chicago, Illinois, April 1994.

[KR90] 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.

[Kus94] Bradley C. Kuszmaul. *Synchronized MIMD Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1994. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-645 or `ftp://theory.lcs.mit.edu/pub/bradley/phd.ps.Z`.

[KZ93] Richard M. Karp and Yanjun Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM*, 40(3):765–789, July 1993.

[LAB93] Pangfeng Liu, William Aiello, and Sandeep Bhatt. An atomic model for message-passing. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 154–163, Velen, Germany, June 1993.

[LAD+92] 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*, pages 272–285, San Diego, California, June 1992.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[LRV94] James R. Larus, Brad Richards, and Guhan Viswanathan. LCM: Memory system support for parallel language implementation. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 208–218, San Jose, California, October 1994.

[Mil95] 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.

[MKH91] 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.

[Mos70] 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.

[Mot93] Motorola. *PowerPc 601 User's Manual*, 1993.

[MR87] Piyush Mehrotra and Jon Van Rosendale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, 5:339–361, 1987.

[MSA$^+$85] J.R. McGraw, S.K. Skedzielewski, S.J. Allan, R.R. Odledhoeft, , J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. Sisal: Streams and iteration in a single assignment language: Reference manual version 1.2. Technical report, Lawrence Livermore National Laboratories, Livermore CA, March 1985.

[MWV92] Sunil Mirapuri, Michael Woodacre, and Mader Vasseghi. The Mips R4000 processor. *IEEE Micro*, pages 10–22, April 1992.

[Nik91] R.S. Nikhil. ID language reference manual. Computation Structure Group Memo 284-2, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, Massachusetts 02139, July 1991.

[Nik93] Rishiyur S. Nikhil. A multithreaded implementation of Id using P-RISC graphs. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, number 768 in Lecture Notes in Computer Science, pages 390–405, Portland, Oregon, August 1993. Springer-Verlag.

[Nik94] Rishiyur S. Nikhil. Cid: A parallel, shared-memory C for distributed-memory machines. In *Proceedings of the Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, August 1994.

[PC90] Gregory M. Papadopoulos and David E. Culler. Monsoon: An explicit token-store architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 82–91, Seattle, Washington, May 1990. Also: MIT Laboratory for Computer Science, Computation Structures Group Memo 306.

[Pea80] Judea Pearl. Asymptotic properties of minimax trees and game-searching procedures. *Artificial Intelligence*, 14(2):113–138, September 1980.

[PJGT94] Vijay S. Pande, Christopher F. Joerg, Alexander Yu Grosberg, and Toyoichi Tanaka. Enumerations of the hamiltonian walks on a cubic sublattice. *Journal of Physics A*, 27, 1994.

[POV93] POV-Ray Team. *Persistence of Vision Ray Tracer (POV-Ray) User's Documentation*, 1993.

[PYGT94] Vijay Pande, Alexander Yu, Grosberg, and Toyoichi Tanaka. Thermodynamic procedure to construct heteropolymers that can be renatured to recognize a given target molecule. *Proceeding of the National Academy of Science, U.S.A*, 91(12976), 1994.

[RLW94] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, Chicago, Illinois, April 1994.

[RSAU91] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, Hilton Head, South Carolina, July 1991.

[RSL93] 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.

[SFL+94] 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*, pages 297–306, San Jose, California, October 1994.

[SG90] E. Shakhnovich and A. Gutin. *J Chem. Phys.*, **93**, 5967, 1990.

[SKY91] S. Sakai, Y. Kodama, and Y. Yamaguchi. Prototype implementation of a highly parallel dataflow machine EM-4. In *Proceedings of the 5th International Parallel Processing Symposium*, pages 278–286, May 1991.

[Smi78] Burton J. Smith. A pipelined, shared resource MIMD computer. In *Proceedings of the 1978 International Conference on Parallel Processing*, pages 6–8, 1978.

[Ste88] 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.

[Str69] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354–356, 1969.

[Sun89] Sun Microsystems, Inc. *Sparc Architecture Manual, Version 8*, January 1989.

[Sun90] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[TBK93] 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.

[Thi91a] Thinking Machines Corporation, Cambridge, Massachusetts. *Getting Started in CM Fortran*, November 1991.

[Thi91b] Thinking Machines Corporation, Cambridge, Massachusetts. *Getting Started in *Lisp*, June 1991.

[Thi92] Thinking Machines Corporation, Cambridge, Massachusetts. *CM5 Technical Summary*, January 1992.

[Thi93] Thinking Machines Corporation, Cambridge, Massachusetts. *Getting Started in C**, May 1993.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.

[VR88] Mark T. Vandevoorde and Eric S. Roberts. WorkCrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, August 1988.

[WK91] I-Chen Wu and H. T. Kung. Communication complexity for parallel divide-and-conquer. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 151–162, San Juan, Puerto Rico, October 1991.

[ZO94] Y. Zhang and A. Ortynski. The efficiency of randomized parallel backtrack search. In *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, October 1994.

[ZSB94] 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.