A Parallel File I/O API for Cilk

by

Matthew S. DeBergalis

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

Bachelor of Science in Computer Science

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2000

Author . . . .
Department of Electrical Engineering and Computer Science
May 22, 2000

Certified by. . . . .
Charles E. Leiserson
Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A Parallel File I/O API for Cilk

by

Matthew S. DeBergalis

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2000, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Cheerio is an application programming interface (API) for efficient parallel file input and output (I/O) modeled closely after traditional serial POSIX I/O. Cheerio remedies a long-lived gap in multithreaded programming environments, which typically ignore file I/O and force programmers to implement parallel file I/O semantics on top of the raw operating system I/O API. Not only is this approach clumsy and semantically ugly, it can yield inefficient multithreaded programs.

Cheerio defines three parallel I/O modes, all of which are "faithful extensions" to traditional POSIX I/O, meaning that each mode reduces to ordinary POSIX semantics when running on a serial system. Under *global* mode, all threads share a single file pointer. *Local* mode specifies that each thread has a unique pointer unaffected by I/O operations in other threads. *Serial-append* mode guarantees that the final output of a program is identical to the output of the serial execution order. The serial-append mode is crucial for applications whose processing can be parallelized, but which must output results in a standard serial order. Serial-append mode departs significantly from other parallel APIs, and its definition and implementation form my major contribution.

I present an implementation of Cheerio under Cilk, a C-like multithreaded language which runs efficiently on a wide variety of hardware platforms. Cilk and Cheerio share much of the same philosophy, which makes Cilk suitable as an underlying parallel system. In particular, Cilk is a faithful extension of C in that programs can be reduced to a correct serial execution by deleting the Cilk keywords. Benchmarks confirm Cheerio's efficiency. A database program based on the GNU DBM database facility has a near-perfect speedup on 4 processors using local I/O. A version of the popular `compress` benchmark converted to output using serial-append mode achieves a speedup of 1.8 on 2 processors and 3.75 on 4 processors.

Thesis Supervisor: Charles E. Leiserson
Title: Professor

Acknowledgments

First, thanks to my advisor Charles Leiserson, and group members Harald Prokop and Matteo Frigo for their advice and guidance through the last few years. Your help has been indispensable.

Special thanks to Orca Systems, Inc. for their generous support. I can't imagine a finer place to work, even if it's now in Waltham.

Finally, there are some people who have had a tremendous impact on me over the years. I will not try to make a list, but rest assured that you all have had a profound effect on me, and I am indebted to you for who I am today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parallelism is one of the most popular techniques for achieving high computing performance. Advances in processor speed, parallel architecture design, and compiler technology have all increased the power of parallel computers, but in many cases this has merely pushed the bottleneck elsewhere. For many "Grand Challenge" problems [SC00], input/output (I/O) bandwidth became the limiting factor. More recent work (see [JWB96] for some examples) has successfully addressed the need for higher throughput I/O, but these systems are almost exclusively tailored for the sort of structured I/O patterns associated with regular scientific applications. Nieuwejaar and Kotz's extensions to I/O paradigms to allow efficient strided access [NK96] are a good example of this line of work, as are [MS96] and [DJT96].

The recent availability of cheap symmetric multiprocessor (SMP) systems has helped to shift the focus of parallel computing. Many desktop workstations now have two or four processors and run traditional POSIX-compliant[1] operating systems instead of the highly specialized environments of larger parallel hardware. These advances in turn drive a need for programs, other than the traditional scientific applications, that can run in parallel on such systems.

Supporting parallel I/O under SMP environments presents three challenges. Com-

---

[1] Specifically, the IEEE 1003.1-1990 standard, commonly referred to as "POSIX," specifies a standard application API, including support for I/O. The first standard was approved in 1990. The POSIX.4 realtime extensions were adopted in IEEE Std 1003.1b-1993 in 1993, which extended some of the I/O-related API calls.

pared to scientific computing programs, many applications running on these SMP machines have less structured I/O requirements and cannot make effective use of I/O APIs tuned for the regular I/O access patterns typically found in scientific applications. Another issue is that the POSIX I/O API, though providing some of the necessary features for a parallel I/O API, falls short in several key areas. It does not include many of the necessary synchronization guarantees between threads, nor does it have support for various useful pointer-sharing semantics. While it is true that most POSIX I/O implementations in use today are thread-safe from the perspective of operating-system level threads, these threads may not correspond to the parallel threads of execution in a given parallel environment.

This thesis describes the design and implementation of Cheerio, a parallel file I/O API that addresses these deficiencies. The rest of this chapter introduces Cheerio and outlines the thesis. Section 1.1 presents a motivating example for serial-append mode, the most interesting of Cheerio's three I/O modes. Section 1.2 details the main contributions of this thesis: the design of Cheerio and its implementation in the Cilk runtime system. Finally, Section 1.3 gives a roadmap for the remaining chapters of this thesis.

1.1   A motivating example

As a demonstration of Cheerio's utility, this section walks through a simplified example of an application that is converted to use Cheerio. It begins with a typical serial algorithm for traversing a tree, modifies it into a parallel algorithm, and then uses Cheerio to preserve the original order of the file. Figure 1-1 is a simple binary tree with nodes numbered 1 through 7, and Figure 1-2 shows a straightforward serial (single-threaded) algorithm for traversing it. In this case, the value of each node is printed using the UNIX `stdio` mechanism. Traversing the example tree yields the output `1 2 3 4 5 6 7`.

In a real application there may be a significant amount of computation required at each node that `walk()` visits. If the computation at different nodes is mostly inde-
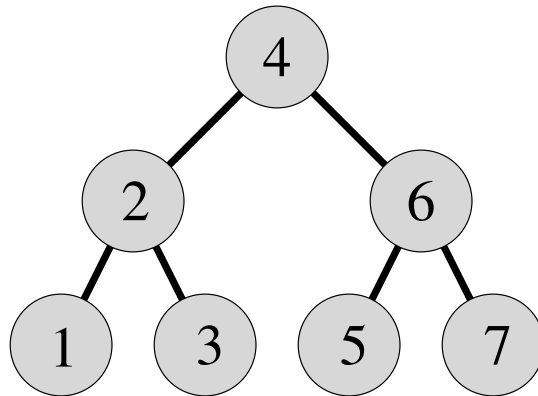
Figure 1-1: Sample 7-node binary tree.

```
f = fopen(name, "r");
walk(root);

void walk(node n)
{
        if (n.left)
                walk(n.left);
        fprintf(f, "%d ", n.val);
        if (n.right)
                walk(n.right);
}
```

Figure 1-2: Serial tree traversal algorithm written in C

pendent, it may be advantageous to write a parallel version of the traversal algorithm that can visit multiple nodes concurrently. Figure 1-3 shows the same algorithm as before, but written in Cilk instead of C. A description of Cilk is outside the scope of this work, but intuitively, instead of recursively calling itself on the left and right children of the current node, `walk()` spawns concurrent threads to visit the children simultaneously. The only synchronization in this example is the `sync;` statement which forces the two recursively spawned `walk()`s to return before their parent completes.[2] Many different outputs from the parallel traversal can occur depending on the particular schedule of an execution; two such possibilities are 4 6 5 2 3 1 7 and 2 4 3 6 1 7 5. This example assumes that the calls to `printf()` are atomic. Once

---

[2]In fact, the `sync;` statement is only needed in this example for extra clarity, since every Cilk procedure ends with an implicit `sync;`.

```
f = fopen(name, "r");
walk(root);

cilk void walk(node n)
{
        if (n.left)
                spawn walk(n.left);
        fprintf(f, "%d ", n.val);
        if (n.right)
                spawn walk(n.right);
        sync;
}
```

Figure 1-3: Parallel tree traversal algorithm written in Cilk

one thread initiates output, it completes that output before another thread can begin. Cheerio preserves this same atomicity guarantee.

If the computation at each node is independent, then the programmer can run the traversal in parallel, boosting performance. Unfortunately, the order of output is nondeterministic, which may not be acceptable under certain cases. If the programmer wants the output of the program in serial order, then either she cannot make use of the available parallelism, or she must buffer the output until all of the preceding threads write their output. Doing that for a 7-node tree isn't too difficult, but such a solution doesn't scale well to larger problems, or from problem to problem. Cheerio's serial-append I/O mode offers a more desirable solution. Figure 1.1 shows the same algorithm as the parallel version written in Cilk, but now it uses Cheerio to open the file in serial append mode. Nothing else changes, which is an important strength of Cheerio. The program still runs in parallel, but now the output is guaranteed to be the output of the serial version, namely 1 2 3 4 5 6 7. Cheerio provides this behavior with little performance overhead and near-zero additional programmer effort.

```
#include <cheerio.h>

f = fopen(name, CIO_SERIAL, "r");
walk(root);

cilk void walk(node n)
{
        if (n.left)
                spawn walk(n.left);
        fprintf(f, "%d ", n.val);
        if (n.right)
                spawn walk(n.right);
        sync;
}
```

Figure 1-4: Parallel tree traversal algorithm written in Cilk using Cheerio. The only difference from Figure 1-3 is the mode specification in the `fopen()` call.

1.2   Contributions

The central contribution of this thesis is the design and implementation of Cheerio, a parallel API designed for unstructured parallel I/O. Cheerio maintains compatibility with traditional POSIX I/O semantics when appropriate, easing the task of porting serial applications to a parallel execution environment. As a proof of concept, the thesis includes an implementation of Cheerio for the multithreaded Cilk language, a parallel language developed at MIT. Cilk is designed to aid programmers in writing parallel applications by hiding the details of scheduling parallel execution, thus simplifying the work of the programmer. Cilk also allows the runtime system to schedule the same program efficiently on a wide range of platforms, from a single-processor machine to a machine with hundreds of processors. The Cilk runtime library is optimized for low overhead when spawning a new thread, so parallel programs can be written to use many concurrent execution streams and not suffer when running on a machine with fewer processors. Cilk has the further advantage of guaranteeing efficient and predictable performance, despite running on such a variety of parallel platforms. Cilk is more fully described in [FLR98] and [Ran98].

Despite Cilk's goal of simplifying parallel software development, it lacks a parallel file I/O framework. Cilk programs must use the standard POSIX file I/O library, and

they must explicitly arrange any necessary serialization or multiplexing. A programmer who wants to use parallel I/O with some deterministic behavior must provide her own I/O routines to do the necessary locking and bookkeeping. User I/O is made difficult because of Cilk's involved mapping of Cilk threads to operating system processes. Even worse, for those userspace routines to be efficient on a wide variety of hardware platforms, they must contain several different routines optimized for each hardware case. This lack of portability goes against the principle that the Cilk system should abstract away that sort of detail, and it breaks the property that the same application runs efficiently on both single-processor and multiprocessor machines.

1.3   Overview

The remainder of this thesis is organized as follows.

Chapter 2 describes the Cheerio API. It defines the semantics of Cheerio's three I/O modes: global, local, and serial-append. A complete reference of Cheerio entry points follows, along with a list of caveats, and a discussion about porting applications to Cheerio.

Chapter 3 presents the rationale for Cheerio's design decisions. It presents the crucial idea of a *faithful* extension, first introduced in [FLR98], along with four criteria for designing Cheerio: similar to POSIX when appropriate, high performance, hardware and platform independent, and minimal but complete. Chapter 3 also presents two other parallel APIs, compares them to Cheerio, and demonstrates why Cheerio is a better API in certain situations.

Chapter 4 describes the implementation of Cheerio's local and global modes in the Cilk runtime library. This description also includes the underpinnings of the third mode, serial-append. This chapter ends with sections on a discussion of implementation and file portability, and it lists the ways that my implementation of Cheerio is not entirely conformant to the specification presented in Chapter 2.

Chapter 5 describes the implementation of Cheerio's serial-append mode in the Cilk runtime library. It builds off of the data structures introduced in Chapter 4,

going on to explain the Cheerio file format, the structure of Cheerio I/O nodes, and an internal API to manipulate them. Chapter 5 concludes with a discussion of accessing the data in Cheerio files after program execution.

Chapter 6 contains some measured results. These results include performance benchmarks of Cheerio that verify it scales well with available processors, does not impose high overhead, and compares favorably with regular I/O throughput.

Chapter 7 presents some conclusions and suggestions for future work.

Chapter 2

The Cheerio Parallel I/O API

The approach I took in designing the Cheerio parallel I/O API was to begin with POSIX and add as little as possible to achieve low-overhead, high-performance I/O. The end result is an API based on POSIX I/O which provides several parallel "modes" appropriate for different programming situations. This chapter describes Cheerio's syntax and semantics, and it discusses porting applications to use Cheerio. Chapter 3 will present the design rationale behind Cheerio and justify the choices made in building the API.

The crucial addition to POSIX necessary for a complete parallel I/O API is a specification of the behavior of the file pointer in the presence of simultaneous threads accessing the file. POSIX does specify an API entry point (`fork(2)`) to create multiple processes, but although it describes the behavior of shared descriptors across a fork, it is insufficient for two reasons. First, related processes under POSIX share a common file pointer. This semantics is useful under some circumstances, but troublesome when threads need to have independent pointer behavior. POSIX does provide a way to duplicate a descriptor to allow for thread-independent pointers, but it is cumbersome to use, requiring an `open()` to create the first descriptor, a `dup()` to create the copy, and then two `close()` calls after the `fork()`, one in each process. In a nutshell, POSIX I/O was not really designed to support tightly-integrated parallel programs. The second issue with just using the POSIX API is that under Cilk, computation threads and UNIX processes are different entities. Even under other

parallel environments, it may be incorrect to assume that the threads provided by the environment use POSIX pointer sharing semantics.

The Cheerio API is similar to the POSIX I/O API. Programs still open files using the `open()` call before using them. The `open()` call returns an integer handle used to identify the open file in all other API calls. The program should close files with `close()` when it is finished with them. An open file handle has a current file pointer, which is an integer offset into the file. The `read()` and `write()` calls read from and write to the file, respectively, starting from the current file pointer and updating it appropriately. Cheerio provides the `lseek()` function to move the pointer. A specification of these API entry points follows, but the only major difference between these functions and their POSIX equivalents are that `open()` takes an additional parameter specifying the *Cheerio mode* of the file. The Cheerio mode specifies the behavior of the file pointer across parallel Cilk threads.

The Cheerio API provides three modes for file I/O: global, local, and serial-append. The modes specify different behaviors for the file pointer, and are otherwise equivalent. This chapter describes global mode in Section 2.1, local mode in Section 2.2, and serial-append mode in Section 2.3. Section 2.4 is a programmer's reference to the Cheerio API, and Section 2.5 discusses porting programs to use Cheerio.

2.1 Global mode

This section describes global mode, the simplest mode in the Cheerio API. All threads using a file descriptor in global mode share a common file pointer. Figure 2.1 describes this mode pictorially. File access is atomic, so when two threads read from the same descriptor, they read different pieces of the file. Global mode is part of Cheerio for two reasons: it is an obvious counterpart to local mode (described below), and it has a simple implementation. Since global mode is expensive to use, it is only appropriate when threads require the strong synchronization of file access. A well-structured parallel application should require as little synchronization between threads as pos-
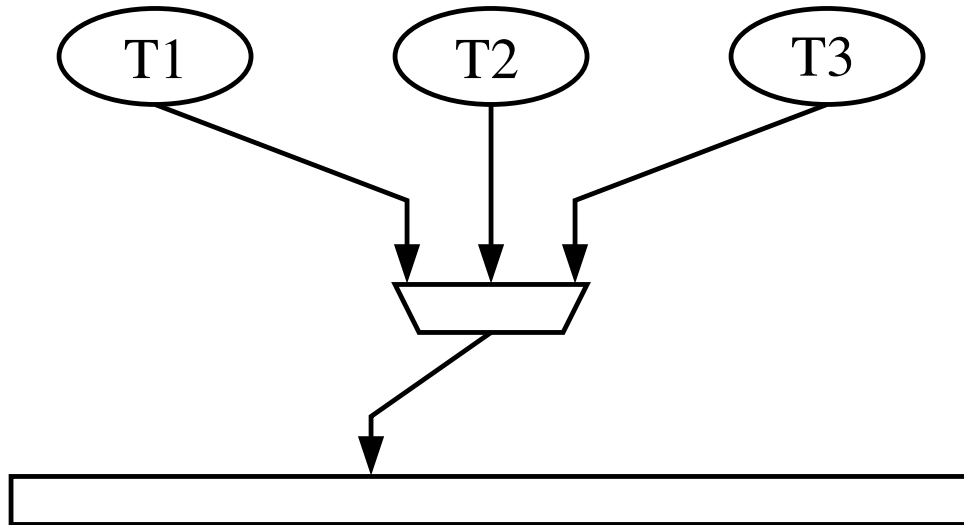
Figure 2-1: Cheerio's global mode. Each thread shares a single common file pointer.

sible. In particular, one does not want file access to require "one at a time" access limits. Global mode is therefore often useful as a stopgap measure when a program is first parallelized. It is the moral equivalent of mutexed access to global variables: it may be necessary at first, but for maximal performance the programmer should restructure the application to avoid synchronized access to those variables.

2.2   Local mode

Local mode is significantly more useful than global mode for parallel applications, but at the cost of a more complex semantics and implementation. This section describes these semantics, particularly the preservation of pointer locations across spawns. It then describes a parallel database library based on the GNU DBM implementation, a good example of local mode's strength.

In local mode, since each thread has its own value of the file pointer, pointer updates in one thread do not affect another thread. Figure 2.2 shows this arrangement. Each thread can read or write to the Cheerio file descriptor independently of the other threads. Using a file opened in local mode is similar to opening a new descriptor for the file in each thread, but without the overhead of so many opens, and with the added advantage of some semantics governing a thread's initial pointer position when
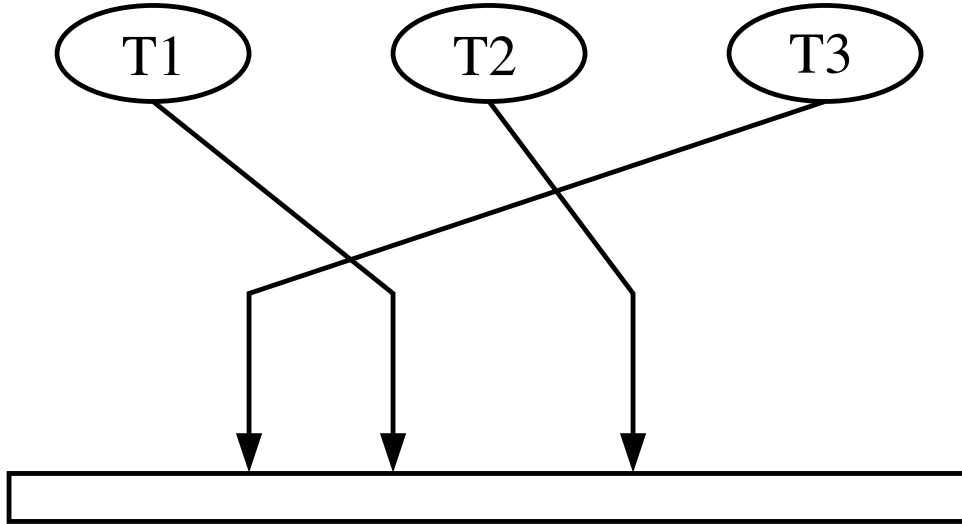
17

Figure 2-2: Cheerio's local mode. Each thread has its own independent file pointer.

it is spawned. Because different threads have independent pointers, the semantics of local mode must also specify the location of the file pointer in both the parent and child of a spawn. Implementing this behavior, however, turns out to be difficult, because the natural parallel semantics and the natural serial semantics conflict with each other. We define the serial semantics to be the behavior of the program where the spawns are replaced with normal function calls, making it a single-threaded serial program. This transformation is called the *serial elision* of the program, whose notion is developed further in Chapter 3. Consider a thread T_p that has an open Cheerio local mode descriptor, and suppose that T_p spawns thread T_c. Two important cases do yield consistent semantics across both models. In the first, suppose that T_c and its children never use the open descriptor. One would certainly expect the pointer in T_p to remain unchanged across the spawn, and this behavior is a sane semantics in the parallel world. In the second case, T_p never uses the pointer again, suggesting that T_c should begin execution with the same pointer location as T_p had at the spawn point. Again this behavior is consistent in a parallel view.

The difficulty begins when both T_c and T_p use the pointer, because the serial elision specifies behavior that runs counter to the natural parallel semantics. Namely, in the serial elision, T_c executes and modifies the pointer, and after T_c returns, T_p sees an updated pointer location. On the other hand, parallel semantics imply that T_c
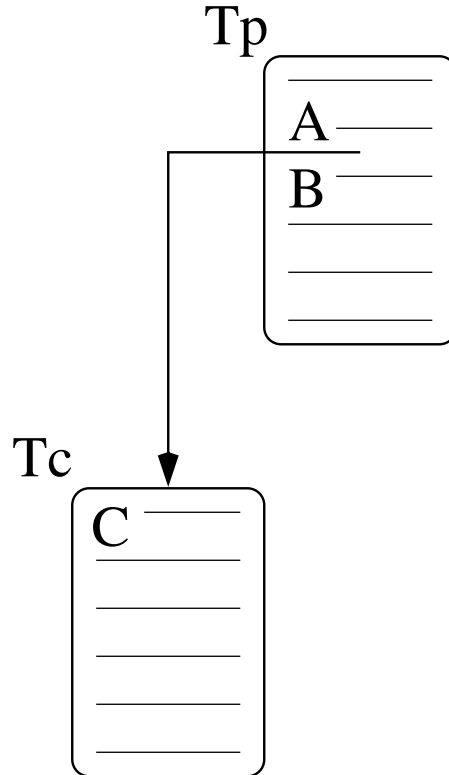
Figure 2-3: Spawns give rise to three possible local mode semantics. In this example, T_p spawns T_c just after statement A. If T_c does not use the open local descriptor then statement B will have the same pointer location that statement A had. If T_p never uses the open descriptor again, then statement C will inherit the pointer position that statement A had. If both T_p and T_c use the descriptor, then the initial pointer after the spawn at both points B and C is left undefined.

and T_p should both start with the same initial file pointer position. Cheerio legislates a solution to the problem: since no initial pointer semantics is consistent in both the serial and parallel views, the value of the file pointer after a spawn *in both the child and the parent threads* is left undefined. As described above, the important exception occurs if one of the two threads doesn't use the pointer, in which case the other thread keeps the value of the pointer from before the spawn. Figure 2.2 illustrates the different possibilities.

These semantics can be considered as *dag consistent*, described in [Fri98], [FLR98], and [Ran98]. In the context of shared memory, dag consistency states that a read can "see" a write only if there is some serial execution which is a valid schedule of the reads and writes in which the read sees the write. Consequently, different executions can

yield different values for the read, if they both correspond to valid serial executions in the dag. While there is only one serial elision of a Cilk program, there can be many valid serial executions; two threads that are spawned without an intervening `sync` can execute in any order and still preserve the dag.

The same concept applies to I/O access, or in particular to the initial file pointer in a thread. When a thread is spawned, we define its initial pointer to be whatever its parent's pointer is. If thread A spawns threads B and C and then does a `sync`, then the dag allows either B or C to execute first in a serial path. In the case where A spawns B and does no I/O itself, there is no write in the serial path regardless of which path is taken, so B is guaranteed to start with A's pointer value. Similarly, if B does no I/O, then again there is no write regardless of path, so A's pointer is preserved. If both A and B attempt I/O, then different paths can give rise to different pointer values, and thus Cheerio cannot guarantee an initial pointer value in either A or B.

Some clarification is in order here. Only the *initial* pointer value is dag consistent. Inside a function, local mode semantics specify that only the current thread can affect the value of a pointer. On the other hand, the actual I/O access is always dag consistent, because two threads pointing at the same location in a file might write to it in either order if there are valid serial executions including both orders in the dag.

Local mode has numerous applications in the parallel programming space. The GNU database manager (GDBM) [GNU] was one of the first applications ported to Cilk and Cheerio as a validation of the Cheerio API design. It remains an excellent example of the power of Cheerio's local mode semantics. The serial version of GDBM maintains a table of ⟨*key, value*⟩ pairs. Queries search the table for matching keys using a hash table of the keys for greater efficiency. The port to Cilk parallelizes queries by spawning multiple threads, each of which search a different part of the key space. Parallel GDBM also extends the GDBM API with a parallel query, which can search for multiple keys in parallel, returning a list of answers once the query has completed. This model of multiple threads each searching a difference piece of the key space is perfectly suited for local mode, because each searching thread operates

independently of the rest of the searching threads.

2.3   Serial-append mode

Serial-append mode represents the bulk of this thesis work. This mode opens a whole class of applications to parallelization that were previously difficult. This section describes its semantics, which are simple and straightforward. Specifically, serial-append specifies that the final output of a parallel program doing appending writes is the same as the output of the serial elision of the program, regardless of the actual order of thread execution.[1] These are simple semantics, but an efficient implementation of serial-append is tricky. Chapter 5 is devoted to explaining the implementation of serial-append under Cilk. This section also gives several examples of applications that benefit from serial-append.

Global and local modes support appending opens by specifying a flag to the Cheerio `open` call, just as files are opened for appending in the POSIX API. Serial-append mode is different, because the semantics rely on an appending write in the serial case. A mode which guarantees serial output regardless of pointer updates would be far more difficult to implement, and it doesn't appear to have many useful applications.

Serial-append is most useful for when an application needs to be parallelized, but the output must still correspond to a serial version. This semantics allows parallelization of programs that were previously difficult to port to Cilk because of their inherent serial I/O needs. The range of such applications is tremendous. Compilers, text processing tools such as LaTeX and `troff`, and various formatting tools like `fmt` and `indent` all share this attribute. As a specific example, `fmt` refills paragraphs of a text file to have even margins. Parallelizing `fmt` without serial-append is challenging, because much of the program's work is I/O bound, and paragraphs may be larger or smaller after filling due to the addition or deletion of spaces and carriage returns. With serial-append the parallelization is almost trivial. The program simply reads in

---

[1]This definition assumes that the program is deterministic. In the nondeterministic case, the semantics must be reduced to guaranteeing that the output of the program is equivalent to *some* execution of the serial elision.
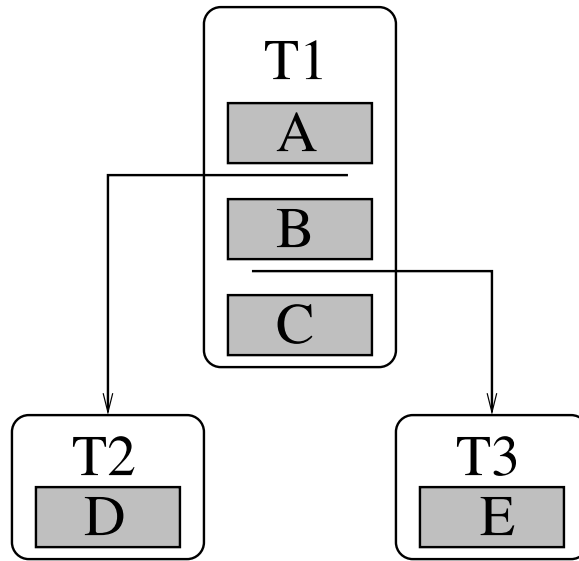
Figure 2-4: A small tree of threads in serial-append mode. Capital letters are regions of I/O and arrows represent spawns. In this example, the correct serial ordering of the output is A D B E C.

a paragraph and spawns a thread to fill it. Each thread writes the filled paragraph back to disk. None of the global structure of the program has to change, and there is no need to have the entire file in memory at any time, yet `fmt` can make full use of multiple processors and enjoys a high speedup when running in parallel.

Other types of applications are also appropriate, including anything that needs to log a record of execution. Serial-append and local mode with appending writes form a nice complement here. Imagine a chess-playing program that logs each evaluated board position. Suppose that many threads are evaluating different positions, and each thread writes some summary information to an open Cheerio file for each visited position. If the file is opened in local mode with the append flag set, then the output is a record of evaluated boards in order of their evaluation, with the various threads output interleaved. Changing the mode from local to serial-append in the open call changes the behavior so that the written output file is a record of the positions evaluated, but this time in the order that the serial elision would have visited them. The version of the program that uses serial-append mode could yield, say, a depth-

first search order of the tree, which may be far more useful to analyze that some random ordering of the positions. Generally there are some cases where the output should reflect the parallelism, but for those where it should not, serial-append allows the programmer to leverage the benefits of parallel execution while still producing the necessary output.

2.4   Cheerio API reference

This section is a complete Cheerio API reference.[2]

- `int open(const char *path, int cheerio_mode, int flags, mode_t mode)`

  The file name specified by `path` is opened for reading and/or writing as specified by the argument flags and the file descriptor returned to the calling thread. The flags specify whether the file is being opened for read or write, appending behavior, and a variety of other modifiers on the open descriptor. If successful, `open()` returns a nonnegative integer, termed a file descriptor. It returns -1 on failure. The file pointer used to mark the current position within the file is set to the beginning of the file. The variable `cheerio_mode` is one of `CIO_GLOBAL`, `CIO_LOCAL`, or `CIO_SERIAL_APPEND`. In the case of `CIO_SERIAL_APPEND`, the presence of `O_APPEND` in the flags is implied. The `open()` call is the only Cheerio entry point whose prototype differs from the corresponding POSIX call.

  The `open()` call is purposely incompatible with the traditional POSIX prototype because a programmer using Cheerio should be required to explicitly pick the correct parallel mode of I/O for each open descriptor. Cheerio does support an alternate form of the `open` call which omits the `cheerio_mode` parameter. In this case, the mode defaults to global mode.

- `int close(int d)`

---

[2]The prototypes and much of the descriptions of the Cheerio API functions come from the NetBSD Programmer's Manual, which is itself derived from the 4.4BSD release described in [MBKQ96].

The `close()` call deletes an open descriptor from the table of descriptors. If the descriptor being closed is the last reference to the underlying file, then the object is deactivated. In particular, the current seek pointer associated with the file is lost. In the case of serial-append mode, which might buffer file metadata in memory, all metadata is flushed to the output file on close. There is an implicit close of every open Cheerio descriptor when the Cilk program completes.

- `off_t lseek(int fildes, off_t offset, int whence)`

- `int read(int d, void *buf, size_t nbytes)`

- `int write(int d, void *buf, size_t nbytes)`

  These three functions work exactly as their POSIX counterparts. The `lseek()` function repositions the pointer according to the values of `offset` and `whence`. The `read()` and `write()` functions use and update the current file pointer in whatever manner is appropriate for the Cheerio mode of the open file, and either read into a buffer or write the contents of the buffer to the file.

- `int dup(int oldd)`

- `int dup2(int oldd, int newd)`

  The `dup()` function duplicates an existing Cheerio descriptor and returns its value to the calling thread. The integer `oldd` must be a currently open Cheerio descriptor. The new descriptor shares the old descriptor's underlying file object, including the seek pointer and Cheerio mode. The `dup2()` variant allows the program to specify the desired new descriptor number as `newd`. If that descriptor is already open, it is first implicitly closed.

2.5   Porting applications to use Cheerio

This section describes several issues that arise when porting applications that do not use the POSIX I/O API to Cheerio. Porting applications that already use POSIX

I/O to use Cheerio is quite easy, owing to the intentional similarly in the APIs. The compile-time headers rename all of the necessary functions so that calling `open`, `read`, and the like invokes the Cheerio-aware implementations instead of the underlying operating system calls. Typically, the programmer need not concern herself with such details, however, and needs only to modify the initial `open` call to specify a Cheerio mode.

Many programs do not use raw POSIX I/O calls, which complicates the picture slightly. The implementation of Cheerio described in this thesis allows other I/O implementations to work as well, particularly the standard I/O library typically provided by UNIX-like operating systems. Cheerio supports these calls differently from the way it supports traditional POSIX calls. Instead of reimplementing the standard I/O library for Cheerio, which would be a significant undertaking, the Cheerio implementation takes advantage of the fact that the standard I/O library, and indeed any library that provides an I/O framework, must use the POSIX API to do the actual I/O. By playing tricks with the system toolchain and the runtime library paths, Cheerio inserts itself between these I/O libraries and the POSIX system call entries, so that an unmodified vendor-provided standard I/O library will use Cheerio instead of traditional POSIX for its underlying I/O. This technique also allows programs that use the `curses` library[AA86] to be parallelized, since the curses calls would eventually turn into an entry into the Cheerio library.

Chapter 3

Cheerio Design Rationale

Cheerio was designed to fill a need that current parallel I/O APIs do not address. Instead of targeting specific file access patterns, Cheerio is designed to accommodate programs that have I/O access patterns more typical of a traditional serial program. This chapter describes the rationales behind Cheerio's design. Section 3.1 begins by describing the concept of a *faithful* extension, which forms the basis of Cheerio's design, and then Section 3.2 discusses tradeoffs between Cheerio's four other design criteria: similarity to POSIX, high performance, hardware and platform independent, and minimal but complete. Finally, Section 3.3 compares Cheerio to more traditional parallel APIs like MPI-IO and Vulcan.

3.1   Cheerio as a faithful extension of POSIX

Cilk's philosophy is to be a parallel extension of C. One way of expressing this philosophy is to consider the *serial elision* of a Cilk program, which is obtained by eliding all of the Cilk keywords from the program text. The result is a valid C program, which is a correct serial implementation of the original parallel program. The ability to construct a valid serial version of the original program is how Cilk can be called a *faithful* extension of C [FLR98]. That concept is a crucial part of Cilk's design, and one of its consequences is that the porting of serial programs to Cilk is simplified. Since every serial C program is already a valid Cilk program, a programmer merely

converts pieces of the program into parallel functions and can do as much or little as she needs.

In a similar vein, Cheerio is a faithful extension of POSIX. Cheerio extends POSIX I/O in the natural parallel way, but in this case there are three different modes, global, local, and serial-append, each corresponding to a different natural extension. Again, looking from the reverse perspective, the serial elision of each Cheerio mode degenerates into the normal serial POSIX semantics. Similarly, all Cheerio modes behave exactly the same way when running on a single processor. Just as Cilk's extension of C makes Cilk an easy porting target, being a faithful extension of POSIX I/O makes Cheerio particularly easy to target. Applications do not need to fundamentally change their I/O framework to take advantage of Cheerio's parallel semantics, and in many cases a programmer can get away with only minor modifications to select the right parallel mode.[1]

There is an important distinction between sharing a philosophy with Cilk and tying Cheerio to a particular parallel environment. Cheerio's modes are all faithful extensions of POSIX I/O, something not found in other parallel APIs. An API that is a faithful extension of POSIX offers advantages regardless of the parallel environment in which it is used. Cheerio certainly meshes well with Cilk's own philosophy, but that doesn't mean Cheerio is any less appropriate for doing I/O under Pthreads or some other parallel environment.

3.2   Criteria for a parallel I/O API

Given that Cheerio is intended to be a faithful extension of POSIX I/O, it must also satisfy four criteria. This section describes Cheerio's four design criteria. These are similarity to POSIX, high performance, hardware and platform independent, and minimal but complete. The order of these criteria is significant: as an example, Cheerio sacrifices higher performance for the sake of being POSIX-compliant. The

---

[1]The UNIX `compress` utility provides a good example. Porting `compress` from C to Cilk required some effort, but solving the I/O ordering problem required changing two calls to `open()` to specify the serial-append Cheerio mode, and nothing more.

opposite tradeoff quickly leads one into the realm of parallel I/O APIs designed for specific classes of applications instead of being appropriate across a wide range of programmer needs.

Similar to POSIX when appropriate

First and foremost, Cheerio is intended to be as similar to the traditional POSIX I/O API as possible. Just as Cilk does not require any fundamental change in the structure of the program nor any change in the syntax of the program,[2] converting a program to use Cheerio should not require any significant changes in the program's structure or syntax. Similarity to POSIX need not extend to 100% compatibility, and that leeway is justified by a few factors. First, I/O operations occur at well-defined points of a program, and some of them like `open()` occur fairly rarely. For example, if Cheerio required a change in the interface to `open()`, it wouldn't be unreasonable to change each invocation of `open()` appropriately. Less desirable, but still acceptable, would be to change the interface to `read()` or `write()`, which occur fairly frequently. Least acceptable would be an API that replaced the notion of reading and writing a file using a file pointer with some other paradigm. Such a change would require a complete rewrite of the I/O calls, and possibly an even more drastic change in the program's organization.

The second factor reducing the importance of maintaining complete POSIX compatibility is that parallel I/O can be quite different from serial I/O. No single file pointer semantics makes perfect sense for all applications; rather, several different extensions to the serial semantics are all appropriate in different contexts. Declaring one of them to be the default semantics is merely confusing and offers little advantage. Typically, programs ported to use Cheerio are either serial programs that were not written with concurrent I/O in mind, or they are programs with internal parallel I/O

---

[2]The straightforward parallelization of an efficient serial algorithm does not necessarily lead to an efficient parallel algorithm, for there are many examples where such a parallelization is far less efficient than a different algorithm. Rather, the point is that the naive parallelization is easy to program, and modulo any issues with the algorithm itself, the naive parallelization will have reasonable parallel speedup.

code which is better replaced with Cheerio API calls. In either case, the program's I/O logic needs to be closely examined, and so there is little disadvantage in forcing small changes in certain I/O API entry points. Cheerio does in fact change the interface to `open()`, and as expected this change has not adversely affected the ease of porting applications to use the API.

High performance

The API certainly must not prevent high-performance I/O. Performance is typcially governed by the implementation of the API, not its design, but it is important to avoid designs that effectively paint the programmer into a low-performance corner. Examples would be APIs that can only be implemented with multiple system calls per I/O request or APIs that require excessive amounts of memory, such as an API interface that requires a large data buffer even for small reads and writes.

Hardware and platform independent

One of Cilk's virtues is that it runs efficiently on a wide range of hardware. The runtime system can distribute work across many processors, but especially on machines with few processors, the overhead from the Cilk runtime system is nearly negligible. The I/O system should also run efficiently on a varying number of processors, so that it the programmer need not worry about how many processors the target machine runs. A properly designed API yields maximum I/O performance across the entire supported hardware space.

The API must also hide platform variations from the end user whenever possible or appropriate. Files written on a little endian machine should be readable from a big endian machine. As with traditional I/O, the application is still responsible for preserving the correct byte order of the file data itself, but any metadata in the file used by the runtime system must be properly interpreted on either endianness. Similarly, files written on a 32-bit machine should work on a 64-bit machine, and vice-versa. Finally, the underlying file system should be transparent, so that Cheerio

operations on top of an NFS other remote file system should work just as well as files written to local disk, modulo performance differences between the file systems.

Minimal but complete

A good parallel API walks a fine line between useful semantics and efficient implementation. At one extreme are APIs which hide all of the complexity of parallel I/O from the programmer but require excessive resources or have overly complex implementations. At the other extreme are extremely simple and efficient APIs that are cumbersome to use in practice, because they don't encompass enough functionality. Since the best solutions to problems in computer science tend to be elegant and minimal, Cheerio follows that design philosophy as much as is practical. A minimal set of APIs simplifies porting Cheerio to other architectures. It also allows a programmer to grasp Cheerio more easily than an alternative API with dozens of modes and options.

Cilk is a minimal extension to C, and the Cheerio API exhibits the same spirit as much as possible. A minimal API eliminates redundant modes and modes that are easily implemented in terms of other modes, and it also chooses simpler interfaces to API functions over more complicated, overly generic interfaces. Other existing parallel APIs are generally geared toward massive amounts of I/O that have uniform access patterns, and as a result these APIs tend to differ significantly from the POSIX API ([POS90] and [POS93]) A good example is the MPI-II standard ([MPI97a]), which includes an API for parallel I/O. While compatibility or even similarity with POSIX is not a requirement, it is an advantage. Cheerio's similarity to POSIX eases the task of parallelizing I/O intensive applications using Cilk, whereas implementing an entirely different paradigm for parallel I/O may entail extensive rewrites of parts of applications.

As an example of eliminating a mode that can be trivially emulated using other modes, the Cheerio API does not support synchronizing I/O at granularity higher than single `read()`s and `write()`s. One can imagine a program that wants to output several lines to a file and ensure that they appear sequentially. There are several possible solutions at an API level, including a `continuing_write()` function that

does a normal write but maintains a lock on the descriptor until a normal `write()` releases it. Another possibility is to have a `queue_write()` which queues writes in some intermediate buffer until a `commit()` call. Either of these solutions works, but it is just as easy for an application that needs this functionality to enclose the required atomic operations inside a lock. Alternatively, the application can buffer up its output internally before issuing a single write, leaving the API uncluttered. Allowing the application to handle this leaves Cheerio as the desired minimal set of primitives.

3.3   Comparison between Cheerio and other parallel APIs

Cheerio is unique among parallel I/O APIs in several respects. This section surveys other APIs in the parallel I/O space and compares them to Cheerio. While there are many other solutions for parallel I/O, to date they have all clustered around the needs of scientific applications. Specifically, they are not faithful extensions of POSIX I/O and do not satisfy all four of Cheerio's criteria.

MPI-IO

The Message-Passing Interface (MPI) standard, defined in [MPI97b] and extended in [MPI97a], provides a set of primitives for message-passing that enables the development of portable parallel message-passing programs. MPI-IO is an extension to MPI that defines I/O primitives appropriate for these sorts of parallel applications. It is highly geared for structured scientific applications, and it includes many pieces of other parallel I/O frameworks, including PPFS [HER+95], PIOUS [MS94, MS96], and Vesta [CF94].

MPI-IO is targeted at scientific applications, and it specifically favors performance over functionality [CFF+96]. It does not attempt to be compatible with the UNIX (POSIX) file interface, which is acceptable because programs must generally

be rewritten from scratch to make use of the message-passing paradigm.

Other APIs typically provide support for parallel I/O by defining modes specifying different semantics for simultaneous file access, much as does Cheerio. MPI-IO departs from this model to define I/O as a form of message passing, resulting in an API that encourages highly structured file access. Applications define the sort of high-level data type they will be reading and writing, declaring a stride and interleave pattern for file access. This model works well for scientific applications but falls well short of Cheerio's goal to support arbitrary I/O in an efficient manner. With some effort MPI-IO can support file access patterns similar to those allowed in global and local modes, but it offers nothing similar to the semantics of Cheerio's serial-append mode.

Parallel file systems

Another common approach to solving the parallel I/O problem is to build a parallel file system, such as Vesta. Here, the problem is reduced to a systems design task: the goal is to accelerate I/O by striping data across multiple disks as in RAID [PCGK89], or using multiple I/O processors as in the Vulcan massively parallel processor [SSA+94]. As with MPI-IO, the API for these solutions is intended for applications reading or writing large amounts of data to disk in a structured manner, and they fail to satisfy the needs of many applications that can benefit from Cheerio, which efficiently implements semantics similar to POSIX instead of forcing programs to use an entirely different I/O API.

Chapter 4

Implementation of Global and Local Modes

Chapter 2 describes an API which is largely independent of the choice of underlying thread implementation and operating system. Indeed, independence from the threading environment is one of the design goals of Cheerio. The implementation of Cheerio under Cilk, in contrast, is highly integrated with the Cilk runtime, leveraging various aspects of the Cilk implementation to improve performance. This chapter discusses how the semantics of Cheerio's global and local modes can be efficiently realized under Cilk.

The initial implementation of Cheerio was a set of wrappers that were explicitly compiled into a Cilk program. The wrappers served as a useful prototyping framework to evaluate different API designs. Performance and complete POSIX compatibility was not particularly important in the prototype, which allowed for a simple but inefficient implementation and a faster design cycle.

Once Cheerio was 90% designed, I abandoned the wrappers in favor of a "real" implementation that prioritized performance and compatibility. In this form, Cheerio was made part of the the Cilk runtime library, which contains the Cilk scheduler, locking primitives, and bootstrapping code. Some of the library, such as the scheduler, works behind the scenes to support parallel execution. Other parts serve as APIs exported to the Cilk user code. Cheerio is split between the two: the Cilk runtime

library exports a user-visible API for parallel I/O, but a good amount of user-invisible runtime support is also required.

The user-visible API overrides the traditional POSIX I/O entry points using the cilk_compat.h header file, just as Cilk replaces calls to functions like malloc() with Cilk-compatible versions. Programs can explicitly call the native OS versions of the Cheerio entry points, but there is typically no reason to do so, since OS descriptors can behave in unexpected ways under Cilk.

This chapter details the implementation of global and local modes. Section 4.1 explains how Cheerio's implementation handles Cilk's worker process model. This is relevant to all three Cheerio modes. Section 4.2 then explains the implementation of global mode, and Section 4.3 describes local mode's implementation. Finally Section 4.4 discusses parts of the implementation that are not entirely conformant to the specification in Chapter 2 and portability issues that arise when using Cheerio on multiple platforms.

4.1 Supporting I/O under multiple workers

Cilk creates an native operating system process or thread for each available processor in the machine. This presents a difficulty for Cheerio, which must make open Cheerio files available to threads running on any processor. This section describes how Cheerio reopens native descriptors on each worker as necessary to allow transparent access to Cheerio files on every processor.

The Cheerio implementation under Cilk is a layer on top of the native POSIX I/O API. Each Cheerio read or write maps to a POSIX read or write, and under ideal conditions there is little additional overhead. Certain conditions require additional POSIX calls for a Cheerio read or write, but the implementation tries fairly successfully to minimize the amortized cost of each read and write. Open and close calls are allowed to have more overhead, since applications tend not to call those as frequently or in critical paths.

When a process opens a file in Cilk, the runtime system makes a corresponding

call to the native `open()` function and saves the returned native descriptor in an internal table, indexed by the Cheerio descriptor. When a Cilk program issues an I/O request, the runtime system looks up the corresponding native file descriptor in the table and passes the call on to the operating system. Opening a single operating system descriptor for each Cheerio descriptor is insufficient, however, due to the way Cilk uses multiple processors in a machine. For each available processor, Cilk creates an operating system process or thread, depending on how it was configured. These native processes execute Cilk threads, and when a processor runs out of work to do, it steals work from another processor. Thus, from the perspective of the operating system, an executing Cilk program consists of n concurrent processes, where typically n is equal the number of processors.[1]

The Cilk worker processes are created during the Cilk runtime's bootstrap sequence before any of the programmer's Cilk code runs, so a descriptor opened on one worker are not shared with any of the other workers. Therefore, when a thread opens a Cheerio file descriptor, the corresponding operating system descriptor is only opened in the worker running on that particular processor. If another Cilk thread tries to use that Cheerio descriptor while running on another processor, there is no corresponding operating system descriptor open in that worker process, so the runtime system must have a way to reopen the file on other workers as it becomes necessary. To do so, the runtime system saves away a copy of the arguments to the original Cheerio open call so that other worker processes can reopen the file on the fly as necessary. Consequently, each Cheerio call must first verify that the current worker has a valid operating system descriptor open, and if not, must first open it and reset the file pointer before proceeding with the specified operation.

To keep track of native descriptors and file descriptors on each worker, the Cilk runtime library keeps an internal table, indexed by the Cheerio descriptor, which

---

[1]Cilk allows the user to specify how many of these worker processes to create. One can specify fewer than the number of available processors, in which case some processors remain idle. One can also specify more workers than available processors, which causes the operating system to task switch among various workers on the same processor. Modulo performance issues related to task switch overhead, cache properties, and the like, the parallelism of Cilk comes from multiple worker processes and has nothing to do with the number of physical processors in a system.

includes the native descriptor for each processor, the information needed to open the file on other workers, and pointer information appropriate for the descriptor's mode.

When a Cilk program opens a file, the Cilk runtime library opens a corresponding native OS file descriptor to the same file. This native descriptor is only valid on the current worker process, so the runtime system must have a way to reopen the file on other workers as it becomes necessary. In particular, assume process P_a has a Cilk thread at the top of its stack, and process P_b steals it. If the thread tries to perform I/O using a Cheerio file descriptor opened on P_a, the corresponding OS descriptor opened on P_a is useless, because native file descriptors are not shared among worker processes. Consequently, the runtime system must reopen a new descriptor for P_b's worker before proceeding with the thread's I/O request. To do so, the runtime system saves away a copy of the arguments so that it can reopen the file on other processors as that becomes necessary.

A similar issue arises when a Cilk thread closes a Cheerio file descriptor. The runtime system can immediately close the corresponding native descriptor, but the other native descriptors associated with other processors must be closed by their own worker processes. The current implementation uses a lazy closing scheme. When a thread closes a Cheerio descriptor it closes the native descriptor on the current worker, but the rest of the native file descriptors associated with the Cheerio descriptor are left open. The runtime system sets an invalid flag for the rest of the workers, signifying that the open native descriptor for that Cheerio descriptor is no longer valid. Eventually, a Cilk thread opens another file and reuses that Cheerio descriptor. When the Cilk runtime system accesses the native descriptor on a Cheerio read or write, it checks the valid flag. If set, the worker closes the stale native descriptor and then reopens a new descriptor as described in the previous paragraph.

4.2 Global mode

This section describes the implementation of global mode and an potential optimization that is not implemented in this thesis. Global I/O in Cheerio is straightforward.

Since Cilk threads update the same pointer in global mode, the runtime system cannot rely on the native pointer associated with OS descriptors. Instead, they use shared memory to keep track of the pointer for each Cheerio descriptor. The runtime system updates the kernel pointer for the current worker from the shared Cheerio copy, performs the native read or write operation, and then updates the shared pointer based on the result of the I/O operation. A running Cilk thread is never preempted, but since two threads on different processors may be accessing the same Cheerio descriptor, the pointer updates and the I/O operation must occur atomically. Global I/O has significantly reduced throughput because of this need for a global lock around each Cheerio API call.

One possible improvement which I chose not to implement for this thesis was to create an initial pool of descriptors before the runtime system forks off its pool of worker processes. These descriptors would then share a common pointer in the kernel, and the runtime system could leverage the shared native descriptors to have a cheaper way of maintaining cross-processor pointer consistency. If global mode were more useful in parallel applications, this improvement may have been worth the programming effort, but there are several tricky issues involved in preserving compatibility under this scheme that made it not worth the additional effort.

4.3   Local mode

This section details the implementation of local mode, whose analysis is considerably more involved that global mode's. It describes the portability and performance problems that beset the initial implementation in the user-level wrappers. A description of the current Cheerio implementation follows, including an explanation of how to efficiently preserve the pointer inheritance semantics described in Section 2.2.

When a file is opened in local mode, each thread maintains its own local pointer for each Cheerio descriptor. Recall that the Cheerio implementation converts each API request into a native operating system system call. Without any further work beyond passing each Cheerio request to the corresponding POSIX call, the I/O uses

whatever offset is stored in the kernel associated with the current worker process. Two different Cilk threads running on the same worker process might have different offsets, so the runtime system must set the kernel file pointer to the right place for the current Cilk thread and perform the requested I/O operation as an atomic transaction. This naive approach is exactly what the initial user-level wrappers implemented. Each per-thread pointer was stored in a `private` variable, and each Cheerio request was turned into a lock, a call to `lseek`, a call to `read` or `write`, and an unlock.

There are several problems with this approach. Primarily, each Cheerio request balloons into two system calls as well as an explicit lock around them. Both types of operations are expensive. Furthermore, private variables are not implemented on all Cilk platforms, sol programs that use the parallel I/O facilities are not as portable as those that do their own I/O multiplexing.

Fortunately, some properties of the Cilk scheduler can significantly reduce the need for extra seeks and locks. Cilk does not preemptively switch between threads; the Cilk scheduler is entirely cooperative, and control never returns to the scheduler unless a processor exhausts its work queue. When that happens, and only then, the scheduler steals the top frame from another processor. Consequently, a currently running thread never stops running on its processor unless it explicitly spawns another thread. This observation leads to the first optimization of the naive approach: there is no need to lock around the seek and the following read or write, because a thread can never be interrupted between those two operations.

Next, note that while we cannot completely dispense with saving a per-thread pointer, there are many cases where the runtime system can use the in-kernel pointer associated with the worker process' native descriptor without first setting it using an expensive seek operation. In fact, only a few situations require any synchronization of the kernel pointer to a per-thread pointer. Imagine thread T_p with open local mode Cheerio descriptor D spawns thread T_c, and T_c does not use D. As long as T_p is not stolen while T_c executes, the kernel pointer on the worker processor remains untouched during T_c's execution, and thus it contains the proper value when T_p resumes. If T_c uses D but T_p does not, then T_c begins with the correct pointer location. Finally, if

both T_p and T_c access D, then the semantics of local mode do not specify what pointer location the two threads see, and the implementation can therefore do anything it likes.

The only difficulty with this scheme arises on a steal. A thread can only be stolen if it has spawned a child thread which is currently running. If T_c does not touch the descriptor, then the runtime system is required to preserve T_p's pointer location. But here, all of the invariants work together in our favor. When T_p is stolen, the runtime system unconditionally sets the thief's pointer location to where the victim pointed, opening a new OS descriptor on the fly if necessary. If T_c has not used the descriptor, then T_p sees the correct value of the pointer after the steal, since it hasn't changed since the spawn of T_c. If T_c has touched the pointer, then T_p resumes execution with some undefined pointer location. If T_p does not use that pointer, then there is no concern. If it does, then we are in the case where both parent and child are using the pointer, relieving the runtime system from the need for synchronization. The upshot is that the runtime system can always use the OS descriptor pointer for reads and writes with no explicit locking, and the only overhead on an already expensive steal is a blind pointer copy per open Cheerio descriptor.

4.4 Portability and nonconformance

This section describes issues that come up when porting Cheerio to new Cilk platforms, differences in behavior across platforms and architectures, and the sharing of Cheerio files between different machines. It also discusses several areas where the implementation of Cheerio under Cilk is not entirely conformant to the API specification in Section 2.4. These nonconformances mirror typical operating system limits on POSIX I/O, so they were not judged to be fatal.

Portability

Cheerio is designed to be portable to Cilk on different architectures and to different threading libraries. Cheerio currently runs as a 32-bit library on Solaris and Linux.

It is entirely machine-independent code and uses Cilk primitives for locking and other machine-dependent tasks, so porting to new architectures should be easy. Cheerio has not been tested on any 64-bit architectures, but care has been taken to keep the Cheerio data structures 64-bit clean. Each new operating system presents some incompatibilities, but judging by the amount of work required to port Cheerio from Linux to Solaris, the changes are minimal and straightforward. Luckily, POSIX I/O behaves fairly consistently across vendor operating systems, which isn't always true of the rest of the POSIX APIs.

Porting to a different threading library is a far more comprehensive undertaking. The *design* of Cheerio is compatible with other threading packages, but the implementation is closely tied to Cilk, mostly for performance reasons. The entire treatment of Cheerio nodes under serial-append mode is particularly challenging, since it relies on work stealing instead of spawns to maintain consistency. Maintaining metadata during spawns is both more costly and requires more bookkeeping, so an efficient implementation of Cheerio on a nonstealing (i.e., nearly any) threading library would require some clever programming. Similarly, Cheerio currently only runs under the Cilk process model. The Cilk runtime can also be built to use threads instead of processes for its worker tasks, and some work would be required for Cheerio to support such a configuration.

Another aspect of portability is ensuring Cheerio files are portable across various architectures and threading platforms. The implementation of Cheerio described in this thesis is written to be independent of word size and endianness. Cheerio has not yet been run on a 64-bit machine, but care has been taken to maintain output file compatibility regardless of the word size of the machine. Cheerio does run on both big endian and little endian machines, and files written on one can be read by the other. The endian-independence is only for the Cheerio metadata that maintains the serial order of the data. The data itself is stored in whatever byte order the application writes it in, so without any further care, an application writing word-sized data on a big endian machine cannot read the same data correctly on a little endian machine. This behavior is consistent with POSIX I/O; it it not the place of Cheerio to enforce

40

any natural byte order.

Nonconformance

Cheerio's implementation under Cilk deviates from its specification in three areas.

- Cheerio currently relies on the operating system's guarantee that writes of size at most BUFSIZ occur atomically. The API makes the same guarantee, but it could be argued that Cheerio should allow any sized write to occur atomically. Implementing this property requires an extra locking operation in some cases, as well as a test of the lock in all cases. Since it has not become an issue thus far, the runtime system does not do the extra work to avoid interleaving of large writes.

- As a simplifying assumption, Cheerio only allows a fixed number of open Cheerio files at once. This number can be increased by recompiling the Cilk runtime library with a greater limit. The default limit is 50.

- Because Cheerio uses a different native descriptor for each processor, closing a Cheerio file does not immediately close all of the native descriptors. This can cause unexpected behavior: residual open descriptors can prevent users from umounting file systems and affect which data is left in the buffer cache. Cheerio does not currently support the sync() call to force a flush of dirty pages to disk, but this has not been an issue thus far.

Chapter 5

Implementation of Serial-Append Mode

The most important contribution of this thesis is Cheerio's serial-append mode. This chapter details an efficient implementation of serial-append mode in the Cilk runtime system. Section 5.1 begins with a formal definition of serial-append's semantics and a conceptual overview of how this appears to the programmer. Section 5.2 then describes the runtime system's basic algorithm for writing serial-append data with minimal overhead. This section introduces the Cheerio file format: a file system designed for efficient serial-append semantics, which is implemented on top of a file that resides in the host operating system's file system. Next, Section 5.3 presents some optimizations on that basic algorithm which improve the efficiency of reading data from a Cheerio file, and lastly Section 5.4 describes three options for reading data back out of a Cheerio file.

5.1 Formal definition and conceptual solution

Formally, the Cheerio implementation must be able to reconstruct the serial-elision order of writes from any parallel execution. At a low level, the challenge is that once thread T_p spawns thread T_c, then both T_p and T_c may do I/O simultaneously, yet all of T_c's output must appear in the output file before any of T_p's output after the

spawn. The runtime support must adhere to the following three constraints:

- The runtime system should allow both T_p and T_c to write to disk simultaneously, maintaining sufficient information to reconstruct the correct serial order of the output.

- The ordering metadata itself should be written to disk as soon as possible. Being able to define a limit of how much memory is required for serial-append overhead as a function of the number of processors and number of open files would be nice. The limiting case of writing all of the metadata once the parallel program exits is insufficient, since the space required for metadata alone may grow unacceptably large under some pathological cases.

- The final output of a program must be in an format amenable to fast reading. Conversion to a traditional flat file should be efficient, as well as directly reading the file with Cheerio.

Implementing serial-append mode under these constraints is significantly more difficult than implementing the global or local I/O modes. In addition to opening files on multiple processors, the system must have a way to keep track of the correct ordering of data. One possible way to accomplish serial ordering is to keep all of the output of each thread in memory. Each thread's output is appended to the output file once all the threads that come before it in serial order have completed. Although this approach certainly preserves the desired ordering of the output file, it incurs the potentially high cost of keeping most of a program's output in memory for the duration of execution. Thus, this solution is unacceptable for programs of even moderate size.

Conceptually, the correct solution that writes the data to disk immediately requires an primitive insert operation. Figure 5-1 shows how a single process might view an appending write. The shading region represents data that has been written, and the file pointer indicates where the next write will begin. Figure 5-2 shows the difficulty extending this to the case of two threads writing simultaneously. If T_1 and
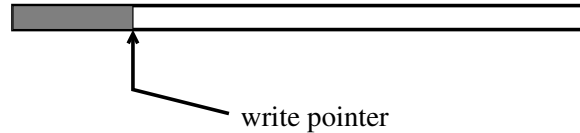
Figure 5-1: A single process doing an appending file write. Gray represents data that's been written, and the file pointer indicates where the next write will go.
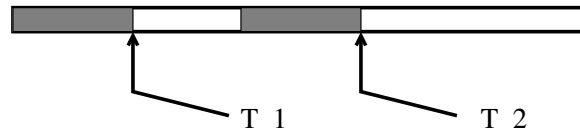

Figure 5-2: Two threads doing an appending file write. Gray represents data that's been written, and the file pointers indicate where each thread will write to. The issue is how to avoid the hole between T_1 and T_2 regardless of how much T_1 writes.

T_2 are spawned in that order, then the serial-append output must have all of T_1's data followed by all of T_2's data. The challenge is that the operating system does not provide an insert operation, and so T_2 must start its write at some fixed location. Since there is no way to know at that time how much data T_1 will eventually write, it is impossible to pick the correct spot that avoids either a hole or an overlap in the output file.

Since the underlying operating system does not provide the necessary interfaces to allow the necessary type of insertion, there are two solutions. The first is to extend the operating system interfaces to provide insert primitives. I chose not to go in this direction because it ties the implementation to a single operating system, but there are advantages to this approach which I cover in Chapter 7. The other option is to build a user-level file system on top of the OS-provided file operations. The user-level file system is contained within a single file, and it can be made OS and machine independent. Thus, Cilk programs using serial append therefore output to a file which is not in standard UNIX format. This strategy certainly has drawbacks, but is not as limiting as it may initially seem. Section 5.4 contains a full discussion of the tradeoffs.

5.2 Initial implementation

This section describes the basic implementation of serial-append in the Cilk runtime system. It describes the Cheerio file format, a structured nonflat file format that allows out-of-order writes but allows the file to be read as a flat serially-ordered file. This moves into a discussion of Cheerio nodes, which are the basic unit of metadata used to preserve the correct ordering of data in both in the runtime and in the Cheerio file, an internal API used to manage them, and a locking protocol that ensures node consistency in a parallel environment. Finally, I present the basic algoritm for serial-append writes which uses the Cheerio node API and file format.

The Cheerio file format

Cilk writes serial-append files in the Cheerio file format, which is structured as a sequence of blocks. Blocks contain either file data or metadata that describes the correct ordering of the data blocks. The Cheerio format allows parallel applications to continuously write data to disk as it is generated, and yet Cheerio maintains enough information to efficiently regenerate the serial ordering of the data.

Each block in the Cheerio file is `CIO_BLOCKSIZE` bytes long, set to 8 kilobytes in the current implementation. A single counter in shared memory, protected by a mutex, points to the next available block in the output file. The counter starts at 1 (leaving the first block free for magic headers) and increments each time a thread requests a new file block. A file data block occupies an entire Cheerio block. Metadata is stored as Cheerio I/O nodes which are packed 8 to a block, similar to how inodes are packed in FFS or LFS [SBMS97] file systems.

Because each thread allocates its own block when it needs space for output data, any number of threads can write to their own private blocks without conflicting with any other threads. Apart from resource contention on the next available block counter, threads can lay down parallel data to disk without any Cilk-induced synchronization. The file metadata, described in the remainder of this section, does require further synchronization.
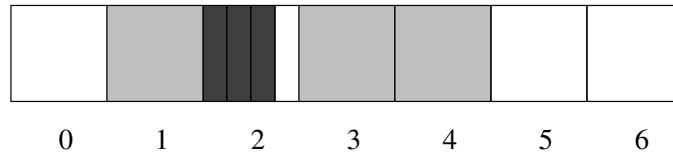
Figure 5-3: A simple Cheerio data file. Blocks 1, 3, and 4 contain file data, and blocks 5 and 6 are empty. Block 2 contains metadata nodes. The next metadata node will go in the last available slot in block 2, and the next data block will occupy block 5. Block 0 of every Cheerio file is reserved for magic headers.
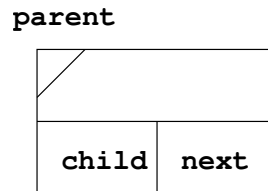

Figure 5-4: A Cheerio I/O node contains three pointers: a pointer back to the node's parent, a pointer to a chain of children, and a pointer to the next node in the node's own chain. The rest of the node contains block metadata.

File metadata

Cheerio breaks the output of a parallel program into what are called Cheerio nodes. Since all of the data assigned to a node appears sequentially in the final output file, the problem is reduced to properly ordering the nodes. Figure 5.2 shows an empty Cheerio node.

A program begins execution as one thread T_1 running on one processor, say P_1, which has an associated Cheerio node N_1. All of the program's output at this point goes into N_1, which is a short way of saying that the data itself immediately goes to the next available Cheerio file block, and the information about where that block is located is written into N_1. Nothing special needs to happen when T_1 spawns T_2. Cilk is designed such that the spawn is little more than a function call, and so T_1 is effectively suspended while P_1 runs T_2. Since this execution corresponds exactly to the serial elision execution, thread T_2 can continue using the same Cheerio node N_1 for its output. If T_2 returns, thread T_1 resumes execution at the point immediately

following the spawn, and continues to write data into N_1. Indeed, when running on one processor, Cheerio uses exactly one Cheerio node for the entire execution of the program, incurring negligible overhead in the limiting case.

Complications arise when processor P_2 steals work. Once thread T_2 is spawned, thread T_1 can be stolen by another processor, leading to T_1 and T_2 running concurrently. To maintain the correct ordering of the file, Cheerio creates a new Cheerio node N_2 during the steal operation, associating it with the newly resumed T_1 running on P_2. Now, T_1 and T_2 can concurrently write data into their private Cheerio nodes without synchronizing against each other. The runtime system preserves the order by adding N_2 to the head of N_1's children chain. The serial order of the file is recovered by reading data from N_1 followed by data in N_2. In general, whenever an idle processor steals a suspended thread, it creates a new Cheerio node, inserts it at the head of the victim's children chain, and then both threads proceed in parallel. Recovery of serial order begins with reading data from a node, then reading data from the nodes in the children chain (following the `child` pointer), then reading data from the nodes in the parent's own chain (following the `next` pointer). Figure 5.2 shows an example of a Cheerio I/O node tree generated with these rules.

When a processor exhausts its work queue, it releases its active Cheerio node. The exact sequence depend on the strategy. In some cases the node may be immediately written to the Cheerio disk file. In other cases the node may be folded into another node, or kept around in memory in hopes that it can be coalesced with another node in the future. Sections 5.2 and 5.3 discusses the two strategies.
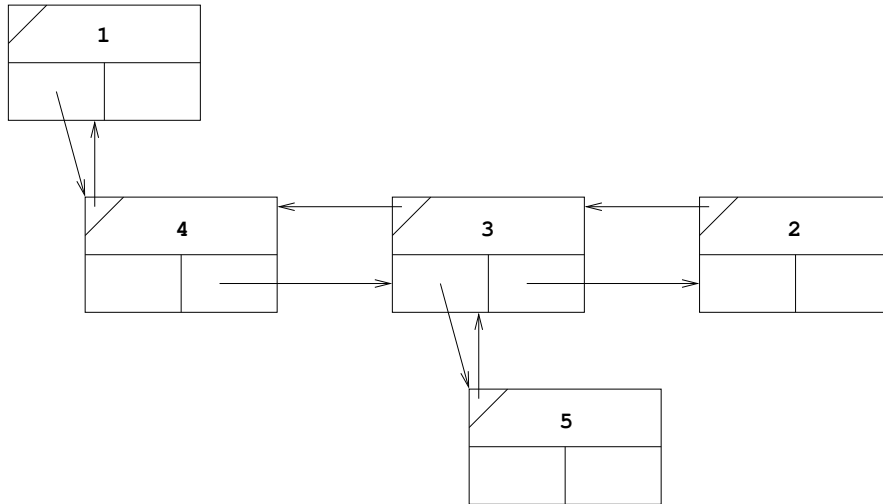
Figure 5-5: A simple example of a tree of Cheerio nodes. Node 1 was the initial node. Nodes 2, 3, and 4 were created in succession each time a new processor stole work from the processor running node 1. Node 5 was created when a processor stole work from node 3's processor. The correct serial ordering of the data is obtained from reading the data in nodes 1, 4, 3, 5, and 2 in succession.

The Cheerio node is defined in the Cilk runtime library as follows:

```
struct cio_node_t {
        struct cio_node_t *parent;              /* my parent */
        struct cio_node_t *child;               /* my first child */
        struct cio_node_t *next;                /* my next sibling */
        struct cio_buf_chain *data;             /* my file data */
        struct cio_buf_chain *curdata;          /* my current chain */
        int chain_offset;    /* first free slot in current chain */
        int child_number;    /* which child i am of my parent */
        int cur_block;       /* my current file I/O block */
        int cur_offset;      /* my current offset in that block */
        int len;             /* my data size + data size of children */
};
```

Cheerio keeps track of data in disk blocks with `cio_buf_chain` structures:

```
#define CIO_BUF_CHAIN_LENGTH 124

struct cio_buf_chain_t {
        struct cio_buf_chain_t *next;
        int blockno[CIO_BUF_CHAIN_LENGTH];
};
```

Cheerio node API

The runtime system manipulates Cheerio nodes using the following internal API. Neither this API nor the Cheerio nodes it manipulates is exposed to the Cilk programmer. The implementation of the node API forms a large part of the whole serial-append implementation, and the enhancements of serial-append described in Section 5.3 of this chapter come from changing pieces of this API's implementation.

- `create_io_node`: Create a new Cheerio node.

  When a worker steals, it creates a new Cheerio node for each open serial append file descriptor. The node is created as the first child of a specified parent node, and is inserted as the first entry in the parent's child chain. Once created, a worker can send output to this Cheerio node.

- `remove_io_node`: Delete a "useless" node from a node tree.

  If a node is no longer in use (has been closed), contains no data buffers, and contributes nothing to the structure of the node tree, then it can be removed from the tree. If the node is a leaf, then deleting it is trivial. If it is in the middle of a parent's child chain, then the node is removed from the chain. If the node is the first in the child chain, `remove_io_node` must reassign "first-child" status to the new first child.

- `close_io_node`: Close an Cheerio node from further use.

  Once a worker runs out of work to do, it closes its node. At a minimum, this function just indicates to the runtime system that no further nodes will be added to the child chain and no more data will be associated with this node. Some strategies may choose to do more with the node at this point, including sending it to disk or attempting to merge it with other nodes. Other strategies may defer further processing of the node.

- `commit_io_node`: Commits Cheerio node to disk.

A closed Cheerio node can be committed to disk, which moves the node and its buffer chains to the first free node slots in the output file. Disk nodes are not allowed to point back into memory, but since in-core nodes can point to nodes on disk, this procedure updates the committed node's child, next, and parent nodes to point to the specified node's new disk location.

Node locking protocol

Since I/O nodes are accessed by multiple processors, there must be a locking protocol that preserves the consistency of the node tree without introducing deadlocks. The need for locking in this scenario is quite similar to the needs of several concurrent B-tree implementations, discussed in [Wan91]. Cheerio uses a novel locking paradigm which is provably correct and easy to implement.

We preserve consistency by stipulating that if a node's pointer moves, then both that node as well as the node being pointed to must be locked before updating the pointer. This protocol ensures that the destination of the pointer hasn't moved from underneath the Cheerio runtime while the node is repointed. If a node itself is moving, that requires its parent to repoint at it, so the node and its parent must both be locked. Moreover, if the node has a valid child or next pointer, that pointer, and hence the pointer's destination, must also be locked.

To avoid deadlocks, we establish an ordering of the I/O nodes to avoid deadlocks. Luckily, there is already an ordering to the nodes, namely, the order defined by the correct serial-append order of the node's data. Thus, the runtime system cannot request a lock on a node if it already holds a lock on one of that node's children. Workers cannot deadlock against each other under this scheme, because there is one serial ordering to any node tree.

Hence, the locking protocol for moving `node` to disk is as follows:

1. Follow `node`'s parent pointer, and lock `node`'s parent.

2. Lock `node`.

3. Write `node` to disk.

4. Change `node`'s parent to point to new on-disk location of `node`.

5. Unlock `node`'s parent.

6. Lock `node`'s child, update its parent pointer, unlock child.

7. Lock `node`'s next, update its parent pointer, unlock next.

The protocol for locking when inserting a node:

1. Lock the node.

2. Lock the child node.

3. Create a new node, and lock it.

4. Adjust the node's child pointer, the new node's next pointer, and the previous child's `CIO_FIRSTCHILD` flag is cleared.

5. Unlock the nodes.

Writing the Cheerio file

The following is the first algorithm the runtime system uses to write Cheerio files in serial-append mode. It uses the Cheerio I/O node API described above. It also adheres to the constraints discussed previously. The key feature (or simplification) of this algorithm is that when each node is closed it is immediately committed to disk. Thus, nodes are never kept in memory once they have been closed, limiting the total number of nodes in memory to the number of active processors. In order to minimize critical path overhead, the Cheerio implementation has low amortized overhead on I/O and does various tasks during steals. It has zero overhead on spawns. The events that require processing are steals, I/O requests, and emptying of a processor's work queue.

This algorithm maintains a Cheerio I/O node tree for each open Cheerio descriptor in serial-append mode. In this algorithm there is no advantage to maintaining separate trees, since each open descriptor has an identical tree structure. More advanced algorithms coalesce nodes together when possible. Which nodes can be combined

depends on the use of each descriptor in each thread, making it advantageous to maintain a separate tree for each descriptor.

The algorithm is described in "event : action" style. For each of the events listed with a bullet, the subsequent paragraph(s) describes the action of the algorithm.

- Processor P steals from processor P_{orig}.

  The worker running on processor P creates a new Cheerio node for each open serial-append descriptor and saves a private pointer to each of them. Each new node's parent points back to the corresponding node for the same descriptor on processor P_{orig}, and each new node becomes its parent's first child by inserting itself at the head of the parent's child chain, setting its CIO_FIRSTCHILD flag, and clearing CIO_FIRSTCHILD from the node previously at the head of the child chain. The create_io_node() API call performs most of this work.

- Processor P empties its work queue with issuing any writes.

  In many cases a processor steals a thread from a victim but never performs any I/O to a given seral-append descriptor before it finishes the thread and clears its work queue. In this case, the Cilk runtime calls remove_io_node() to delete the useless node from the tree. This optimization is so simple that it is included in the basic algorithm. It also reduces the number of I/O nodes and hence the size of the Cheerio output file. In fact, without this optimization it is trivial to write pathological cases that write arbitrarily large output files containing 1 actual byte of output. Things are not quite so bad with normal programs, but the savings can still be significant if many threads don't actually write to the file.

- Processor P makes its first write to the file.

  If processor P has never written to the file, its Cheerio node has a null data field, which normally points at a cio_buf_chain. The buffer chain holds a list of blocks that contain data from writes. The first write to a node causes the runtime system to allocate a new cio_buf_chain. Each buffer chain can hold

1024 buffers, each `CIO_BLKSIZE` bytes big (8096 bytes in the current implementation). Therefore, on the first write, or on a write where the buffer chain is full, the runtime system allocates a new buffer chain before it performs the write itself (see next).

- Processor P writes to the file.

  All data writes go to the disk block and offset held in the Cheerio node's `cur_block` and `cur_offset` fields. The `cur_block` field mirrors the last filled-in entry in the current `cio_buf_chain` table. If the requested write is too large to fit entirely within the current block, then the runtime system fills the current block with as much data from the write as can fit. It then requests a new block for the rest of the data, updates `cur_block` and `cur_offset`, and adds the new block entry to the node's current buffer chain. If the chain itself is full, then the runtime system attaches a new chain to the end of the chain, bumps the `curdata` field in the I/O node to point at the new end of the buffer chain, and resets `cur_block` and `cur_offset`.

- Processor P empties its work queue.

  When a processor finishes its work queue, the runtime system closes all of its open Cheerio nodes by calling `close_io_node()`, since nothing can add more data to its I/O buffers. The simple algorithm then immediately calls `commit_io_node()`, sending processor P's the node and its buffer chains to disk. As part of the commit operation, any nodes in memory that point to P's node are modified to point at the on-disk copy. The node's parent is also modified to repoint the appropriate `child` or `next` field to the node's new location, even if the parent is on disk. It is not necessary to modify parent pointers of on-disk nodes, because the on-disk structure is always rooted with the first node and can be fully traversed from that point.

The serial I/O of a node can be constructed by starting with the node's own I/O, followed by each child's I/O, starting from the current child pointer and working through the chain to the first allocated child.

5.3 Improved implementation

The improved implementation is based on the "collapsing" algorithm, which gets its name from its attempt to write a smaller Cheerio node tree to the output file by collapsing nodes together. All of the changes from the simple algorithm presented above are in the stage corresponding to processor P emptying its work queue, which is replaced with the following action.

- Processor P empties its work queue.

  As in the simple algorithm, the Cilk runtime immediately closes all of processor P's nodes by calling `close_io_node()`. Suppose the current node is designated with N. Now, instead of just committing node N to disk, the runtime system tries to merge N into its parent or sibling. The algorithm consists of the following steps:

  1. Examine each of N's children, beginning with the highest-numbered child. The highest-numbered child is first in the child chain and corresponds to the most recently stolen thread. If the child node (which we call N_c is in memory and marked as closed, append its block list to N's block list, prepend its child node chain to N's child node chain, and delete N_c. Repeat with the next child node until the traversal reaches a node that is not closed or has been written to disk.

     This step is safe because if N is closed, no new data or children can be added to its child chain. If N_c is also closed, then no new data or children can be added there either, which means the two nodes can be combined. If N_c has children itself, those can be inserted in N's child chain, because this operation preserves the serial data ordering.

  2. Follow N's `parent` pointer (call this node N_p). If N_p is open or if it is on disk, stop here.

  3. If node N_p is closed and in memory, examine its `cur_child` field. If `cur_child` points to N, then restart the merge algorithm on N_p.

This step can merge N with its siblings, because they are all children of N_p. It is safe because it begins with a closed node, just as in Step 1.

4. If the total number of nodes in memory exceeds some threshold `max_in_core_nodes`, commit N to disk. The parameter `max_in_core_nodes` is settable at compile time.

The collapsing algorithm is guaranteed to terminate. Here is a simple proof. Each step of the algorithm can cause one of three things to happen. It can merge two nodes, thus decreasing the total number of nodes in the tree by 1; it can cause the algorithm to terminate; or it can restart the algorithm at one level higher in the tree. Since the tree has a finite number of nodes and has finite depth, eventually the algorithm either exits, reaches a node with either an open parent or no parent, or collapses the entire tree into one node.

The algorithm contains the adjustable parameter `max_in_core_nodes` which indicates when to commit nodes to disk. The limiting case of the collapsing algorithm has no threshold. Thus, the entire metadata tree is kept in memory, and once all nodes are closed, it provably collapses to a single node which gets written to the Cheerio file. This strategy gives an efficient file from which to read, but at the unacceptable cost of keeping metadata in memory for too long. Empirically, setting the threshold to be 4 times `nproc` seems to work well. A topic of future research is characterizing a more precise relationship between the chosen threshold and resulting output file size and efficiency.

5.4   Reading the Cheerio file

The output from a serial-append Cheerio file is not in flat format; the file contains out-of-order data, plus the necessary metadata to derive the serial order. The implementation of Cheerio for Cilk attempts to write Cheerio files that can be read efficiently, first by minimizing the number of nodes, and second by setting the length field in each node. This section describes three ways to read the data from a Cheerio

file. Serial conversion is the simplest and most efficient procedure on a single-processor machine. Parallel conversion can take advantage of multiple processors to generate a flat serial file in less time than the serial algorithm. Lastly, the Cheerio API can read Cheerio files directly, so any program that uses Cheerio for its I/O can transparently read serial-append files. This section then discusses the advantages of writing a structured file instead of having Cheerio programs using serial-append write serial data to flat files at runtime.

Serial file conversion

The simplest way to recover the serial order of data in a Cheerio file is to sequentially walk through the Cheerio nodes. As explained above, the data is recovered by appending the data belonging to a node to the output stream, then recursively walking the node's child chain, then recursively walking the node's next chain. This method is reasonably efficient: each segment in the Cheerio file is visited once, and the overhead is determined by how many metadata nodes are in the file.

Parallel file conversion

The Cheerio file can also be converted into a flat serial file in parallel, using a conversion program I wrote in Cilk. It is structured similarly to the serial converter, but it uses the `length` field in the Cheerio node to allow parallel conversion. The runtime system sets a node's `length` to be the number of bytes contained in the node, plus the lengths of all the node's children. Consequently, a converter program can walk the child chain and the next chain in parallel, because it knows precisely how much data the child chain contains by consulting the `length` field. Figure 5-6 shows the parallel walking algorithm. The parallel converter does not require any fewer disk accesses, but if the Cheerio output file is still in the operating system's buffer cache or if the file resides on a RAID, then converting in parallel can be faster.

```
walk_node(node, flat_file_offset)
{
        int res;

        res = output_node(node, flat_file_offset);   /* write node's buffers to
                                                        output file */

        /* write the children immediately after the data from this node.
        /* use the supplied offset + the length just written. */

        spawn walk_node(node–>child, flat_file_offset + res);

        /* simultaneously, start writing the next node */
        /* skip the length that node + its children needs */

        spawn walk_node(node–>next, flat_file_offset + node–>len);

        sync;
}
```

Figure 5-6: Parallel pseudo code for flattening Cheerio files.

Reading using Cheerio API

Finally, Cilk program can read data directly from a Cheerio file by using the normal
`read()` interface. The Cheerio implementation recognizes that it is reading from a
Cheerio file, and again uses the `length` field to walk down the node tree, find the
correct segment of data, and return it to the user.

Writing an intermediate Cheerio file format which is not flat is not as undesirable
as it may initially seem. In particular, writing a Cheerio file that isn't flat reduces
program execution time, which can be beneficial in some cases. One such example is
a computer chess program, which might search millions of board positions in parallel.
Suppose the programmer wishes to keep a log of the evaluated board positions. She
uses Cheerio's serial append mode, because she wants to have the search tree logically
ordered instead of in some random execution-dependent order. The log is not the
primary output of the program, though. During the execution, the only required
output is a single move, and in the case of a computer chess player, that output
should be produced as quickly as possible. Cheerio's file system is well-suited for this

57

set of priorities, because it supports the fastest possible execution time while recording enough information to efficiently recover the serial output order. An implementation that writes a flat output file would necessarily require more execution cycles, which might adversely affect the number of positions the chess program could search in a give time.

Chapter 6

Benchmarks

This chapter presents some performance analysis and benchmark results. Section 6.1 discusses the sources of overhead in Cheerio operations. Table 6.1 summarizes the results of this analysis, which shows that read and writes are low-overhead operations and opens and closes carry higher overhead. Section 6.2 presents the performance impact of Cheerio on a program that stresses I/O overhead. Table 6.2 contains the results. Two items from this section are most important: the effect of each Cheerio mode on the speedup of a sample application and the overhead Cheerio imposes in the single processor case. Finally, Section 6.3 describes serial-append's use in the `queens` and `compress` programs.

6.1   Sources of overhead in Cheerio

The implementation of Cheerio in the Cilk runtime system is an added layer between the program and the POSIX API calls, so Cheerio operations carry with them some amount of overhead compared to the corresponding POSIX operations. This section lists those overheads for each mode, pointing out how most of the overhead has been pushed into the rarer operations, thereby keeping the frequent reads and writes fast.

All of the overhead in Cheerio occurs in one of three classes of operations. The first are opens and closes, which we assume happen infrequently. The second are reads, writes, and pointer seeks, which we assume can happen with great frequency.

The third are steals, whose frequency depends on the number of processors and the structure of the program.

In all three modes, a Cheerio `open()` call translates into a POSIX `open()` call, plus some additional time to set up shared data structures. All three modes require a `cio_info_t` structure, and serial-append requires an initial Cheerio I/O node. There is little overhead when opening a file, but more importantly, opens don't occur very often. Programs that open files before each write are already inefficient; the entire model of POSIX I/O is based on the idea that an `open()` call is slow and need only be done once per file. The same is true for `close()`, which frees some shared structures and calls the native `close()`. In serial-append mode, `close()` also flushes any remaining Cheerio nodes to disk. A compile-time constant limits the number of Cheerio nodes that can stay in memory at any time, so there is a limit to the amount of overhead incurred by a `close()`.

Each mode has different overhead for reads and writes, but they all share the need to reopen files on different worker processors. The reopen happens on demand during the first `read()` or `write()` issued on the new processor. Reopening the file requires a POSIX `open()` and `lseek()`, which entail significant overhead. On the other hand, a file can only be reopened as many times as there are processors, so in the case where the number of reads and writes is much larger than the number of processors, the amortized overhead of the extra opens is small.

Global mode has the highest read and write overhead. Each read and write first acquires a lock, sets the seek pointer if necessary (which requires an `lseek()` call, then issues a POSIX `read()` or `write()` call, updates some internal state, and finally releases the lock. If one processor is doing most of the operations on a file descriptor, then the seek is typically not needed, since the runtime system remembers which processor moved the pointer most recently and only adjusts it if it isn't the same as the processor issuing the current request. Even in the single processor case, though, acquiring the lock incurs some amount of overhead.

Local mode is the opposite; it has near-zero overhead on reads and writes. Each Cheerio call immediately translates into a POSIX `read()` or `write()` call followed

by an update of the private copy of the file pointer. Nothing more happens on reads and writes. Local mode is intentionally as lightweight as possible.

Serial-append mode also does quite well on writes (serial-append does not support reads). The structure of the Cheerio I/O nodes is not affected by individual writes, so the only overhead is occasionally acquiring a new Cheerio file block (requiring a shared lock), and periodically allocating new buffer chains (which is an entirely local operation). Cheerio data blocks are 8192 bytes, so every 8KB of writes requires a new file block. Amortized against the typical write size, the overhead is low. If an application does much larger writes, it is possible to change the Cheerio block size to something more appropriate, but the tools to read the Cheerio file do not yet support dynamic block sizes.

The last relevant operation is the steal. Steals are not technically Cheerio operations, but Cheerio maintains some bookkeeping information across steals that leads to added overhead. Importantly, steals are already high overhead operations in the Cilk runtime. Cilk optimizes the more frequent `spawn()`, and in return slows down the much rarer steal. Global mode requires almost no added overhead on the steal. Since each read or write needs to update the pointer anyway, there's no work to be done while stealing, except to invalidate the "last processor" flag for each open descriptor. Local mode requires a call to the internal function `cio_reload_local_pointers()` to copy the cached pointer from the victim to the stealing processor. This function implements the semantics of preserving the pointer in either the parent or the child of a spawn. It carries medium overhead, since for each descriptor it copies a value and then issues an `lseek()` to bring the new processor's descriptor in line with the saved pointer location.

Serial-append has the highest and most variable overhead on steals. All of the operations that affect the structure of the Cheerio tree take place during steals. At the least, a steal requires closing the previous Cheerio node for the processor and creating a new node. Some nodes may eventually be deleted, but every node that is not deleted is eventually committed to the file. The commit itself may need to rewrite up to two other nodes already committed on disk, so at worst a single commit writes

61

Mode	Local	Global	Serial-append
read/write overhead	very low	high	low
open/close overhead	low	low	low
steal overhead	high	low	high

Table 6.1: Summary of the overhead in Cheerio operations.

a node to the Cheerio file and writes two 16-byte values to other places in the file. Since only a fixed number of nodes can remain in memory at any time, each spawn on average creates one node and commits one node.

To summarize the above discussion, Table 6.1 summarizes the amount of overhead for each operation of each mode.

6.2   Scaling with the number of processors

The first important performance measure is how well programs that use Cheerio scale with the number of processors. This section describes the performance impact of using each Cheerio mode by benchmarking `dag-write`, a program that stresses the I/O overhead of a parallel system. It stresses the favorable scaling properties of serial-append and local modes, and it examines the performance on a single processor.

I compare the speedups of three different versions of a program called `dag-write`, which creates a deeply nested spawn tree where each leaf node performs some I/O.[1] The first version is the normal program using Cheerio for its I/O. The second version is the same, but with the Cheerio calls replaced with their corresponding POSIX calls. Certainly the output of this program is no longer correct, but comparing it to the normal program shows how much added overhead Cheerio entails. The third version of the program is the same, but with all of the I/O calls stripped out entirely.

The added Cheerio overhead varies according to Cheerio mode, as would be expected. Local mode adds the least overhead. Global mode, because of its need to acquire a lock and perform an extra seek on each I/O operation, has the greatest. One interesting result is that the overhead does not seem to vary significantly across

---

[1]Although the program is named `dag-write`, it tests both read and write performance.

different programs. The lack of variation makes sense, because the Cheerio overhead is contained two three events: opens, reads and writes, and steals.

A particularly important piece of this benchmark is the behavior when running on one processor. Running on a single processor measures the overhead of Cheerio where Cheerio's added benefits are useless. Much as Cilk incurs near-zero overhead on one processor, the Cheerio modes also incur near-zero overhead, with the exception of global mode. It would be a simple matter to detect the single processor case and not do the unnecessary locking and seeking, but that hides the fundamental inefficiency of global mode. Indeed, given how poorly global mode performs, there is little incentive to optimize certain cases of it, lest programmers use it when they are better off restructuring their code to avoid global mode in the first place. As expected, local mode scales well to one processor, since it has little runtime overhead under any number of processors. Serial-append mode also scales well. When there is only one worker process, there are never any steals, and so all writes go into a single Cheerio I/O node that covers the entire file. The overhead for serial-append writes is very low; nearly all of the work happens during a steal when a node is committed and a new node created.

Table 6.2 lists the overhead of each mode as a percentage above the running time of a program using normal POSIX I/O. This overhead measures the added work of using Cheerio, not the time required to do actual disk I/O. Unlike other parallel file I/O solutions, Cheerio makes no attempt to accelerate the actual writing of data to disk. The program benchmarked here creates a deep spawn tree and performs I/O at each leaf. It is intended to be a worst-case example of Cheerio, since there is little work other than the I/O itself.

6.3 A closer look at serial-append mode

Serial-append mode is particularly suited for parallelizing applications. Because it is so different from previously available paradigms for I/O, modifying programs to use serial-append falls into two classes: either changing a parallel program to use

Number of processors	Global	Local	Serial-append
1	10	2.2	2.0
2	20	13	9.1
3	32	17	17
4	47	20	24
5	63	21	32
6	88	21	47
7	102	22	60
8	121	21	74

Table 6.2: The overhead for each Cheerio mode, expressed as a percentage increase in time required to output using Cheerio instead of the raw POSIX calls.

serial-append, or using serial-append to help parallelize a serial program that needs to preserve ordered output. This section describes porting two programs to serial-append mode. The first adds logging to `queens`, an already parallelize program. The second parallelizes `compress`, a serial program.

Adding serial-append to an already parallelized program

Adding serial-append to an already parallelized program is almost exclusively for adding logging or tracing to a parallel program. Since there really aren't any parallel programs today that expect serial ordering of their output, any preexisting parallel program does not contain serially ordered I/O operations. The program might be buffering data or otherwise using some bizarre technique for ordering data, in which case using Cheerio's serial-append mode would require completely reworking that aspect of the program. Alternatively, and more likely, the program will not be doing any ordered I/O, so using serial-append would change the purpose of the program. Consequently, it is most useful for adding logging or tracing to a program.

As an example of this technique, I added tracing to the `queens` program that comes with the Cilk 5.2 distribution [cil98]. The `queens` program solves the classic problem of placing n queens on an $n \times n$ board such that no two queens attack each other. The version modified with Cheerio outputs a log of the visited board positions. By opening the file in local mode, the output is a log of the positions in the order

they were searched. Changing the mode to serial-append yields a list of positions in lexicographically sorted order, which is the order the program spawns threads.

Parallelizing a serial program with the aid of serial-append

The real strength of serial-append mode is that it allows applications that depend on serially ordered data to be parallelized. My example of this is the UNIX compression program `compress`, which compresses normal files. This turned out to be an appealing target for parallelization: the code is written with few globals that must be restructured or mutexed, and file compression is done on independent chunks of the file. Using serial-append mode means few changes were needed to the I/O calls. Performance of `compress` scales well, with a speedup of 1.8 on 2 processors and 3.75 on four processors.

Chapter 7

Future Directions

Cheerio serves an important role in parallel programming. It allows programmers to write efficient parallel programs that run under an environment similar to traditional POSIX/UNIX operating systems rather than a specialized parallel environment that can be more difficult to target. Cheerio's serial-append mode is particularly interesting, because it opens the door to parallelizing a host of applications that were once "off-limits" due to their need for serially ordered output. This chapter discusses several natural extensions to the work presented in the previous chapters.

7.1 Improvements to existing implementation

There are several improvements that could be made to the existing implementation of Cheerio in the Cilk runtime library.

- Short write buffers in Cheerio nodes

  Each Cheerio node is 1024 bytes, but most of this space is reserved for future expansion. Some of it could be used to implement a "short write" buffer which would hold the first, say, 512 bytes written to a Cheerio node. In some cases this could eliminate the waste of an entire block plus an extra buffer chain, leading to a smaller Cheerio file. Initial tests with this show a few percent reduction in size when using the `compress` benchmark, but the system needs to be updated

to read these extended Cheerio files.

- Port Cheerio to a different threading environment.

  As discussed before, Cheerio is intended to be portable to other parallel environments. An implementation of Cheerio under some other environment would verify that its design is sufficiently independent of Cilk's design to be successful as a generic API for parallel file I/O. The Pthreads standard, described in [But97] and elsewhere, is a popular thread interface available on most UNIX operating systems. Many popular programs like Apache and BIND can make use of Pthreads as a lower-overhead way of achieving concurrency than by forking multiple processes. It is unclear whether there is a user-space implementation of serial-append mode under Pthreads that can be as efficient as the implementation for Cilk, because thread schedulers tend to mimic kernel schedulers by preempting one thread and replacing the execution context with that of a new thread. At a first glance, such a model loses some of the properties of the Cilk environment that allow an efficient implementation, most importantly that most spawns of new threads never actually cause any concurrency, and thus do not require runtime overhead to maintain serial ordering.

7.2 Moving Cheerio into the operating system

One "threading" environment that is an obvious target for Cheerio is a UNIX kernel itself. A kernel implementation of Cheerio has several important advantages. It enables concurrency at the process and kernel thread level, which is still a very popular mode for introducing concurrency in applications. It also moves the Cheerio file metadata support in the kernel, where it can take advantage of the preexisting buffer cache mechanism. There would also be no need to expose the Cheerio file format to the user. Any application reading the file with `read()` would see the flat version; any conversion from the hierarchical Cheerio format would happen transparently.

Some work has been done in this direction, but not in the context of parallel

file I/O. The Zodiac project [NCS97, CMNY98] has built a media file system that supports efficient inserts.

7.3   Research of theoretical interest

In addition to the systems projects described above, there are three areas of theoretical research that may prove fruitful.

Describing the Cheerio file format

The structure of the Cheerio file's metadata is determined at runtime by the pattern of steals. This can lead to different files corresponding to the same serial order. A better understanding of the metadata tree's depth, node count, and structure would prove useful for making guarantees on how efficiently data at arbitrary offsets in the file can be read. Similarly, there may exist better algorithms that can generate a more consistent or efficient tree; the current algorithm is more heuristic-based than one might like.

Fully distributed locking

Global and serial-append modes both require centralized locks on certain system resources. It is difficult to imagine how one would eliminate locking in the global case where every thread needs to share the same pointer, but in serial-append mode, there may be alternative Cheerio file formats that do not require the central block allocation lock. Eliminating contention on centralized locks can improve performance, but further benchmarks are necessary to ascertain the potential performance benefit from such a change.

Additional parallel modes

In addition to global, local, and serial-append modes, there may be other useful semantics worth supporting in a general parallel API framework. Some of these may

mirror other APIs, such as MPI-IO's support for structured (strided) I/O patterns. Others may be yet different behaviors for the shared pointer besides those already implemented. Even further out is the possibility of some paradigm vastly different from the flat file with pointer approach that is so entrenched. Perhaps some sort of record-based approach would mesh well with Cheerio's hierarchical format.

Bibliography

[AA86]   Kenneth Arnold and Elan Amir. Screen updating and cursor movement optimization: A library package. In *4.4BSD Programmer's Supplementary Documents*, chapter 19. April 1986.

[But97]   David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley, Reading, MA, USA, 1997.

[CF94]   Peter F. Corbett and Dror G. Feitelson. Design and implementation of the Vesta parallel file system. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 63–70, 1994.

[CFF+96]   Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong. Overview of the MPI-IO parallel I/O interface. In Ravi Jain, John Werth, and James C. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, volume 362 of *The Kluwer International Series in Engineering and Computer Science*, chapter 5, pages 127–146. Kluwer Academic Publishers, 1996.

[cil98]   Supercomputing Technology Group, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, Massachusetts 02139. *Cilk-5.2 (Beta 1) Reference Manual*, July 1998. Available on the World Wide Web at `http://supertech.lcs.mit.edu/cilk`.

[CMNY98]   Tzi-cker Chiueh, Tulika Mitra, Anindya Neogi, and Chuan-Kai Yang. Zodiac: A history-based interactive video authoring system. In *Proceedings of ACM Multimedia '98*, Bristol, England, September 1998.

[DJT96]   Dannie Durand, Ravi Jain, and David Tseytlin. Improving the performance of parallel I/O using distributed scheduling algorithms. In Ravi Jain, John Werth, and James C. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, volume 362 of *The Kluwer International Series in Engineering and Computer Science*, chapter 11, pages 245–270. Kluwer Academic Publishers, 1996.

[FLR98]   Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN*

'98 Conference on Programming Language Design and Implementation (PLDI), pages 212–223, Montreal, Canada, June 1998.

[Fri98]   Matteo Frigo. The weakest reasonable memory model. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1998.

[GNU]   GDBM: The GNU database manager. Available on the Internet from `http://www.gnu.org/software/gdbm/gdbm.html`.

[HER+95]   Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM Conference on Supercomputing*, pages 385–394, Barcelona, July 1995.

[JWB96]   Ravi Jain, John Werth, and James C. Browne. *Input/Output in Parallel and Distributed Computer Systems*. Kluwer Academic Publishers, 1996.

[MBKQ96] M. McKusick, K. Bostic, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, 1996.

[MPI97a]   MPI-2: Extensions to the message-passing interface, 1997. Available on the Internet from `http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html`.

[MPI97b]   MPI: A message-passing interface standard, May 1997.

[MS94]   Steven A. Moyer and V. S. Sunderam. PIOUS: A scalable parallel I/O system for distributed computing environments. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.

[MS96]   Steven A. Moyer and V. S. Sunderam. Scalable concurrency control for parallel file systems. In Ravi Jain, John Werth, and James C. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, volume 362 of *The Kluwer International Series in Engineering and Computer Science*, chapter 10, pages 225–244. Kluwer Academic Publishers, 1996.

[NCS97]   T. N. Niranjan, Tzi-cker Chiueh, and Gerhard A. Schloss. Implementation and evaluation of a multimedia file system. In *IEEE International Conference on Multimedia Computing Systems*, Ontario, Canada, July 1997.

[NK96]   Nils Nieuwejaar and David Kotz. Low-level interfaces for high-level parallel I/O. In Ravi Jain, John Werth, and James C. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, volume 362 of *The Kluwer International Series in Engineering and Computer Science*, chapter 9, pages 205–223. Kluwer Academic Publishers, 1996.

[PCGK89]   D. A. Patterson, P. Chen, G. Gibson, and R. H. Katz. Introduction to redundant arrays of inexpensive disks. *Digest of Papers of COMPCON Spring '89*, 1989. IEEE Computer Society Press.

[POS90]   *System Application Program Interface (API) [C Language]*. Information technology—Portable Operating System Interface (POSIX). IEEE Computer Society, 345 E. 47th St, New York, NY 10017, USA, 1990.

[POS93]   *Portable Operating System Interface (POSIX)—Part 1: System Application Programming Interface [C language] (Volume 1)*. Information technology—Portable Operating System Interface (POSIX). IEEE Computer Society, 345 E. 47th St, New York, NY 10017, USA, 1993.

[Ran98]   Keith H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

[SBMS97]   Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. The design and implementation of the 4.4BSD log-structured file system. In *Proceedings of the 1993 Winter USENIX*, San Diego, California, January 1997.

[SC00]   National Science and Technology Council. *High Performance Computing and Communications: Information Technology Frontiers for a New Millenium*. 2000.

[SSA+94]   Craig B. Stunkel, Dennis G. Shea, Bülent Abali, Monty M. Dennean, Peter H. Hochschild, Douglas J. Joseph, Ben. J. Nathanson, Michael Tsao, and Philip R. Varker. Architecture and implementation of Vulcan. In *International Parallel Processing Symposium*, pages 268–274, April 1994.

[Wan91]   Paul Wang. An in-depth analysis of concurrent B-tree algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 1991. Also available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-496.