The Theory and Practice of I/O-Efficient Computation

by

Darren Erik Vengroff

B.S.E. Princeton University, 1989

Sc.M. Brown University, 1993

Thesis

Submitted in partial fulfillment of the requirements for the

Degree of Doctor of Philosophy in the Department of Computer Science

at Brown University.

May 1997

This dissertation by Darren Erik Vengroff
is accepted in its present form by the Department of
Computer Science as satisfying the
dissertation requirement for the degree of Doctor of Philosophy.

Date 4/12/97 [illegible]
Jeffrey Scott Vitter

Recommended to the Graduate Council

Date 4/22/97 [illegible]
Roberto Tamassia

Date 4/16/97 Thomas H. Cormen
Thomas H. Cormen

Approved by the Graduate Council

Date 5/3/97 [illegible]

This thesis is dedicated to the memory of Paris C. Kanellakis. He was a teacher, a colleague, and a friend.

Paris' untimely death in December of 1995 touched all of Computer Science. More importantly, however, we were touched by his remarkable life, and are better for having been.

Acknowledgments

First and foremost, I thank my wife, Hilary Hoover, for her constant support and encouragement during the years I spent researching and writing this thesis. During that time we followed one another's careers back and forth across the country–from Seattle to Storrs, to Providence, Boston, Durham, and Ann Arbor, back to Seattle, and back to Ann Arbor. Through it all, Hilary has always been there whenever I needed her.

I also received a tremendous amount of support and encouragement from my parents Linda and Richard, my stepmother Cindy, and my sisters Lisa and Marriah. Their expectations of success have always been the highest but their love and understanding have always remained unconditional.

I thank Jeff Vitter, my advisor, for first pointing me towards I/O and for guiding me along the path that led to this thesis. The other present and former members of my thesis committee, Roberto Tamassia, Tom Cormen, and the late Paris Kanellakis also provided many helpful comments and suggestions, both on my early work and on this thesis itself.

Many colleagues contributed to the development of the ideas presented herein. Some of these were co-authors on papers in which many of the ideas in this thesis were originally presented. These include Lars Arge, Yi-Jen Chiang, Mike Goodrich, Eddie Grove, Jyh-Jong Tsay, Roberto Tamassia, and Jeff Vitter. Others who provided helpful technical discussions of various aspects of the work include Rakesh Barve, Tom Cormen, Philip Klein, David Kotz, Peter Bro Miltersen, T. M. Murali, Mark Nodine, John Savage, Elizabeth Shriver, and Len Wisniewski.

During my time as a graduate student, I had the opportunity to travel extensively outside of Brown, both in physical space and in cyberspace. I was graciously hosted by a number of individuals and institutions, including the Department of Computer Science at Duke University, Yale Patt and the HPS group in the Department of Electrical Engineering and Computer Science at the University of Michigan, Ed Lazowska and the Department of Computer Science and Engineering at the University of Washington, Mitch Alsup and Eric Sprangle at Ross Technology, and Tom Cormen and David Kotz at Dartmouth College.

In addition to my official hosts during my travels I had a number of unofficial hosts who generously opened their homes to me. These include T. M. Murali, and Jeff, Sharon, Jillian, Scott, and Audrey Vitter in Durham, and Eric Sprangle and David, Amy, Alexander, and Evan St. Angelo in Austin.

Last, but certainly not least, I thank all those who helped me successfully navigate the administrative and financial seas one must inevitably cross to complete a Ph.D. These include Mary Andrade, Trina Avery, Dawn Nicholaus, and Jennet Kirschenbaum at Brown, Cathie Caimano, Ellen Denham, the late Anna Drozdowski, Tina Gaither, Sondra Haithcock, Denita Thomas, and Kathleen Williams at Duke, and Michelle Chapman at Michigan.

Contents

Acknowledgments iv

I Fundamentals 1

1 Introduction 2

1.1 Hierarchical Memory 3

1.2 A Brief History of I/O-efficient Computation 5

1.3 Related Work 7

1.3.1 Algorithmics 7

1.3.2 I/O-Efficient Computation Systems 8

1.4 Contributions of this Thesis 13

2 Models and Mathematical Preliminaries 16

2.1 Modeling the Behavior of Disks 16

2.2 The I/O Model of Computation 18

2.3 The Parallel Disk Model 20

2.4 Paradigms of I/O-Efficient Computation 23

2.4.1 Scanning 23

2.4.2 Distribution and Merging 26

2.4.3 Block Linking and Meta-data 29

3 **Permutation and Sorting** 32

3.1 Lower Bounds: Linear Complexity vs. Permutation Complexity 32

3.2 Sorting 34

3.3 Comparing Independent Disk Sorting and Striped Disk Sorting 37

II **Algorithms** 40

4 **Combinatorial Algorithms** 41

4.1 Geometric Algorithms 41

4.1.1 Distribution Sweeping, a Generalization of Plane Sweeping . . . 42

4.1.2 Batch Filtering 54

4.1.3 Convex Hull Algorithms 60

4.2 Graph-Theoretic Algorithms 64

4.2.1 PRAM Simulation 65

4.2.2 Generic Simulation of an $O(N)$ Space PRAM Algorithm 65

4.2.3 Simulation of Super-Linear Space PRAM Algorithms 67

4.2.4 Reduced Work Simulation for Geometrically Decreasing Computations 68

4.3 List Ranking 70

4.4 Conclusions 72

5 **Algorithms for Scientific Computation** 74

5.1 Sparse Matrix Methods 74

5.2 Dense Matrix Methods 78

III **TPIE** 87

6 **TPIE, a Transparent Parallel I/O Environment** 88

6.1 The Structure of TPIE 89

6.2 Streams 91

6.3 Scanning 92

6.3.1 Basic Scanning 92

6.3.2 ASCII Input/Output 97

6.3.3 Multi-Type Scanning 98

6.3.4 Out-of-Step Scanning 99

6.4 Merging 101

6.4.1 Implementing Mergesort: An Extended Example 104

6.5 Distribution 108

6.6 Permutation 108

6.6.1 General Permutation 108

6.6.2 Bit Permutation 110

6.7 Sorting 111

6.7.1 Comparison Sorting 111

6.7.2 Key Bucket Sorting 113

6.8 Matrix Operations 113

6.8.1 Dense Matrix Operations 114

6.8.2 Sparse Matrices 115

6.8.3 Element-wise Arithmetic 116

6.8.4 Sparse Matrix by Vector Multiplication 117

6.9 Conclusions 118

7 Additional Examples of TPIE Implementations **119**

7.1 Convex Hull 119

7.2 List-Ranking 127

7.3 Conclusions 139

8 TPIE Prototype Performance **140**

8.1 Scanning – The NAS EP Benchmark 140

8.2 Sorting 142

8.3 Sparse Matrix Benchmarks 144

8.3.1 The NAS CG Benchmark. 144

8.3.2 The SMOOTH Benchmark 147

8.4 Dense Matrix Benchmark 148

8.5 Conclusions 149

9 Conclusions **150**

9.1 Algorithm Design and Analysis 150

9.2 TPIE: A Framework-Oriented I/O System 151

9.3 Experimental Analysis of I/O-Efficient Algorithms 151

9.4 Extending Our Work in I/O-Efficient Computation 152

9.5 Final Thoughts 153

Bibliography **154**

Part I

Fundamentals

Chapter 1

Introduction

The subject of this thesis is I/O-efficient computation. Thus, it is only natural that we should begin with some definitions that make it clear exactly what we mean by I/O and I/O-efficient computation. I/O, which stands for Input/Output, is the process of moving data between the main memory of a computer and external memory devices, such as a magnetic disks. Some researchers also use the term to describe communication with other computers through a network, but that definition will not be used herein.

I/O-efficient computation is computation designed to make the most effective use of I/O resources. This is extremely important in large scale applications, because I/O devices are orders of magnitude slower than processors and main memories, and thus I/O often represents a very significant bottleneck in overall program efficiency.

I/O-efficiency is accomplished in two ways: first, by minimizing the amount of data that is moved back and forth across the I/O interface; and second, by moving that data as quickly as possible, with minimal disruption of other resources in the machine, such as the CPU.

1.1 Hierarchical Memory

Hierarchical memory models generalize the Random Access Machine (RAM)[1] model of computation by assigning different costs to accessing different data items depending on their location. In a hierarchical memory system, the time required to access a datum depends on its location within the memory system. Typically, the larger the address, the longer it takes to access data from that address. Hierarchical memory systems are an integral part of virtually all digital computer systems; they always have been, and barring a radical change in our understanding of the laws of physics, they always will be. In 1946, at the dawn of the era of electronic digital computation, Burks, Goldstine and von Neuman wrote,

> Ideally, one would desire an indefinitely large memory capacity such that any particular ... word would be immediately available.... [however] We are ... forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity that the preceding, but which is less quickly accessible. [BGvN89]

Almost 50 years later, today's workstations and high-performance personal computer systems employ memory hierarchies that begin with tens to hundreds of registers in a physical register file, continue through one or more levels of cache memory to tens or hundreds of megabytes of DRAM main memory and finally move on to several gigabytes of disk space.

Despite the fact that hierarchical memory systems were postulated very early and are ubiquitous today, the majority of programmers still do their work under the assumption that they have a RAM in which all data is accessible with equal cost. There are good reasons for this. RAMs are easy to program, and are the model assumed by most programming languages.

[1]Readers unfamiliar with the RAM model of computation are referred a standard algorithms text, such as [CLR90] or [AHU74].

Additionally, computer architects, compiler designers, and operating system designers have all contributed mechanisms that provide the illusion of a RAM to application programmers. At the architectural level, cache management policies implemented in hardware heuristicly determine which data is in the cache at a given time. Compilers solve the register allocation problem to map data to architectural registers which are in turn mapped to physical registers by a hardware register renaming scheme. Operating systems implement paged virtual memory, in which pages of data that are considered unlikely to be needed in the near future are moved to disks to make room for pages that are needed immediately (demand paging) or thought likely to be needed soon (prefetching).

A natural question to ask given the prejudice towards simulating the RAM model at a level beyond the reach of the application programmer is whether or not doing so is appropriate for all applications. The answer is that it is not. There are a number of important classes of programs whose performance suffer if the actual behavior of the memory hierarchy is not taken into account. Poor performance occurs because these applications do not exhibit the locality of reference that RAM simulations require. One example is database systems, which must have the ability to implement their own prefetching and page replacement algorithms instead of relying on the general purpose algorithms provided by most operating systems. Another example is scientific computation, where an unfortuitous combination of array sizes and cache sizes can result in severe performance problems. Scientific computation is discussed in Chapter 5. Additional examples are combinatorial problems on large data sets, which are discussed in Chapter 4.

Rather than attempt the Herculean task of efficiently implementing all possible classes of applications at all levels of the memory hierarchy, our work focuses on a moderately sized set of problems and a specific level of the hierarchy. In particular, we are interested in I/O-efficient combinatorial and scientific computations. These classes of problems were chosen because they occur widely in large-scale applications programs that deal with data sets are too large to fit in main memory.

We have chosen I/O-efficient computation as the subject of our research for three reasons. First and foremost, there are a large number of applications that depend of I/O-efficient computation. These applications include large scale sorting; geometric computing; geographic information systems; relational and object-oriented databases, statistics; graph theoretic computation; and scientific computation. Second, we believe that despite the trend towards larger and larger main memories, secondary storage devices, whether they be magnetic or optical, will continue to play an important role in computer systems. This will happen not only because certain applications, such as multimedia, demand so much storage space, but also because secondary storage will continue to be more cost effective than solid state technologies. Finally, I/O systems have unique features that make their efficient use particularly challenging. Foremost among these is the use of block transfer to amortize seek latency over large amounts of data. Block transfer forces us to devise algorithms that exhibit a high degree of spatial locality.

1.2 A Brief History of I/O-efficient Computation

The earliest use of what could reasonably be called I/O-efficient algorithms actually predates what most of us think of as computer systems. I/O-efficient algorithms were applied to the sorting and tabulation of punched paper cards in the late 19th century. The main concern in the design of sorters and tabulators was reducing the amount of physical movement of cards through various bins and tabulators. A fairly detailed account of the history of their development, including numerous references to early descriptions of the various electro-mechanical devices and algorithmic techniques used can be found in [Knu73].

Paper cards remained in common use for at least eighty more years, well into the age of electronic computer systems. By the end of World War II, however, it was becoming apparent that magnetic tape devices were both more efficient and more reliable than punched cards for many tasks. Additionally, unlike the early mechanical sorting and tabulating devices, electronic computer systems in the late 1940s and early 1950s had relatively high-speed internal memories. The combination of random access main memory and tape drives gave us systems not entirely unlike those available today. The main differences are that the older systems were physically much larger, much slower, and had significantly smaller storage capacities. One of the most widely studied operations on tapes in the 1950s and 60s was merging, which, at the time was often called collating. Merging is the process of taking several sorted runs and combining them into a single large sorted run. This is accomplished by interleaving the input runs in an appropriate manner. Merging is still an important technique today, and is discussed in a number of contexts in this thesis.

The next important innovations were the development of magnetic drums and, more importantly, magnetic disks. These developments occurred in the 1950s. Unlike tapes, which had to be read and written sequentially, drums and disks have moving heads which can quickly move from track to track over the surface of rotating magnetic media. Thus, the cost of accessing data stored near the end of the device directly after accessing data stored near the beginning of the device is much lower than it would be if the data were stored on a tape.

Although drums have disappeared, magnetic disks are still widely used today. They are, in fact, the device of choice in the vast majority of external memory systems. In order to increase the bandwidth of disk systems, some systems use many disks in parallel. These systems are called Redundant Arrays of Inexpensive Disks (RAID) [Gib92, PGK88, PCGK89], and are becoming increasingly common.

1.3 Related Work

Prior to the work described in this thesis, a substantial amount of research had been done to address problems similar to those we consider. This work can be divided into two camps. On the one hand, theoretical computer scientists have developed models of I/O and algorithms that run on those models. On the other hand, computer systems researchers have designed and implemented a variety of high performance parallel file systems designed to support high-bandwidth, low-latency I/O. In this section, we will briefly review significant work that has been done in both areas in order to establish the context for our work.

1.3.1 Algorithmics

The fundamental difference between the study of I/O-efficient algorithms and the analysis of algorithms in general is that in the latter we are generally concerned with the amount of computation required to solve a given problem, whereas in the former we are primarily concerned with minimizing the costs associated with moving data from one place to another within a computer. In particular, the greatest concern is for the costs associated with moving data from one type of memory to another. As discussed in Section 1.2, a variety of different memory technologies have been used over the years, and algorithms have been developed that are particularly suited to each of their characteristics.

The earliest algorithms specifically designed for disks were probably sorting algorithms based on the merge sorting techniques used for data on tapes. This work was done in the early 1960s, though there is very little formal literature describing it. Knuth [Knu73] mentions an unpublished paper by George Hubbard that was presented in 1963 which examined merges specifically designed for disks.

The earliest widely known work specifically concerned with disks was that of Floyd [Flo72], which considered the problem of permuting data on disks. This work was the predecessor of numerous more recent papers that have discussed various aspects of permutation on a variety of I/O-models.

Knuth, in his classic work on sorting and searching [Knu73], devoted a significant amount of space to discussion of external sorting algorithms, primarily for tapes, but also for disks.

The I/O-complexity of sorting, fast Fourier transform (FFT), matrix multiplication, and related problems were considered by Aggarwal and Vitter [AV88], who developed a model of sequential I/O on which much of the work in this thesis is based. Vitter and Shriver generalized this model to include multiple independent disks operating in parallel. [VS94a] Both of these models are discussed in detail in Chapter 2.

Analysis of the complexity of computation in more general memory hierarchies has led to the development of a whole series of models, both sequential [ACS87, AACS89, ACF90, ACFS94, VS94b], and parallel [AP94b, ACF93, NV92, NV93c, VS94b].

This work contributed a large number of theoretical results to the field of I/O-efficient computation, but largely ignored issues related to the implementation of algorithms of real systems. These issues have also been studied, although in many cases by different researchers with different motivations and models of computation, as discussed in the next section.

1.3.2 I/O-Efficient Computation Systems

There are two challenges in the design of software systems to support the implementation of I/O-efficient code. First, we must provide programmers with semantics appropriate to the task. This means that programmers should be able both to write functionally correct code and to be able to judge, at least at a high level, how the program will perform based on how it is written. Second, we must provide support, in the form of compilers and/or run-time libraries, that allows the semantic constructs provided to programmers to run efficiently.

In many cases, programmers think more in terms of what has to be computed than where data is physically located. In fact, in programming one typically assumes unit cost for accessing any data item. This notion is encouraged by the structure of many popular programming languages, such as C [KR78], C++[Str86, ES90], FORTRAN [X3J78, ANS90, Ell90], and Pascal [JW75, Wir71].

Unfortunately, for programs to be I/O-efficient, data placement and movement is of great importance, and must be handled either by the programmer or the programming environment. We have identified three classes of systems which solve this problem (or leave it to be solved by the programmer) in different ways. These three classes are array-oriented systems, access-oriented systems, and framework-oriented systems. The distinctions between these classes are almost entirely based on the view of I/O presented to programmers that use them. In the end, solutions to a common problem might result in very similar sets of I/O requests being issued by each of the different types of systems, but the manner in which the programmer wrote the code that led to those requests would typically vary.

Array-Oriented Systems

An array-oriented system is one in which data stored in external memory is accessed primarily through the specification of element-wise access to arrays of data items of a specific type. Array-oriented systems can be very effective for scientific computations. Such computations typically involve regular strides through arrays of data that are written either as iterative loops or explicitly data-parallel operations.

Array oriented systems are by far the most common of the three types. The main reason they are widely used is that there has been extensive demand from the supercomputer community for high-performance parallel I/O in applications such as computational fluid dynamics, molecular dynamics, and weapon system design and simulation. In these applications, array based languages such as FORTRAN predominate, so it is not surprising that array-based I/O systems have been proposed to support them.

A classic example of an array-oriented system is PASSION [CBH+94], which consists of a run-time system and a FORTRAN compiler. PASSION's run-time system is largely devoted to supporting access to regular sections of external memory arrays, such as contiguous sub-arrays and regularly strided subsets of array elements. This makes array-oriented systems particularly well-suited to scientific applications with regular, but non-unit strided, data access patterns.

ViC* [CC94] is a similar system, but it is based on C*, a data parallel variant of the C programming language [KR78]. ViC* generalizes C*'s data parallel shapes[2] to external memory. Programmers using ViC* write essentially the same C* code they might otherwise write, except that they mark a certain subset of shapes with the keyword `outofcore` to indicate that they reside in external memory. A specially constructed compiler translates references to these shapes into a combination of internal computation and calls to library functions that perform the I/O required to access the shape.

Panda [SW94] is another array-oriented system, but it also supports some semantics of access-oriented systems, as discussed below.

The weakness of array-oriented systems is that they are ill-suited to irregular or combinatorial computations. Their weakness in this area is not surprising, since scientific and combinatorial algorithms, whether in the I/O-efficient domain or in the RAM model, demand very different semantics from the systems on which they are implemented. One would not expect most programmers to choose FORTRAN as the language for solving geometric problems; nor should one expect to see them choose an array-oriented system to solve those problems in an I/O-efficient manner.

[2]For readers not familiar with C*, it is sufficient to simply think of shapes as arrays of arbitrary dimension that can have their elements operated on in a data-parallel manner.

Access-Oriented Systems

An access-oriented system is one which provides language constructs and/or library functions that explicitly move data between main memory and external memory. In some cases this data may be strongly typed, as in array-oriented systems, but in general this need not be the case. Elements of access-orientation must clearly be present at lower levels of array-oriented systems (typically in run-time libraries) in order to move data to and from external memory; the distinction between the array-oriented systems and access-oriented systems lies in the degree to which explicit I/O functionality is exposed to the programmer.

One of the simplest and most familiar access-oriented systems is the UNIX file system. In UNIX, one performs I/O by making a system call specifying a buffer of data and its length. Data is then transferred between the buffer in main memory and a disk or other external device.

In general, programmers writing I/O-efficient applications can benefit from higher-level I/O semantics than are provided by a simple typeless access-oriented approach such as the UNIX I/O interface. As will become apparent in Chapters 4 and 5, there are certain classes of I/O operations that are prevalent in I/O-efficient algorithms. For example, one often wishes to load the entire contents of main memory with data from a disk, or to read or write a sequence of data across several disks. A system that provides this level of functionality frees programmers from having to provide it for themselves. An example of a system of this type is the Whiptail file system [SW95, SWC+95].

The Panda [SW94] system resembles an array-oriented system, since external memory data is stored in arrays, but it is actually more of an access-oriented system, since I/O is performed as a result of specific invocations of I/O routines by the programmer, rather than implicitly by I/O code inserted by a compiler.

The MPI-IO initiative [CFH+95], which seeks to integrate I/O into the MPI parallel programming environment [GLS94] is another example of an access-oriented system.

Appropriately designed access-oriented systems can support the efficient implementation of many I/O-efficient combinatorial algorithms, eliminating one of the weaknesses of array-oriented systems. They do not, however, solve all the problems associated with the implementation of such algorithms. In particular, it is still up to the programmer to manage main-memory resources and explicitly perform all I/O.

Framework-Oriented Systems

The final type of system is a framework-oriented system. Framework-oriented systems can be seen as an extension of access-oriented systems which allow the programmer to specify computation that is to take place as an integral part of a particular data access.

The driving notion behind a framework-oriented system is that there are a relatively small number of structural paradigms that are used over and over again in different I/O-efficient algorithms. By a structural paradigm we mean a general method of structuring I/O. For example, many I/O-efficient algorithms read streams of data from external memory and distribute the input data items into a number of output streams. Within this framework, however, different algorithms may use radically different decision-making processes to determine which specific data items go into which output streams. Similarly, the number of output streams may vary significantly depending on the hardware resources available to the running program. Nevertheless, the distribution paradigm is common to all of the algorithms and systems they run on.

Rather than requiring programmers to reconstruct the distribution paradigm for each algorithm, a framework-oriented system provides the paradigm at a high level and leaves programmers with only the task of filling in the functional details. An additional benefit of the framework-oriented approach is that the use of proven paradigms, rather than ad-hoc approaches to solving problems, is encouraged.

TPIE, which is discussed at length in Chapters 6–8 of this thesis, is the first known example of a framework-oriented system.

1.4 Contributions of this Thesis

The goal of this thesis is to demonstrate that a wide variety of scientific and combinatorial problems are amenable to I/O-efficient solutions. This is done through contributions to both the theory and practice of I/O-efficient computation. In particular, this thesis

1. Devises new I/O-efficient algorithms for a variety of combinatorial and scientific problems, and analyzes and improves a number of existing algorithms;

and

2. Presents the design, implementation, and application of TPIE, a Transparent Parallel I/O Environment.

On the algorithmics front, the contributions of this thesis include the development of a number of new algorithmic techniques and the detailed analysis and comparison of several existing algorithms. The algorithmic techniques we developed include

- Distribution sweeping (Section 4.1.1), which produces I/O-efficient algorithms for many problems that are solved by plane sweeping in main memory.
- Batch Filtering (Section 4.1.2), which allows us to answer many geometric queries simultaneously in an I/O-efficient manner.
- An I/O-efficient three-dimensional convex hull algorithm (Section 4.1.3).
- A PRAM simulation technique which is optimal for geometrically decreasing computations (Section 4.2.1).
- New analysis of sorting algorithms (Section 3.2) and dense matrix multiplication (Section 5.2).
- New algorithms for both sparse and dense matrix multiplication (Sections 5.1 and 5.2).

On the implementation side, our contribution is the development of the concept of a framework-oriented system and the design and implementation of the first prototype of such a system. Specifically, our work includes

- The categorization of existing and potential new I/O-efficient computation systems (Section 1.3.2).
- The design and development of TPIE (Chapter 6).
- The implementation of a number of algorithms and benchmark applications in TPIE.
- The analysis of TPIE performance.

We begin our discussion in Chapter 2 by presenting a series of formal mathematical models of I/O-efficient computation. In Chapter 3, we compare striped and independent disk algorithms for sorting and evaluate the conditions under which one or the other are superior. In Chapter 4, we develop a number of new techniques for the I/O-efficient solution of combinatorial problems such as geometric and graph-theoretic problems. These techniques include distribution sweeping (Section 4.1.1), batch filtering (Section 4.1.2), and I/O-efficient simulation of PRAM algorithms (Section 4.2.1). In Chapter 5, we consider I/O-efficient algorithms for scientific computations. In Section 5.1 we develop an I/O-efficient algorithm for multiplying sparse matrices by vectors. In Section 5.2, we compare two algorithms for dense matrix multiplication. We demonstrate that although they are both optimal to within a constant factor, one has a significantly smaller constant factor on its leading term than the other.

Turning from algorithms to implementation, we introduce TPIE in Chapter 6. A number of algorithms have been implemented using TPIE. Some of these are discussed in detail in Chapter 7. We have gathered performance results on a number of benchmarks implemented using TPIE. These are discussed in Chapter 8. In Chapter 9, we conclude by reassessing our contributions to the theory and practice of I/O-efficient computing and discuss research directions we feel are worth pursuing in the future.

Chapter 2

Models and Mathematical Preliminaries

This chapter introduces the models of computation that will be considered in this thesis. We begin by considering the performance of disks relative to that of solid-state random-access memory. We then examine two models of computation designed to take the behavior of disks into account. The first model is the I/O model of computation, as formalized in [AV88]. The second model is the parallel disk model of Vitter and Shriver [VS94a].

After introducing these models, we discuss and analyze some important algorithmic building blocks designed to enable the development of I/O-efficient algorithms. Algorithms based on these techniques are discussed in Chapters 3, 4, and 5.

2.1 Modeling the Behavior of Disks

Compared to random access main memory, disks are extremely slow devices. At the time of this writing, random access dynamic RAM is is widely available with 60ns access time. On the other hand, disks have access times in the 9ms range—approximately 15,000 times slower. We expect that this gap will not only persist, but widen in coming years. If we compare disks to the caches that feed the execution cores of modern microprocessors, the gap is even wider. Cycle times of 5ns are not uncommon today, and 2-3ns cycle time machines are on the near horizon. Superscalar processors [Joh91] push the gap between disk and processor performance even further, by executing multiple instructions in a single clock cycle.

In recent years, CPU throughput has been increasing at an annual rate of 40–60%, whereas disk access times have only been improving at an annual rate of 7–10% [RW94]. What this tells us is that for applications that require access to very large data sets, there is likely to be a very significant bottleneck in the I/O interface between main memory and the disk. If this bottleneck is not adequately dealt with, then the efforts and expense devoted to CPU and RAM development will be of no use to these applications. The CPU and RAM will simply not be able to be kept supplied with data to work on.

Although disks have high access times, they have reasonable peak bandwidth for sequential accesses. For example, the Barracuda family of disks from Seagate, which includes drives with capacities of 4 and 9.1 gigabytes, have average seek times of 8 to 9.5ms, and internal transfer rates of 47.5 to 120 Mbits per second (5.1 to 15 Mbytes/sec.). Hewlett Packard's SureStore Hard Drive 2000LP offers similar performance, with a transfer rate of 45.7–64 Mbits/sec., and a 9.5ms average seek time. Other manufacturers offer drives with similar performance.[1] By using several disk drives in parallel, we can increase the overall disk bandwidth linearly, without access time penalty. For example, eight of the Hewlett Packard disks in parallel wold provide aggregate bandwidth of 366–512 Mbits/sec.

Data on disks is almost always organized into blocks, which are sequential regions of the disk that are always read or written as entire units. By dealing with large blocks, the latency associated with a single disk access can be amortized over all the data read

[1]This is by no means an exhaustive list of models and manufacturers. The performance figures quoted here come from the world wide web pages of the manufacturers mentioned.

or written in the disk access. Of course, an overall computation will only be efficient if most or all of the data in the block is of use in solving the problem at hand. This locality requirement is probably the single largest motivating factor in the design of I/O-efficient algorithms.

In the models of computation that follow, we will consider the cost of accessing any single block on a disk to be constant. It is possible to construct more detailed models of disk drive behavior, as done in [RW94], but such models are necessarily tied to specific makes and models of disk drives, and do not allow us to reason about the I/O complexity of algorithms on a more general level. Our philosophy in modeling and designing I/O-efficient computation is that by treating each I/O operation as if it had unit cost, we will construct algorithms that will minimize the number of I/O operations, and thus operate efficiently on a variety of different types of disks. This is essentially analogous to assuming that all operations on a random access machine are of unit cost despite the fact that they certainly aren't on real machines.

2.2 The I/O Model of Computation

The I/O model of computation was introduced by Aggarwal and Vitter [AV88]. It formalized and generalized the models of I/O used by a number of earlier researchers [Knu73].

In the I/O model, we have a single CPU and uniform cost random access memory, just as in the RAM model, but we also have a single disk. The CPU cannot perform computation directly on data residing on the disk, but instead must move it into main memory first. Because the main memory has a limited capacity, it is also often necessary to move intermediate results of a computation from the main memory back out to the disk in order to free up space for more data to be read in from the disk. The I/O model is illustrated in Figure 2.1.

In the I/O model, the capacity of the main memory is M, which is assumed to be in

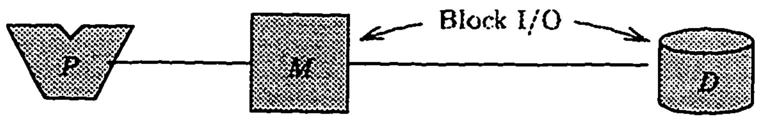

Figure 2.1: The I/O model of computation. In this model, there is a single processor P connected directly to a random-access main memory of fixed capacity M. Beyond this main memory is a disk, whose capacity is unbounded. Data on the disk cannot be directly accessed by the processor. Instead, it must first be transferred into the main memory. Data that that been processed or generated into main memory by the processor must be transferred to the disk to make room for new data.
All data transfer between the main memory and the disk is done in large contiguous blocks. Each block consists of B data items. The I/O complexity of a task in this model is measured in terms of the number of block transfer operations required to complete the task.

units of some basic object over which computation is performed. Such objects consist of a constant number of machine words. For example, if we are dealing with a graph represented as a set of edges, then M would be the number of edges that can fit in main memory. If each edge took 32 bytes to represent, and a total of 16 megabytes of memory was available, then M would be 2^{19}.

The capacity of the single disk attached to the system is assumed to be unbounded, so that all intermediate and final results that it might need to store can be handled. The surface of the disk is divided into a series of blocks, each of which has a capacity B. The blocks are numbered sequentially, starting with 0. Any input to a particular algorithm is assumed to be stored sequentially at the beginning of the disk. In other words, a problem consisting of N input objects will be stored on the first $\lceil N/B \rceil$ blocks of the disk.

Data is transferred to and from the disk one block at a time. Each block transfer is counted as a unit of I/O complexity. In between I/O operations (I/Os for short) the CPU is permitted to perform an arbitrary amount of computation.

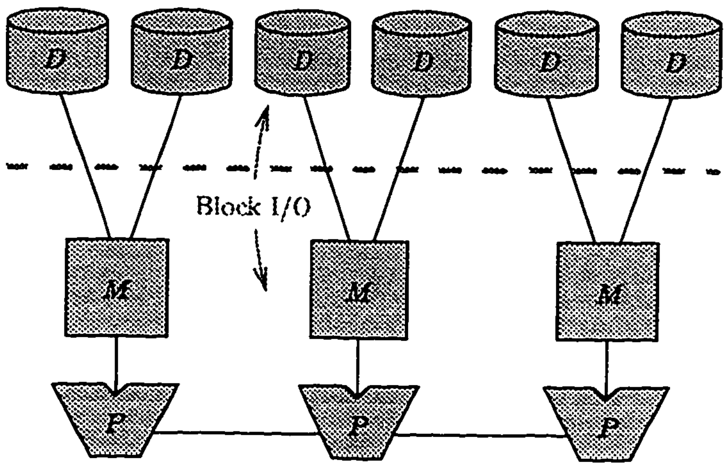

Figure 2.2: The parallel disk model of computation. This model generalizes the I/O model shown in Figure 2.1 by replacing the single disk with several disks in parallel. There may also be several processors in parallel, as shown here, although this is not strictly required by the model. If there are multiple processors, then the main memory may be distributed among them, as shown here, or shared. The reason the details of the processor and memory architecture are not fully specified in the model is that its goal is primarily to model the amount of I/O required in solving a problem (that is the number of I/O operations that bring data across the heavy dotted line in the illustration), rather than the amount or structure of the internal computation done in conjunction with the I/O.
In a single I/O operation, the parallel disk model can transfer a block of data to or from each and every disk in parallel. Thus, much greater aggregate disk bandwidth can be achieved in this model than in the single-disk I/O model.

2.3 The Parallel Disk Model

The I/O model of computation can be generalized to include parallel processors and parallel disks. The former essentially turns the processors and main memory into a PRAM [FW78], while the latter introduces a number of interesting twists into the I/O picture. An example of this generalization is shown in Figure 2.2.

In the parallel disk model, we can access one block on each of the D disks in the system in a single I/O operation. We restrict the total main memory capacity M to be strictly greater than DB, in order to ensure that there is adequate main memory

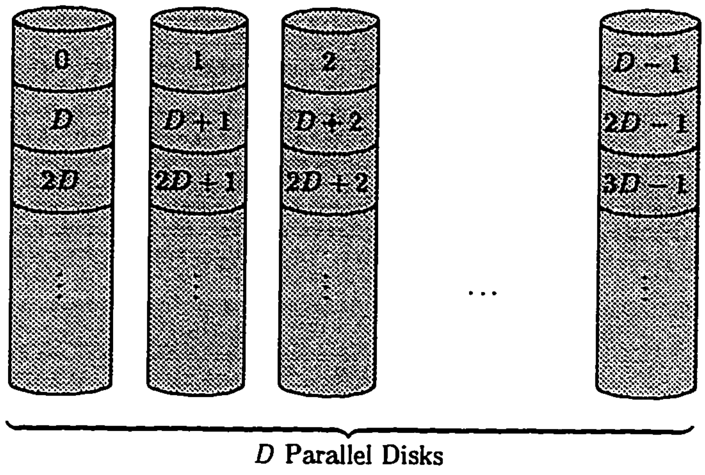

Figure 2.3: Parallel disk block ordering. In this illustration, the blocks on each of the D disks are indicated by slices of the disks. The block numbers are assigned by beginning with the first block of the first disk, in the upper left corner of the illustration. The next block is the first block on the second disk. Numbering continues until the first block on each disk has been assigned a number, at which point we begin with the second block of the first disk and proceed through the second blocks of all the other disks. This process then continues through all logical block positions.

to support such I/O operations.

On a single disk there is an implied logical ordering of blocks based on their physical location on the disk. When several disks are used in parallel, we need a way of generalizing this ordering. The ordering used in the parallel disk model is illustrated in Figure 2.3. We order blocks by their physical position on a single disk, and order blocks at the same physical location on different disks by a set of logical disk numbers assigned to the disks.

There are two common approaches to performing I/O on parallel disk systems. The first method, called disk striping [Kim86, LKB87, SGM86], is illustrated in Figure 2.4. Disk striping reads or writes a block of data in the same logical position on each of the D disks in a single I/O operation. For example, we might read logical block 17 from every disk, or write to logical block 3 of each disk. This technique is called striping

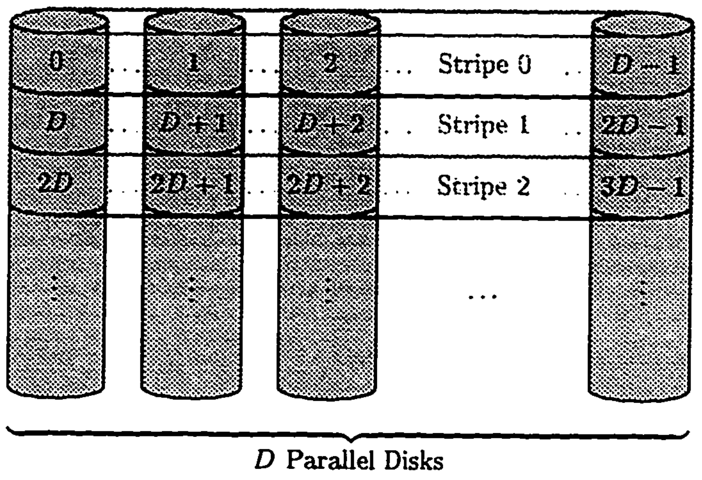

Figure 2.4: Parallel disk striping. In this approach to managing data on parallel disks, an I/O operation must consist of either one read or one write to each of the D disks. Furthermore, the blocks referenced on each of the disks must be in the same logical position on each disk. This is illustrated with the highlighted "stripes" shown above, which we can number logically based on the logical numbers of the blocks that make them up. These stripes look very much like large blocks of size $B' = DB$ data items which must read as a group. The net effect is that the D parallel disks essentially simulate a single larger disk with block size B'.

because it "stripes" data objects across the disks as if they were a single large disk with blocks of size $B' = DB$.

The second method of accessing data on parallel disks is called independent access. As its name implies, independent access allows the machine to read and/or write blocks in different positions on each of the D disks in a single I/O operation. This is illustrated in Figure 2.5.

Clearly the independent disk model is at least as powerful as the striped model, since it can trivially mimic the behavior of a striped disk system. It can be shown theoretically that this model admits algorithms that, in the limit as problem sizes grow extremely large, require less I/O than any that can be implemented on striped disk systems of similar size. However, it is also the case that such algorithms tend to be less

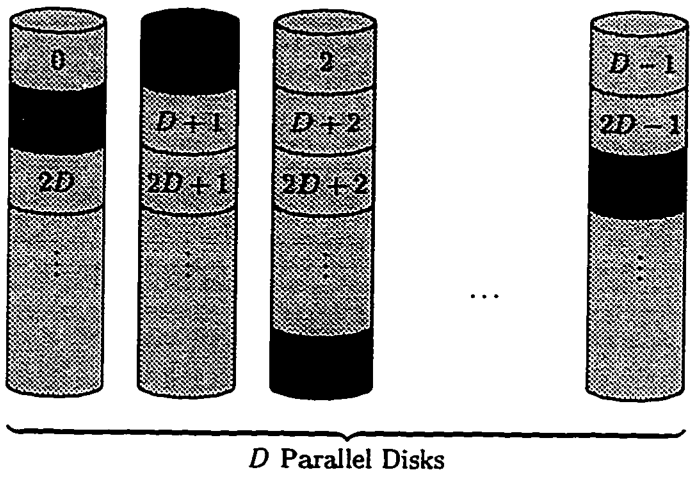

Figure 2.5: Independent access to parallel disks. In a single I/O operation one block can be read from or written to each disk. Unlike striped access, the blocks being accessed on different disks need not come from the same logical location. For example, a single I/O operation might read each of the shaded blocks in this illustration.

efficient than striped disk algorithms on systems of moderate size. The relationship between these two models in this context is discussed at greater length in Section 3.3.

2.4 Paradigms of I/O-Efficient Computation

There are a number of paradigms that are repeatedly in I/O efficient computations. Three of the most important are scanning, distribution, and merging, which are discussed in the following three sections.

2.4.1 Scanning

Scanning is the process of examining each element of a collection of objects stored in external memory, possibly performing some transformation on each, and then writing them back out to external memory.

Scanning is done in an I/O-efficient manner through the use of *buffering*. Buffering is done by allocating a region of main memory, called a buffer, of size B. A block of data from the disk is read into the buffer, and then the contents of the buffer are examined sequentially. Once every item in the buffer has been examined, the next block is read into the buffer and its contents are examined sequentially. When all blocks of data have been examined, the process is complete.

If a scan wishes not only to examine data, but to transform it in some way, the transformed data items are placed one-by-one into an output buffer of size B which resides in main memory. Once this output buffer is full, it is written out to a disk in a single I/O operation, and then the main memory buffer space can be re-used to buffer more output.

An example of a scan is shown in Figure 2.6. Data is read from the disk at the left of the figure one block at a time into the main memory. The input data items x_i are read from the buffer one at a time and the CPU applies some transformation to them, producing output data items y_j. These items are written into an output buffer of size B. When this output buffer becomes full, it is written out to the disk in a single I/O operation.

It was mentioned above that a scan might only read data, and not produce any output. The reverse can also be true. A scan can write data without reading any, if the program that is performing the scan is one that is written so as to generate streams of output data dependent only on its changing internal state.

We can summarize the I/O-complexity of scanning with the following simple lemma, whose proof is apparent from the description of scanning given above.

Lemma 2.1 *A scan operation that reads N_i items and writes N_o items can be performed using*

$$O\left(\frac{N_i + N_o}{B}\right)$$

I/O operations.

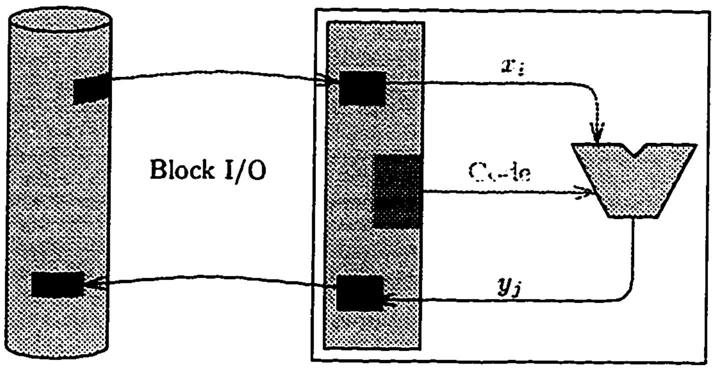

Figure 2.6: Scanning a collection of data items using input and output buffers. At the top of the figure, a block of B input items is read from the disk into main memory. Once this block is in main memory, the individual items x_i that it contains are fed sequentially through some function computed by the CPU. An application of the function may produce an output item y_j. If is does, y_j is placed in an output buffer of size B in main memory.
Once all the items from the input buffer have been exhausted, a new block of input is read from the disk into the main memory buffer in a single I/O operation. If the output buffer holding the y_j gets full, then its contents are written out to a disk in a single I/O operation and it is emptied to allow more output items to be generated.

In the parallel disk model, Lemma 2.1 generalizes as follows:

Lemma 2.2 *In the parallel disk model, a scan operation that reads N_i items and writes N_o items can be performed using*

$$O\left(\frac{N_i + N_o}{DB}\right)$$

I/O operations.

Proof: Instead of using buffers of size B, we use buffers of size DB and read and write stripes of data from or to all the disks in a single I/O operation. □

The low I/O complexity of scanning is by far the most important reason it is widely used. The most important aspect of this complexity is the presence of the B (or DB) in the denominator, indicating that we are making full use of the potential of blocking.

The I/O complexity of scanning is the external memory analog of linear time ($O(N)$) in the RAM model, and we will often refer to it as *a linear amount of I/O* or *linear I/O complexity*. The difference between linear time in a RAM and a linear amount of I/O is that in external memory data must be processed with a significant degree of locality to compensate for the effects of blocking, whereas in a RAM, it can be processed in any order. This leads to an important distinction between linear and non-linear I/O complexity when it comes to algorithms that must perform permutations. This distinction is discussed in Section 3.1.

Scanning is used extensively in the algorithms discussed in Chapters 4 and 5. It is also discussed at length in Chapter 6, where techniques for simplifying its incorporation into I/O-efficient programs are discussed.

2.4.2 Distribution and Merging

Distribution and merging are both generalizations of scanning. Instead of reading or writing individual streams of inputs and outputs, distribution reads a single stream of input and writes objects from that stream into many output streams. Merging is similar, except that it reads many input streams and combines them into a single output stream.

The number of input or output streams merged or distributed is limited by the availability of main memory. Given a main memory of size M, we can buffer M/B inputs or outputs in the single disk model.

Distribution allows us to handle many problems using divide-and-conquer approaches. We distribute the input into subproblems, and then recursively solve those. When the problem instances become smaller than M, we solve them in main memory. Merging, being the opposite of distribution, allows us to construct solutions for larger problems from the solutions to smaller subproblems. Distribution is illustrated in Figure 2.7.

Merging is illustrated in Figure 2.8. It is essentially the same as distribution, except that data moves in the other direction, from many input buffers to a single output buffer.

The I/O complexity of merging M/B input streams, each of length N_{in}, into one output stream of length $N_{in}M/B$ is $O(N_{in}M/B^2)$, since all reads and writes are fully blocked. If we start with inputs of size M, then after one merge we would have intermediate streams of length M^2/B. If we merged M/B of these intermediate streams, we would end up with streams of length M^3/B^2. We could repeat this process over and over again, and given enough original inputs, generate outputs of enormous size.

In many applications, we have N items originally divided into sets of size $O(M)$, which are small enough to be read once, processed entirely in main memory, and then written back out to disk. In order to merge these sets back into one set of size N using the procedure just outlined, the number of I/Os needed would be

$$\frac{N}{B} \log_{M/B} \frac{N}{M/B} = O\left(\frac{N}{B} \frac{\log N}{\log(M/B)}\right).$$

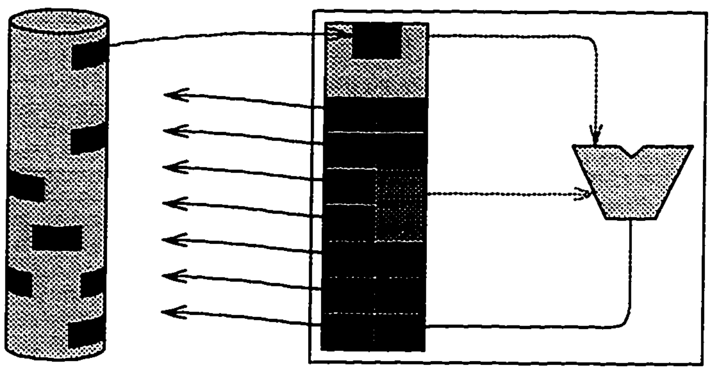

Figure 2.7: I/O-efficient distribution of data. Distribution is very similar to scanning, except that there is a single input buffer and many output buffers. In most cases, the number of output buffers is limited by the amount of main memory available in the machine. The code decides which output buffer each input item should be placed in. As was the case in scanning, the input buffer is refilled when all data has been read from it and the output buffers are written to the disk when they get full. The sequence of blocks written from a particular output buffer is treated as a logical collection of data items that is separate from those data items written from other output buffers, even though they may reside on the same disk. Mechanisms for implementing this logical relationship between blocks are discussed in Section 2.4.3. Once all data has been distributed, each of the output collections of data can be used as input to another distribution or scan.

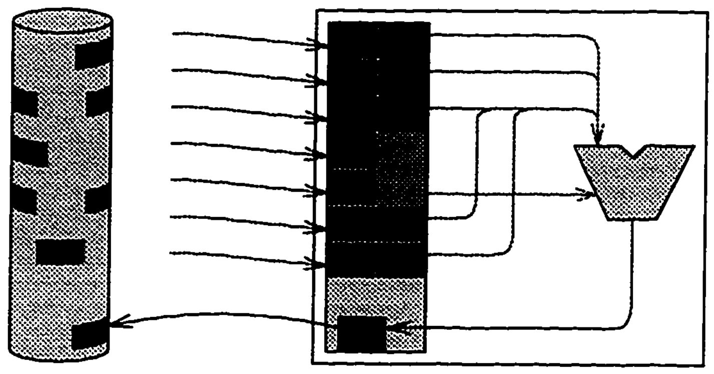

Figure 2.8: I/O-efficient merging of data. Merging is, in a sense, distribution turned around. We have as many input buffers as our main memory capacity allows, and a single output buffer. The code that supervises the merge must decide how to interleave the input items into the output.

This quantity is essentially the analog of $O(N \log N)$ time in the RAM model of computation. We can sort in this bound, using merging, and many other problems that have $O(N \log N)$ time algorithms in the RAM model can also be solved within this I/O bound. Many algorithms that use distribution also operate in this I/O bound.

2.4.3 Block Linking and Meta-data

In our descriptions of scanning, distribution, and merging, we assumed that there was no cost associated with reading the next block from a stream other than the single I/O operation required to read the block itself. If data in blocks are arranged in physical position order on the disk, this assumption is valid. Unfortunately, this arrangement limits us to one logical stream of data per disk drive. In practice this is unreasonable and must be dealt with by providing some mechanism to group blocks from different parts of the disk into ordered logical collections.

The problem of ordering blocks can be handled in one of two ways. The most common way is to rely on the same file system mechanisms that are designed to link blocks into a logical file, namely file system meta-data. Technically, since file system meta-data is resides on disk, its manipulation requires I/O and main memory buffer space in addition to that actually used for data. This overhead, however, is generally quite small. Indeed, since most file meta-data consists of pointers to blocks of data or additional file meta-data, the overall size of meta-data for a large file is on the order of B times smaller than the file itself.

The second way of linking blocks is by reserving a few bytes at the beginning or end of each block for a pointer to the next block in a collection of blocks. This eliminates extra I/O for meta-data, but it also reduces the effective size of each block slightly.

Assuming data items and pointers are of the same size, the tradeoff is that a file of size N with meta-data takes roughly $N/B + N/B^2$ blocks, while one using pointers within blocks takes

$$\begin{aligned}\frac{N}{B-1} &= \frac{N}{B}\left(\frac{1}{1-1/B}\right)\\ &= \frac{N}{B}\left(\sum_{i\geq 0}\frac{1}{B^i}\right)\\ &= \sum_{i>0}\frac{N}{B^i}\end{aligned}$$

blocks. The space overhead is very similar to that of meta-data, since in both cases the leading term of the overhead required on top of the N/B blocks needed just to store data is N/B^2. If data items are larger than pointers and blocks have room for exactly an integral number of data items of the same size, then the analysis is essentially the same. The only difference is that the overhead for meta-data will be a constant factor smaller that that for pointers. In the case where data items are large, but there is extra space at the end of each block because a non-integral number of items fit in each block, then it may be possible to integrate pointers with no overhead whatsoever. In any case, however, the overhead is quite small.

The more important difference between the use of meta-data and pointers is that the pointer method requires that code be written to explicitly manage and follow pointers when locating blocks of data. Care must be taken not to overwrite pointers with data, since they reside in the same blocks.

In most of this thesis, we will not directly concern ourselves with which of the two methods is used. Instead, we will simply refer to collections or streams of data on disk and assume that an appropriate mechanism is being used.

Chapter 3

Permutation and Sorting

Permutation and sorting are among the most widely studied types of I/O-efficient algorithms. Although they are not the primary focus of this thesis, they are important building blocks for many of the algorithms that will be discussed in Part II. Additionally, a thorough understanding of their subtleties is required by anyone wishing to implement I/O-efficient computations in practice.

In this chapter, we will discuss lower bounds on permutation problems and analyze the relationship between two classes of algorithms for parallel disks.

3.1 Lower Bounds: Linear Complexity vs. Permutation Complexity

In order to verify the optimality of various algorithms we might wish to construct for the models outlined in Chapter 2, it is important that we understand the fundamental lower bounds on the amount of I/O required to solve a particular problem. We will initially explore these lower bounds for the single disk I/O model, and then extend them for the parallel disk model.

The first lower bound we will consider is for the fundamental problem of reading in an input of N data objects, so that each data object is, at some point, in main memory. This problems is sometimes referred to as the "touch problem" because it requires that every input object be touched at least once [ACF90, NV90, VS94b]. The trivial lower bound on this problem is given by the following lemma:

Lemma 3.1 *In the I/O model of computation, solving the touch problem requires at least $\lceil N/B \rceil$ I/O operations.*

Proof: Suppose that less than $\lceil N/B \rceil$ blocks are read to solve the touch problem. Then there exists at least one block that is never read, which contains some input objects that are therefore never touched. Thus the touch problem cannot be solved with less than $\lceil N/B \rceil$ I/Os. □

This lower bound can be achieved by scanning, as discussed in Section 2.4.1. At times we will use the shorthand notation *scan*(N) instead of N/B, in expressing the I/O complexity of algorithms or portions thereof that use scanning or variations of it. This expression is commonly referred to as linear time in the I/O model.

In order to derive lower bounds for the amount of I/O required to solve problems less trivial than the touch problem, it is often useful to look at the complexity of the problem in terms of the number of permutations that may have to be performed to solve it. In an ordinary RAM, any known permutation of N items can be produced in $O(N)$ time. In an N processor PRAM, it can be done in constant time. In both cases, the work is $O(N)$, which is no more than it would take us to examine all the input. In external memory with blocking, however, it is not generally possible to perform arbitrary permutations in a linear number $(O(\mathit{scan}(N)))$ of I/Os. In the worst case, $\Theta(\mathit{perm}(N))$ I/Os are required, where

$$\mathit{perm}(N) = \min\left\{\frac{N}{D}, \frac{N}{DB}\frac{\lg N/B}{\lg M/B}\right\}$$

[AV88, VS94a]. When M is extremely small, the $\Theta(N/D)$ term may be smaller, but for values encountered in any real system, the second term is the smaller of the two.

3.2 Sorting

Sorting is one of the most widely studied problems in algorithmics. Literally hundreds of papers have been written on the subject in all manner of computational models. We will discuss it here in order to illustrate the differences between the models of computation we introduced Chapter 2. Our emphasis will be on fully analyzing the algorithms, including the constant factors often hidden by big-Oh notation, taking implementation details into account.

First, let us consider the single disk I/O model of computation. In this model, the sorting algorithm of choice for comparison sorting is merge sort.[1] As mentioned in Chapter 1, merge sort was used on tapes, and later adapted for use on disks [Knu73].

In order to slightly simplify our analysis, we will assume that M is an integral multiple of B and that N is an integral multiple of M. If this is not the case, a small amount of memory ($< B$ items) can be left unused, and/or the problem can be padded with a small number of data items.

Merge sort begins by reading the first M input items into main memory, sorting them internally, and writing them back out. The next M items are then processed in the same way. This continues until all input has been processed and we have N/M sorted subfiles. The total amount of I/O required to produce these sorted subfiles is $2N/B$.

Next, we merge sets of $O(M/B)$ sorted subfiles together into sorted output files. In order to do this, we allocate a main memory input buffer of $O(1)$ blocks for each subfile and an output buffer for the output file. The data items from a particular input file are read sequentially from that file's input buffer, though the order in which they are

[1]Radix sort is often used when the structure of keys permits it, but our concern is with keys that must be compared as indivisible units, and this are not amenable to radix sorting.

read relative to data from other input files is data dependent. When an input buffer is empty, it is refilled by an I/O. An internal memory data structure, such as a heap or other priority queue, is used to manage the merging process, by properly interleaving data from the input files into a sorted output. Assuming N is large, the result of this first set of merges is a set of $O(NB/M^2)$ sorted subfiles of length $O(M^2/B)$.

The results of the first set of merges are then merged together $O(M/B)$ at a time to form larger sorted subfiles of length $O(M^3/B^2)$. This merging process continues until we have one large file of length N, which is fully sorted.

If we take a closer look at how merge sorting is normally implemented, we find that in most cases *double buffering*, which is the use of two blocks of main memory for each input and output buffer, is used. Double buffering allows I/O to be done on one of a buffer's blocks while the CPU is still accessing data in the other, which lowers the overall running time of merge sort by allowing the CPU and I/O system to be active at the same time. With double buffering, there is enough main memory available to merge $M/2B - 2$ inputs into a single output. The additive term -2 is to account for the use of two buffers for the output. Thus, the number of levels of merging required is

$$\left\lceil \log_{\frac{M}{2B}-2} \frac{N}{M} \right\rceil = \left\lceil \frac{\log \frac{N}{M}}{\log\left(\frac{M}{2B} - 2\right)} \right\rceil .$$

The total amount of I/O performed at each level of the merge is $2N/B$, since every data item is read once and written once. Additionally, there is an initial pass through the data to produce the initial sorted runs, which required $2N/B$ I/Os. Thus, the total amount of I/O performed is

$$\frac{2N}{B}\left(1 + \left\lceil \frac{\log \frac{N}{M}}{\log\left(\frac{M}{2B} - 2\right)} \right\rceil\right).$$

If we extend merge sorting to parallel disks using disk striping, then the total number of I/Os performed in a merge sort is

$$\frac{2N}{DB}\left(1+\left\lceil\frac{\log\frac{N}{M}}{\log\left(\frac{M}{2DB}-2\right)}\right\rceil\right). \qquad (3.1)$$

Asymptotically, the I/O bound (3.1) is not optimal for sorting on parallel disks. There are a number of optimal sorting algorithms for parallel disks [AP94a, Arg94, BGV97, NV93a, NV93b, VS94a]. Most of these algorithms use distribution, as discussed in Section 2.4.2, but they take special care to ensure that data is evenly distributed among the independent disks. One algorithm, [BGV97] is based on merging. The leading term of the I/O complexities of all of these algorithms is of the form

$$2k\left(\frac{N}{DB}\right)\left(\frac{\log\frac{N}{M}}{\log\frac{M}{2B}}\right) \qquad (3.2)$$

for some constant $k \geq 1$ that depends on the particular algorithm chosen. The distribution sort algorithm works from the top down by computing medians in the data and then distributing the data into buckets based on the median values. The buckets are then recursively sorted and appended to one another to produce the final output. The value of the constant k depends on the complexity of finding the medians, the quality of the medians as partitioning elements, and the evenness of the distribution of the buckets over the D disks.

Although the the denominator in (3.2) is larger than the denominator in (3.1) by an additive term of $\lg D$, the leading constant factor in (3.2) is larger than that of (3.1) by a multiplicative factor of k. The value of k for currently known distribution algorithms ranges from approximately 3 to 20. The value of k for the merge sort algorithm of [BGV97] is typically in the range of 1 to 2.

If either k or D is too large, merge sorting using striping may be significantly more efficient than independent disk approaches that are asymptotically optimal. We will examine the exact nature of the relationship between these two approaches and the number of disks in the next section.

3.3 Comparing Independent Disk Sorting and Striped Disk Sorting

Before implementing an external sort on parallel disks, it is essential determine which algorithm is most appropriate to the system on which the the code is to run. To compare striping to independent-disk models, we must examine the circumstances under which the I/O complexity (3.2) for using the disks independently is less than the I/O complexity (3.1) with striping. To simplify our analysis, we will compare the leading terms of (3.2) and (3.1), ignoring the ceiling operators. Doing so, we find that the independent disk model requires fewer I/Os whenever

$$k\left(\frac{2N}{DB}\right)\frac{\lg\frac{N}{M}}{\lg\frac{M}{2B}} < \left(\frac{2N}{DB}\right)\left(\frac{\lg\frac{N}{M}}{\lg\frac{M}{2DB}}\right).$$

Eliminating a common factor of $2N/DB$ and dividing through by $\lg\frac{N}{M}$ tells us that this occurs exactly when

$$\frac{k}{\lg\frac{M}{2B}} < \frac{1}{\lg\frac{M}{2DB}}. \tag{3.3}$$

As the number of disks D in our system grows, the right hand side of (3.3) increases, while the left hand side remains the same. Thus, if the inequality holds for some value of D, say D^*, then it holds for all $D > D^*$. For a given algorithm with a fixed constant k, we would like to find out the minimum number of disks D^* for which (3.3) holds. Making an equality of (3.3) and solving for D gives us

$$\begin{aligned}\frac{k}{\lg\frac{M}{2B}} &= \frac{1}{\lg\frac{M}{2DB}}\\ k\lg\frac{M}{2DB} &= \lg\frac{M}{2B}\end{aligned}$$

$$\begin{aligned}
\left(\frac{M}{2DB}\right)^k &= \frac{M}{2B} \\
\left(\frac{1}{D}\right)^k &= \left(\frac{M}{2B}\right)^{1-k} \\
D^k &= \left(\frac{M}{2B}\right)^{k-1} \\
D &= \left(\frac{M}{2B}\right)^{\frac{k-1}{k}}.
\end{aligned}$$

Thus, D must be at least some root of M/B. The critical issue now becomes the value of k. If $k = 1$, i.e., if we do not need extra I/Os to compute $M/2B$ medians that partition the data evenly and if each resulting bucket is divided evenly among the D disks, it is better to use disks independently. However, if $k = 4$, we need $D > (M/2B)^{3/4}$ in order for using disks independently to be worthwhile. This is a rather unreasonable number of disks for any realistic system. For example, suppose we are sorting on a machine with 256 megabytes ($= 2^{28}$ bytes) of RAM and disk blocks of 8 kilobytes ($= 2^{13}$ bytes). On this machine, we would need

$$\left(\frac{M}{2B}\right)^{3/4} = \left(2^{28-13}\right)^{3/4} = 2^{45/4} = 1449$$

disks for the independent disk algorithm to do better than striping. On the same system, an algorithm with $k = 1.5$ could outperform striped merge sort on a much more reasonable 26 disks. It is not clear at this point how close to 1 that k can get. What is clear, however, is that only algorithms for which k is quite small, such as the merge sorting algorithm of [BGV97], can compete with striping on machines of any reasonable size.

Another important aspect of the behavior of I/O-efficient algorithms for sorting concerns the behavior of the logarithmic factor $\lceil \lg(N/M)/\lg(M/2DB - 1)\rceil$ in (3.1). The logarithmic term represents the number of merge passes in the merge sort, which is always integral, thus necessitating the ceiling notation. The ceiling term increases from one integer to the next when N/M is an exact power of $M/(2DB) - 1$. Thus over very wide ranges of values of N, of the form $M^i/(2DB - 1)^i < N/M \leq M^{i+1}/(2DB - 1)^{i+1}$ for some integer $i \geq 1$, the I/O complexity of sorting remains linear in N. Furthermore, the possibility of $i > 3$ requires an extremely large value of N if the system in question has anything but the tiniest of main memories or a huge number of disks. As a result, although the I/O complexity of sorting is not, strictly speaking, linear in N, in practice it often appears to be over very wide ranges of N.

Many of the algorithms discussed in Part II of this thesis exhibit behavior similar to that of sorting on striped or independent disks. For this reason, they will be discussed in the context of single disk systems. The extension of these algorithms to parallel disks by striping will not necessarily yield theoretically optimal algorithms, though it will produce algorithms that are very efficient in practice.

Part II

Algorithms

Chapter 4

Combinatorial Algorithms

Combinatorial algorithms are algorithms for problems on data sets consisting of discrete structures. In this chapter, we concern ourselves with two important classes of combinatorial problems: geometric problems and graph theoretic problems.[1]

4.1 Geometric Algorithms

Geometric problems, that is to say problems whose inputs and/or outputs are geometric objects, have been widely studied, and solutions to many are well known [PS85]. These solutions typically assume a RAM model of computation, although significant work has also been done for parallel machines, primarily using the PRAM model [ACG$^+$88, AL93].

In this section, we will examine techniques for developing I/O-efficient geometric algorithms. These algorithms have important applications in a number of domains. For example, design-rule checking on a very large VLSI design may involve looking for intersections among tens or hundreds of millions of rectangles. Another set of examples come from geographic information systems (GIS), which store tremendous amounts of

[1]The work presented in this chapter originally appeared in two separate papers: [GTVV93], which was joint work with Michael Goodrich, Jyh-Jong Tsay, and Jeffrey Vitter; and [CGG$^+$95], which was joint work with Yi-Jen Chiang, Michael Goodrich, Eddie Grove, Roberto Tamassia, and Jeffrey Vitter.

inherently geometric data. The particular problems we consider include line segment and rectangle intersection, nearest neighbor queries, and convex hulls in two and three dimensions.

We will discuss geometric algorithms in the context of a single disk model of I/O, though the results can be extended to other models as well. In particular, they can be extended to parallel disk systems through the use of striping, as discussed in Section 2.3. For systems of reasonable size, striping will produce algorithms that are more efficient than the more complicated algorithms required to make optimal use of independent disks.[2] The effect is essentially the same as that seen for sorting in Section 3.3.

4.1.1 Distribution Sweeping, a Generalization of Plane Sweeping

The first technique we have developed for I/O-efficient geometric computation is called distribution sweeping. It is a technique for generalizing plane-sweep algorithms, which represent an important class of geometric algorithms for the RAM model [PS85].

A General Problem Framework

Both plane sweeping algorithms and distribution sweeping algorithms are designed to solve problems that consist of a set of updates $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_N\}$ and a set of queries $R = \{\rho_1, \rho_2, \ldots, \rho_K\}$. Each update σ_i is an operation *insert*(y) or *delete*(y), where each such y is taken from a known total order ω. There is an explicit temporal ordering of the queries and updates, which is given by a time-stamp function $t : \Sigma \cup R \to \mathcal{T}$, where $\mathcal{T}$ is a completely ordered set. In order to solve the problem, we must answer each query τ_i as it would be answered by a dynamic data structure to which all updates with time-stamps less than $t(\tau_i)$ had been applied, in order of their time-stamps.

In order to make this class of problems more concrete, let us consider an example.

[2]By reasonable size, we mean a number of disks that we can reasonably assume will be connected in parallel to a single CPU or a small scale symmetric multiprocessor. This might be on the order of tens, but not hundreds.

Suppose we have $N + K$ line segments, N of which are horizontal, and K of which are vertical. We would like to know which pairs of segments intersect. This problem is called orthogonal segment intersection. We will assume, for the same of simplicity, that we are only concerned with intersections between horizontal and vertical segments, and not overlapping pairs of horizontal segments sharing the same y coordinate or overlapping pairs of vertical segments sharing the same x coordinate.

We let time-stamps correspond to x coordinates in the plane, and let ω consist of y coordinates in the plane. This is illustrated in Figure 4.1. We can construct Σ and R as follows: For each horizontal line from (x_1, y) to (x_2, y), where $x_1 < x_2$, we create two updates. The first update is *insert*(y) and has a time-stamp x_1. The second update is *delete*(y) and has a time-stamp x_2. For each vertical line from (x, y_1) to (x, y_2), we construct a range query with time-stamp x. The query is of the form "return all elements between y_1 and y_2."

To see that each intersection results in a response to a query, we note that for any time-stamp x^*, the set of y coordinates that have been inserted but not yet deleted is exactly the set of y coordinates of horizontal line segments that have one endpoint $x_1 < x^*$ and the other endpoint $x_2 > x^*$. If there is a vertical line segment at $x = x^*$, then a range query based on the y coordinates of its endpoints returns exactly that subset of the horizontal segments represented in the tree that intersect the vertical line segment.

Plane Sweeping

Plane sweeping [PS85] solves problems of the class just described in a very direct manner. It begins by sorting $\Sigma \cup R$ by time-stamp. It then examines the sorted result iteratively, from smallest time-stamp to largest. Each time an update is encountered, it is performed on a dynamic data structure maintained by the algorithm. When a query is encountered, it is performed on that same dynamic data structure.

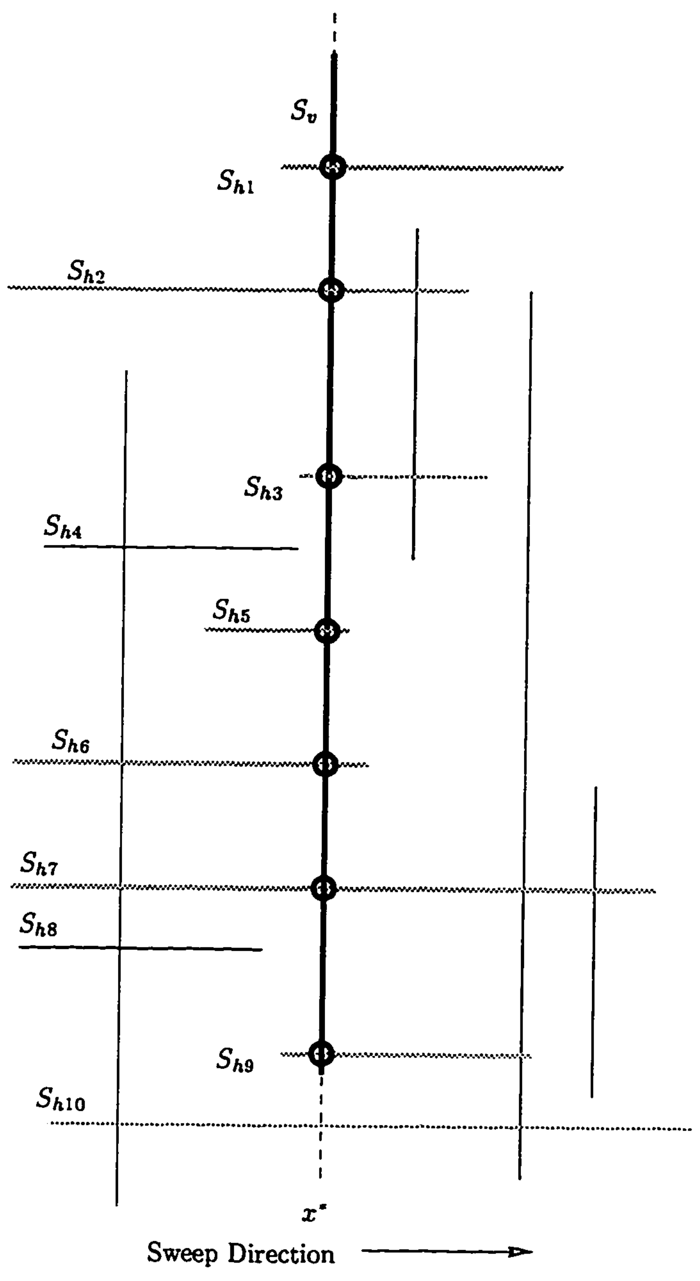

Figure 4.1: Orthogonal segment intersection. The current time-stamp x^* is shown with a dashed line. The vertical segment S_v that has the time-stamp x^* is shown in bold. Horizontal segments that are stored in the dynamic data structure at time x^* are highlighted in grey. Segments S_{h4} and S_{h8} were once in the data structure, but have since been removed. All the grey horizontal segments except S_{h10} are within the y range queried by S_v, and are thus reported to intersect it. Once this reporting is completed, the timestamp will advance, and more queries and updates will be performed.

If time-stamps correspond to x coordinates of geometric objects in the plane, then the order in which the elements of $\Sigma \cup R$ are considered is exactly the order in which the geometric objects from which they obtained their time-stamps would be encountered by a vertical line $x = a$ sweeping across the plane from $x = -\infty$ to $x = +\infty$. This is the origin of the name "plane sweeping."

Simply by selecting an appropriate correspondence between geometric objects in the plane and elements of Σ and R, a wide variety of problems can be solved by plane sweeping. In addition to segment intersection, these include a variety of problems dealing with sets of isothetic rectangles, which are rectangles in the xy plane whose sides are parallel to the x and y axes. The problems we can solve on sets of isothetic rectangles include determining the area and perimeter of their union. Plane sweeping can also be used for the two dimensional maxima problem, and planar point location. Detailed examples of the use of plane sweeping for many of these problems can be found in [PS85].

Analyzing the time complexity of the plane sweeping as applied to orthogonal segment intersection is very similar to analyzing it in general. First of all, we must sort the endpoints of the horizontal segments and the vertical segments by their x coordinates. This can be done in $O(N \lg N)$ time using any of a variety of sorting algorithms, quicksort [Hoa62, Sed75, Sed78], being the most likely candidate. Once the sorting has been completed, we can sweep across the plane by iterating through the sorted list we just generated. Each element of the list generates an update or query to the tree T. If T is a balanced binary tree, such as a red-black tree [Bay72, CLR90, GS78], then each tree update can be done in $O(\lg N)$ time, since the tree never has more than $O(N)$ elements. A range query on a balanced binary tree can be done in $O(\lg N + T_q)$ where T_q is the number of items in the tree that answer the query. Since there are $O(N)$ updates and queries, the total complexity of the plane sweep is $O(N \lg N + T)$, where T is the total number of results returned by all the queries, which is just the total number of pairs of intersecting segments.

A logical question for us to ask is how effective plane sweeping is on large scale geometric problems that do not fit in main memory. We can begin by examining this question in the context of the orthogonal segment intersection example used above to introduce plane sweeping. The initial sorting is not a problem; there are a variety of I/O-efficient sorting algorithms, as discussed in Section 3.2. Scanning the sorted list of objects is also not a problem; we simply read blocks or stripes of data sequentially. The next part of plane sweeping, maintaining a dynamic data structure associated with the current position of the sweep line, is where we can begin to run into trouble. If it were the case that the dynamic data structure always remained small enough to be stored entirely in main memory, then there would be no problem. Experimentation has been done on synthetic sample cases to demonstrate the efficiency of this approach [Chi95]. The same work also included synthetic data sets designed to evaluate the performance of this approach in cases where the dynamic data structure grows larger than main memory. In those cases, performance is much worse, as we would expect. It is not yet known exactly how this technique will work in practice for real problem instances.

If the dynamic data structure grows significantly larger than the capacity of main memory, then we have a very different story. Pointer-based data structures, such as the balanced binary trees typically used in plane sweeping, can be inefficient for data structures that are too large to fit in main memory. The inefficiency arises because following pointers in such a data structure may move from one block of data to another, to another, and so on, without being able to make use of other nodes stored within the same block. The best we can hope for is that the tree remains perfectly balanced and divided into blocks consisting of B nodes arranged as a binary tree of height $O(\lg B)$. In this case, the binary tree is effectively reduced to a form of B-tree [BM72, Com79], which supports updates in $O(\log_B N)$ I/Os.

The idea of using a B-tree in this way was investigated by Chiang [Chi95]. If we use a B^+-tree [Com79], which is a variant of a B-tree that is particularly well suited to answering range queries, then insertions and deletions can be performed in $O(\log_B N)$ I/Os and each query q can be performed in $O(\log_B N + T_q/B)$ I/Os, where T_q is the number of items returned as the answer to the query q. This would give our plane sweep algorithm an overall I/O complexity of

$$O\left(\frac{N}{B}\frac{\lg N/B}{\lg M/B} + N\log_B N + \frac{T}{B}\right)$$

on a single disk system. In general, the $N\log_B N$ term can be expected to dominate the term associated with the sort because it lacks a B in the denominator. The B is missing because an individual I/O-based update or query is made for each and every point the scan line crosses.

Distribution Sweeping

In order to make plane sweep algorithms more I/O-efficient, we have developed a technique called *distribution sweeping*, whose primary feature is that it entirely eliminates the dynamic data structure associated with the sweep line.

We begin, as in plane sweeping, by sorting $\Sigma \cup R$ by x coordinate. We then sweep, but in a different way. Instead of maintaining the dynamic data structure as we sweep, the input events (updates and queries) are divided into buckets $\beta_1, \beta_2, \ldots, \beta_s$ based on their y coordinate as the sweep line moves over them in the x direction. The buckets are delineated by $s-1$ splitters in the y direction, much as if we were doing a distribution sort based on the y coordinate. The splitters are chosen so that there are $\Theta((N+K)/S)$ queries and updates in each bucket. The events distributed to any given bucket are correctly ordered by x coordinate, since all events were sorted by x before the sweeping began. As the events are distributed, portions of the final solution to the problem that involve interactions between objects that are distributed to different buckets are reported. This concept is necessarily vague, since the details of how objects from different buckets interact is highly dependent on the exact problem begin considered. A concrete example is given in Section 4.1.1. Whatever data structures are required to find these portions of the final solution must either fit entirely within main memory or be structured so that the I/O complexity of maintaining them can be amortized over the cost of reporting the overall solution to the initial input problem.

Once the input events have been divided into buckets, the problem is solved recursively on the contents of each bucket in order to report the remainder of the solution. Finally, the solutions to the recursive problems on the buckets are merged into a solution for the overall problem.

The initial sweep is called the *pre-sweep* because it is performed before the subproblems are recursively solved. The process of merging the solutions after solving the recursive subproblems is called the *post-sweep* because it is performed after the recursive subproblems are solved.

In order to divide the problem into recursive subproblems, we decompose each query ρ_i into a set of sub-queries $\rho_{i,1}, \rho_{i,2}, \ldots, \rho_{i,s}$ over the ranges of y that correspond to each of the buckets. In most cases a significant fraction of the sub-queries are known *a priori* to be empty. These empty queries need not be explicitly passed to the recursive subproblems. Not passing them on is critical in avoiding an exponential increase in the number of subproblems created at each subsequent level of recursion. In fact, a key part of the analysis of many distribution sweeping algorithms is proving that at most $O(1)$ queries resulting from the decomposition of any one original query ρ_i exist at any level of recursion.

If the number of buckets into which updates and queries are partitioned is $s = M/B - 1$, then the process of distributing the updates and queries to subproblems can be handled as we scan through them in order to find solutions involving iterations of events in different buckets. We can do this because we have enough main memory to both buffer the input queries and updates and the s output streams of sub-queries and updates. If the total number of sub-queries in all buckets at each level of recursion remains $O(K)$, and the total number of updates in all buckets at each level of recursion remains $O(N)$, then the process of sweeping and distributing the queries and updates takes a total of

$$O\left(\frac{N+K}{B}\frac{\log(N+K)}{\log(M/B)}\right)$$

I/Os.

Similarly, if the size of the answer to each query is $O(1)$, then post-sweeping also takes a total of

$$O\left(\frac{N+K}{B}\frac{\log(N+K)}{\log(M/B)}\right)$$

I/Os.

The only aspect of distribution sweeping that we have not yet covered is how we determine splitter values that will evenly divide the input events into subproblems. In order to do this, we first sort $\Sigma \cup R$ by y coordinate. The medians for the first level of distribution correspond to the y coordinates in positions $(N+K)/s$, $2(N+K)/s$, $3(N+K)/s$, ..., $(s-1)(N+K)/s$, in this sorted list. The medians used at the next level of recursion correspond to the y coordinates at positions $(N+K)/s^2$, $2(N+K)/s^2$, $3(N+K)/s^2$, ..., $(s-1)(N+K)/s^2$, $(s+1)(N+K)/s^2$, $(s+2)(N+K)/s^2$, ..., $(2s-1)(N+K)/s^2$, $(2s+1)(N+K)/s^2$, ..., $(s^2-1)(N+K)/s^2$. This pattern continues down through the levels of recursion. A complete schedule of medians used by the pre-sweeps can be constructed by permuting the sorted events appropriately. The correct permutation will depend on the order in which we want to examine recursive subproblems, such as breadth-first or depth-first. In any case, these permutations can certainly be routed within the I/O bounds required to sort. These bounds match the amount of I/O already required for the pre- and post-sweeps.

Application to Orthogonal Segment Intersection Reporting

In order to make distribution sweeping more concrete, we will demonstrate how it can be applied to the problem of orthogonal segment intersection. This is illustrated in Figure 4.2.

During the pre-sweep, whenever the left endpoint of a horizontal segment is encountered, the segment is inserted into an *active list* A_i associated with the bucket γ_i in which the segment lies. Later, when the right endpoint is encountered, the segment is deleted from A_i. When we encounter a vertical segment R, we examine all the buckets whose y extents are completely spanned by R. We report that all the horizontal segments in the active lists of those strips intersect R. If we implement the active lists as external memory stacks, such that only the top block of each stack is kept in main memory at any given time, then the amortized I/O complexity of inserting a segment into an active list is $O(1/B)$ since there is only one I/O for every B items inserted. Thus the total cost of inserting the N horizontal segments into active lists is $O(N/B)$.

Deleting elements from the active lists can be more problematic, since the order of the location of the right endpoints of segments need not correspond to the order of their left endpoints, and thus we might have to examine an entire active list to find just one point to delete. This examination could potentially cost us a lot of I/O.

The way to get around this problem is to simply forget about deletion events and integrate the process of deleting segments from the active lists into the process of examining the active lists to report intersections. Whenever we encounter a vertical segment S_v, we scan all the active lists corresponding to buckets whose y extent it contains. Each horizontal segment S_v in such an active list either crosses S_v or has a right endpoint to the left of S_v. In the former case, we report the fact that S_h intersects S_v to a buffered output file of intersections and write S_h to a new active list for the bucket. In the latter case, we do nothing, effectively deleting S_h from the active list of its bucket, since it is not copied to the new active list.

Figure 4.2: (On next page) Orthogonal segment intersection using distribution sweeping. The boundaries between buckets are illustrated by dashed lines. At the point in time illustrated, the sweep has reached the x coordinate of the bold vertical line segment S_v. The horizontal segments that are in the active lists of the buckets are highlighted in grey. This intersections that are reported when the active lists are scanned are highlighted with dark circles. Note that the intersection of S_v and S_{h9} is not reported, because S_v does not fully span the bucket containing S_{h9} and thus that bucket's active list is not examined. Instead, this intersection will be reported by a recursive solution to the problem on that bucket.

Segments S_{h4} and S_{h8} are examples of segments that are removed from the active lists of their buckets when S_v is processed, since they do not intersect S_v and thus cannot intersect any vertical segments further to the right.

At this point, all intersections of horizontal segments and S_v within the top three buckets have been reported because S_v fully spans the buckets. This is not the case in the lowest bucket, however. S_v must be distributed to that bucket so that any intersections between it and horizontal segments in that bucket can be found by recursion on the contents of the buckets. In this example, the intersection of S_v and S_{h9} will be found in this way.

In this illustration, the top endpoint of S_v has the highest y coordinate of any endpoint of any segment. This is the situation we would see if the instance of the problem we were trying to solve was a recursive sub-instance consisting of all segments distributed to a given bucket at the next higher level of recursion. Notice that the bottom portion of S_v that gets distributed to the lowest bucket has this same character.

At a given level of recursion, each horizontal segment is inserted into one active list, and read from that active list once for each vertical segment that intersects it and once when it is deleted. The total amount of I/O needed to do this at a level of recursion is $O((N + T_\ell)/B)$, where T_ℓ is the number of intersections reported at the level of recursion.

We have described how to complete the portion of the pre-sweep that finds intersections; we are now left only with the problem of distributing events to subproblems. Horizontal lines are relatively simple. We distribute them based on their y coordinates. Vertical segments are distributed only to buckets they do not completely span, since intersections that occur in buckets they completely span have already been reported by examining active lists.

The only way a segment can interact with a bucket but not completely span it is

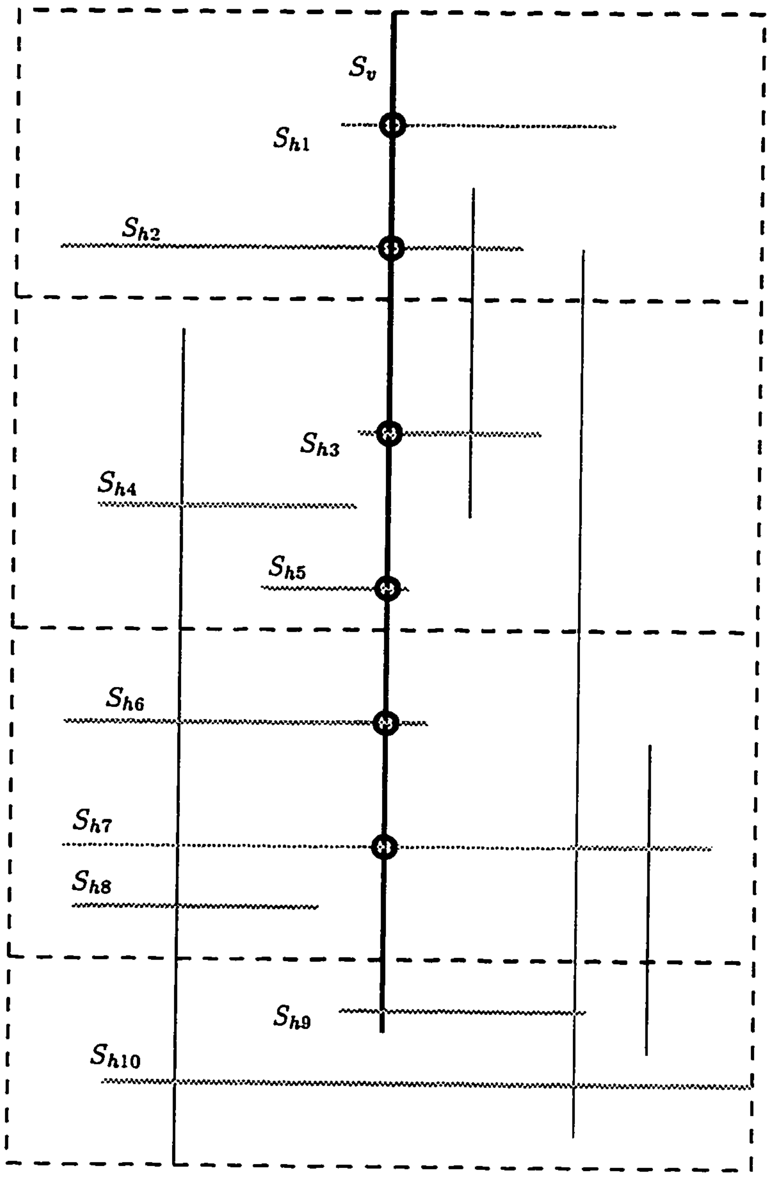

Figure 4.2: Orthogonal segment intersection using distribution sweeping. See caption on facing page.

if it has an endpoint that is in the bucket's interior. If we are dealing with an instance of the problem that was created as a recursive sub-instance of the problem, then an endpoint of a vertical segment can either have been an endpoint of an original input segment or the point at which a segment being distributed to the recursive sub-instance intersected a bucket boundary. The former (as illustrated by the upper endpoint of S_v in Figure 4.2) can be inside a bucket, but that latter (as illustrated by the lower endpoint of S_v in Figure 4.2) cannot. If we consider all sub-instances of the problem at a given level of recursion, only two subsegments of an original input segment can be present, one corresponding to each of the original input segment's endpoints. This property is of critical importance in the analysis of the I/O complexity of this algorithm, since it ensures that the the total amount of I/O required for distribution at a given level of recursion linear in the number of original inputs.

Once distribution has reduced the number of events in a bucket to no more than M, distribution sweeping ends and the intersections occurring within the bucket are reported by reading all of the bucket's events into main memory and solving the problem in main memory, using, for example, a sweep line algorithm.

The behavior of the distribution sweeping algorithm just discussed for the orthogonal segment intersection problem can be summarized by the following theorem:

Theorem 4.1 *Given an input consisting of a total of N horizontal and vertical segments, all T of their pairwise intersections can be reported using*

$$O\left(\frac{N}{B}\frac{\lg N}{\lg M}+\frac{T}{B}\right)$$

I/Os.

Proof: We have shown that at each level of recursion there are at most two distributed segments corresponding to each original input segment. The distribution of these segments is through buffers, so it is fully blocked. Because the endpoints of all segments (both horizontal and vertical) are evenly divided among blocks, the number of levels of recursion is

$$O\left(\frac{\lg N}{\lg M}\right).$$

The total amount of I/O used in distribution is thus

$$O\left(\frac{N}{B}\frac{\lg N}{\lg M}\right).$$

Each of the T segment intersections is reported exactly once. If these intersections are written to disk through a single output buffer than the amount of I/O required to report them is $O(T/B)$. The total I/O complexity of our algorithm is thus

$$O\left(\frac{N}{B}\frac{\lg N}{\lg M}+\frac{T}{B}\right)$$

as claimed. □

4.1.2 Batch Filtering

In this section we demonstrate how, for many query problems in computational geometry, we can represent a data structure of size N in N/B disk blocks in such a way that K constant-sized output queries of the data structure can be answered in $O((N/B + K/B)\log_{M/B}(N/B))$ I/O operations. Because we represent the data structure as a dag (directed acyclic graph) through which the K queries filter down from source to sinks, we call this technique *batch filtering*.

Given a data structure that supports queries, we can often model the processing of a query as the traversal of a decision dag isomorphic to the data structure. We begin at a source node in the dag, and at each node we visit, we make a decision based on the outcome of comparisons between the query value and some number d of values stored at the node. We then make a decision as to which of the node's $O(d)$ children to visit next. This process continues until we reach a sink in the dag, at which point we report the outcome of the query.

The class of dags in which we are particularly interested is planar layered decision dags. In such dags, each node belongs to a particular layer, and the layers are numbered with integers. Every edge in the dag goes from a node at some layer i to a node at level $i+1$. If v_i is a layer i node and v_{i+1} is a layer two node, and there is an edge from v_i to v_{i+1}, then we call v_{i+1} a child of v_i v_i a parent of v_{i+1}. In addition to the layer property, the dags we consider must be planar. Trees are simple examples of a planar layered dags.

An important property of planar layered dags is that batches of queries can be processed through them in an I/O efficient manner. This is accomplished by processing all queries through the first layer of the dag, then processing them all through the next layer, and so on.

Let us consider the details of how this is done. Let $G = (V, E)$ be a planar layered decision dag with a single source such that the maximum out degree of any node is M/B. Assume the source s of G is at layer 0.

We begin processing the queries by placing all K queries into a FIFO queue which we call the layer 0 queue. We then read each input from the queue, using a single block of size B in main memory to buffer the input as we read it. For each query, we make a decision at s as to which child of s the query should directed. We maintain several main memory buffers of size B, one for each child of s. When we have decided that a given query q should be directed to v, a child of s, we place q in the output buffer associated with v. As these output buffers become full, they are written out to disk. If multiple blocks of queries are written to disk for a single child, they are linked together as described in Section 2.4.3. Because the out degree of the root is at most M/B, this process is almost exactly like distribution, as defined in Section 2.4.2.

Once all of the queries have been divided among children of the root, we create a specially structured FIFO queue of all queries. An example of this kind of queue is shown in Figure 4.3. The queue consists of two kinds of entries: queries and pointers to blocked lists of queries in secondary memory. Traversing the children of s from left to right, we check whether the output buffer for each child ever became full. If it did, we append the current contents of the buffer to queries previously written for that child, and place a pointer to those queries in the FIFO queue. If it did not, we insert the data items directly into the queue, which is buffered before being written to disk.

When all children of s have been examined in this way, the queue contains all the queries for each of children, ordered from left to right. Some of these are explicitly in the queue, while others are referred to through pointers in the queue. This is illustrated in Figure 4.3. The queue labeled "Layer i" shows how the queue of queries to four children of s might look. Examining from left to right, it consists of a node with a pointer to queries, two nodes with explicitly represented queries, and another node with a pointer to queries.

To proceed beyond layer 1, we process the nodes of each level i from left to right. All K inputs are represented in the queue for layer i, in order of nodes from left to right, so the queries for each input can be read in turn. Each node's inputs are treated in the same manner that all nodes were treated at the root. The only difference is that the output queues of the nodes at layer i are concatenated into a single input queue for layer $i + 1$, as illustrated in Figure 4.3.

Because the dag is planar, only one child can be shared between two nodes on a given level. The nodes must be adjacent on the layer and the child must be the rightmost child of the first node and the leftmost child of the second. This situation is also illustrated in Figure 4.3, where the first two nodes on layer i share a child on layer $i + 1$. When this occurs, the output buffer for the shared queue remains in main memory after its first parent is processed, and is used for more output from the second parent. Only after the second parent has processed all its queries is the child put into

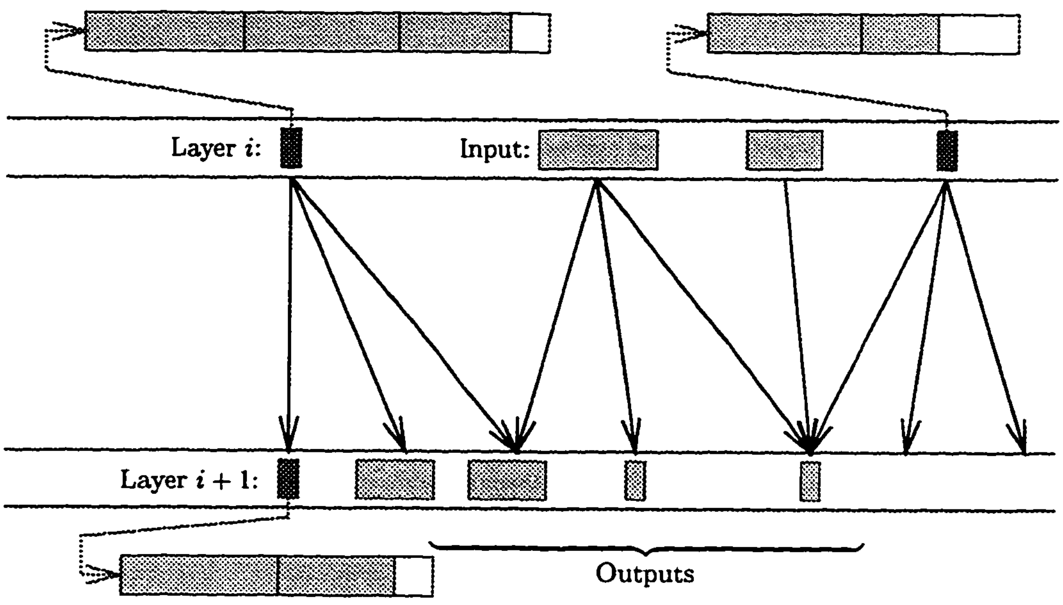

Figure 4.3: Batch filtering queries through a planar layered decision dag. This figure illustrates the process of reading all the queries from the FIFO queue for layer i and distributing them to layer $i+1$. The FIFO queue for layer i is at the top of the figure and the queue for layer $i+1$ is at the bottom. At the moment shown in this snapshot, queries from the layer i node labeled "Input" are being distributed to the layer $i+1$ nodes labeled "Outputs."

Each node in layer i has a representation of the queries that travel through the node. These are shown by rectangles within the layer. There are two types of representations. Nodes having less than B queries are shown with lightly shaded rectangles. These queries are represented explicitly in the queue. Nodes having more than B queries have a pointer in the queue, shown by a thin dark rectangle, which points to a list of blocks containing the actual queries.

In processing layer i, we move from left to right in the queue, processing the queries associated with each node in turn. For each query, we make a decision which edge out of the node to a node in layer $i+1$ the query should follow. We then place it into an output buffer associated with that node. If the output buffer becomes full, we write it to disk and begin to use a pointer representation for the queries at the associated node.

the FIFO queue. Because it is leftmost, it is written first, and this the appropriate ordering of queries for nodes on layer $i+1$ is preserved.

Once all queries reach the bottom of G, we scan the last queue, and report the appropriate outcome for each query based on the node in the queue which was sent to.

The I/O complexity of batch filtering as described above is given by the following lemma:

Lemma 4.2 *Let $G = (V, E)$ be a planar layered decision dag with a single source such that the maximum out degree of any node is M/B. Let $N = |V|$ and let h be the height of G. If G is represented in N/B blocks, with the nodes ordered by layer and the nodes within a layer ordered from left to right, then it is possible to answer K queries on G in $O(N/B + hK/B)$ I/O operations.*

Proof: Each layer of the dag is processed using $O(K/B)$ I/Os to read and distribute queries to the next level. As this is done, the dag itself has to be read so that the nature of the decision to be made at each node can be determined. The amount of data needed to represent each node is linear in its output degree, but because G is planar, Euler's law [Bol79, PS85] tells us that the total output degree of all nodes is $O(N)$. Therefore, since the nodes are appropriately ordered, they can be read as needed by scanning, which takes $O(N/B)$ I/Os. Processing all h levels thus takes

$$O\left(\frac{N}{B} + h\frac{K}{B}\right)$$

I/Os. □

Luckily, the restrictions we have imposed on the type of decision dags we can handle with batch filtering are not too severe. In particular, many computations use decision trees, which clearly constitute a special case of the lemma. Often these trees are binary, but we can divide a binary tree into layers of height $O(\log M/B)$ and then store each node on a layer boundary along with all its descendants in the layer below it as a single node with branching factor M/B. Storing the tree in this manner allows us to reduce h by a factor of $O(\log M/B)$ yet still meet the conditions of the lemma. We will see this approach used again in solving subproblems of the 3-d convex hull problem in Section 4.1.3.

Another way of using batch filtering, as discussed below, is by structuring more complicated decision dags as recursive constructions to get around the planarity restrictions of the lemma.

Application: Multiple-Point Planar Point Location

Planar point location is a fundamental problem in computational geometry. In the version of the problem considered here, we are given a monotone planar decomposition having N vertices, and a series of K query points. For each query point, we are to return the identifier of the region in which it lies. In main memory, this problem can be solved in optimal time $O((N + K) \log N)$ using fractional cascading [CG86a, CG86b]; $O(N \log N)$ is spent on preprocessing and $O(K \log N)$ is needed to perform the queries.

Tamassia and Vitter [TV90] have demonstrated a technique by which the fractional cascading used to solve this problem can be implemented in parallel. Their technique can solve our problem in $O((N/p + K) \log_p N)$ time on a PRAM with p processors. We can use a method based on their construction, but using M/B in place of p to get a data structure that looks like a (M/B)-ary tree augmented with catalogs. We can apply the technique of Lemma 4.2 to the main tree, but the bridge pointers connecting the catalogs make the dag non-planar. To get around this, we note that as queries traverse the edges between nodes in the main tree, they are ordered by the catalog values they query. This ordering is established at the root of the data structure, where a (M/B)-ary tree is used to locate the queries in the first catalog. The use of this approach on a fractionally cascaded tree is shown in Figure 4.4. By relying on this ordering, we can efficiently process the queries that arrive at each node of the main tree. The overall

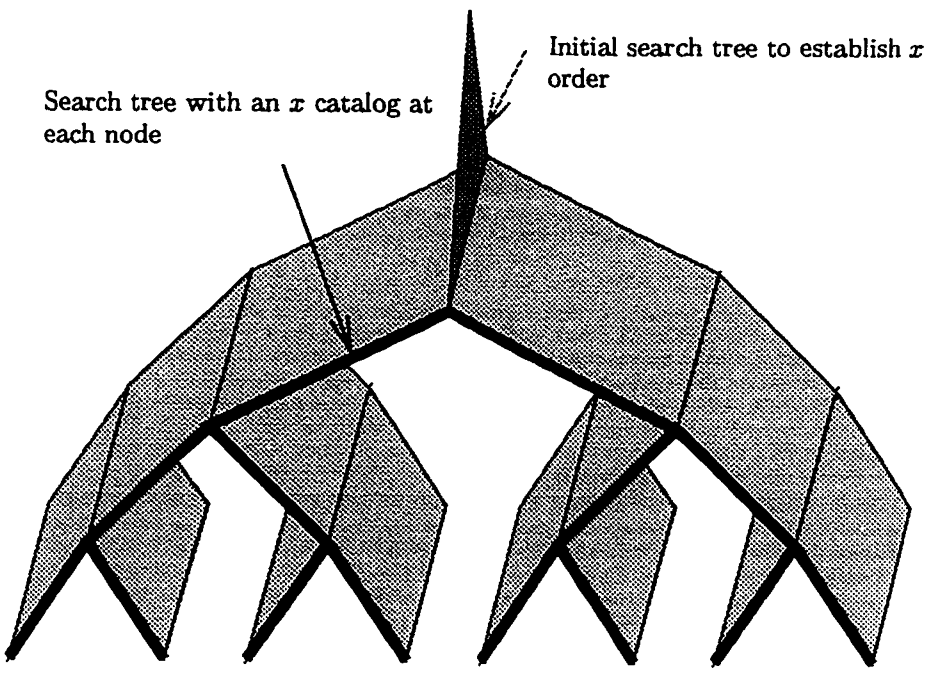
Initial search tree to establish x order
Search tree with an x catalog at each node

Figure 4.4: Batch filtering queries on a fractionally cascaded data structure. Queries are initially filtered through the search tree at the top of the figure to establish an ordering in the x direction. As they are filtered through the remainder of the data structure, this x ordering is preserved among the queries distributed to any particular node.

complexity of this technique is thereby maintained at $O((N/B + K/B) \log_{M/B}(N/B))$.

4.1.3 Convex Hull Algorithms

The convex hull problem is that of computing the smallest convex polytope completely enclosing a set of N points in d-dimensional space. In this section we will examine the problem in external memory for two and three dimensions.

Our 3-d convex hull is somewhat esoteric, so we also give a simplified version of our algorithm which, although not optimal asymptotically, is very fast in practice.

A Two-Dimensional Convex Hull Algorithm

For the two-dimensional case, a number of main memory algorithms have been developed which operate in optimal time $O(N \log N)$ [PS85]. A simple way to solve the problem optimally in external memory is to modify one of the main memory approaches, namely Graham's scan [Gra72]. Graham's scan requires that we sort the points, which can be done in $O(N/B \log_{M/B}(N/B))$ I/O operations, and then scan linearly through them, at times backtracking, but only over each input point at most once. This scanning stage can be accomplished in $O(N/B)$ I/O operations, so the overall complexity of the algorithm is $O(N/B \log_{M/B}(N/B))$.

Three-Dimensional Convex Hulls

The three-dimensional convex hull problem is interesting because of the number of two-dimensional geometric structures closely related to it, such as Voronoi diagrams and Delaunay triangulations [GS85, PS85]. In fact, by well-known reductions [GS85], our 3-d convex hull algorithm immediately gives external-memory algorithms for planar Voronoi diagrams and Delaunay triangulations with the same I/O performance. The instances of 3-d convex hulls that arise in the reduction from Voronoi diagrams has a certain structure that allows a special-purpose approach.

Even in main memory, plane sweep algorithms fail to solve the 3-d convex hull problem, and we must resort to more advanced divide and conquer approaches [PH77]. One idea is to use a plane to partition the points into equally sized sets, recursively construct the convex hull for each set, and then stitch the recursive solutions together in linear time. Unfortunately, we know no way of implementing a merging algorithm of this type in secondary memory because we cannot adequately anticipate all possible paths through the data that might be traversed during the combining phase. Another obstacle is that we need to stitch together on the order of M/B recursive solutions in linear time, rather than just two.

In order to get around the problems associated with a merging approach, we use a novel formulation of the distribution method. We consider the dual problem of computing the intersection of a set of N half spaces all of which contain the origin. Standard geometric duality transformations [PS85] are used to show the equivalence of convex hull and halfspace intersection. Once we are dealing with the dual problem, we can use a distribution-based approach along the lines of that proposed by Reif and Sen for computing 3-d convex hulls in parallel [RS92].

To illustrate, let S be a set of N halfspaces all of which contain the origin. Let the boundary of a halfspace $h_i \in S$ be denoted P_i. Suppose we have a subset $S_0 \subset S$ such that $|S_0| = N^\epsilon$. Let $I_0 = \bigcap_{h_j \in S_0} h_j$. A face of I_0 might have up to N^ϵ edges. We can reduce this complexity by trangulating each face, which can be done by sorting the vertices of I_0 along a vector not perpendicular to any face and then sweeping a plane along this sorted order. By Euler's law, the size of the resulting set of faces is at most $O(N^\epsilon)$. We can now decompose I_0 into $O(N^\epsilon)$ cones C_i, each of which has one of these faces as a base and the origin as an apex. An obvious way of distributing the halfspaces into subproblems is to create a subproblem for each cone C_i consisting of finding the intersection of all halfspaces $h_j \in S \setminus S_0$ whose bounding planes P_j intersect C_i. Unfortunately, a given P_j may intersect many cones, so it is not clear that we can continue to work through the $O(\log \log N)$ required levels of recursion without causing a very large blow up in the total size of the subproblems. Luckily, using a form of random sampling called polling and eliminating redundant planes from within a cone prior to recursion [RS92], we can, with high probability, get around this problem. In this discussion, the phrase "with high probability" means with probability $1 - N^{-\alpha}$, for some constant α.

Algorithm 4.1 is a distribution algorithm for computing the intersection of all $h_i \in S$. Step 1 can be completed with $O(N/B \log_{M/B} N/B)$ I/Os by making a linear pass through S for each sample, as suggested by Knuth [Knu81]. Step 2 consists

Halfspace Intersection
Input: A set S of N halfspaces in 3-d space.
Output: The set of all halfspaces $h_i \in S$ whose bounding planes lie on the boundary of the intersection $\bigcap_{h_j \in S_0} h_j$

1. For $j = 1$ to $\Theta(\log_{M/B} N/B)$, take a random sample S_j of S, where $|S_j| = N^{\varepsilon}$ for a constant $0 < \varepsilon < 1$.
2. Recursively solve the halfspace intersection problem on each sample S_j, giving a set of solutions I_j.
3. Use polling ([RS92]) to estimate the size of the partition of $S - S_j$ that each sample solution I_j will induce. Let S_r be the sample whose solution I_r generates the smallest such partition.
4. For each cone C_i of I_r, compute R_i, the set of halfspaces in $S - S_r$ whose boundaries intersect C_i.
5. Eliminate redundant planes from each R_i, yielding R_i^*.
6. Recursively solve the halfspace intersection problem on each set R_i^*.

Algorithm 4.1: An algorithm for computing the three dimensional convex hull of a set of points.

of recursive calls that will be considered below. In Step 3 we decompose each S_j into cones using a plane sweep. This takes $O((|S_j|/B) \log_{M/B}(|S_j|/B))$ I/Os. We then take a random sample from $S - S_j$ for each S_j. This takes $O(N/B \log_{M/B} N/B)$ I/Os. Finally, we solve a tree structured point location problem on all elements of the sample. This is done by batch filtering as described in Section 4.1.2. The number of I/O operations needed by Step 4 is $O(\frac{r}{B} \log_{M/B} \frac{r}{B})$, where $r = \sum_i |R_i|$. In Step 5, redundant planes are eliminated using a variant of the 3-d maxima algorithm from Section 4.1.1 and a 2-d convex hull algorithm. Both require $O(\frac{r}{B} \log_{M/B} \frac{r}{B})$ I/O operations. Finally, Step 6 recursively solves the subproblems.

By using methods analogous to the approach of Reif and Sen [RS92] for the parallel case, we can develop the following recurrence for the running time of our algorithm:

$$T(n) = O(N/B \log_{M/B}(N/B)) + T(N^{\varepsilon}) \log_{M/B}(N/B) + \sum_i T(|R_i|).$$

The first term on the right-hand side is the I/O cost for sampling and partitioning;

the second term is the I/O cost for sorting the samples; and the last term is for the recursive calls. In the recurrence, the $|S_i|$ terms are actually random variables. We can use Karp's method for solving probabilistic recurrence relations [Kar91] to get the solution

$$T(n) = O\left(\frac{N}{B}\log_{M/B}(N/B)\right)$$

with high probability.

The "distribution" approach used here is different from those of the distribution sort algorithms for the various I/O and memory hierarchy models [AV88, NV93a, VS94a]. In the latter cases, there are two sets of recursive calls rather than one, but the time to partition is faster.

A Practical Variation of the 3-d Convex Hull Algorithm

The 3-d convex hull algorithm discussed above can be considerably simplified by using samples of size $\Theta(M/B)$ instead of N^{ϵ}. This allows us to do the distribution of data to subproblems in a single pass, since we have enough main memory to buffer outputs for all of the recursive subproblems simultaneously.

This approach does not lead to an algorithm that is optimal in the limit, but just as was the case with sorting in Section 3.3, we find that this simpler algorithm is more efficient in practice. The I/O complexity of this simpler algorithm is

$$O\left(\frac{N}{B}\left(\frac{\lg(N/B)}{\lg(M/B)}\right)^2\right).$$

4.2 Graph-Theoretic Algorithms

In this section we will consider a novel technique for solving graph-theoretic problems in secondary memory. Our method is based on the simulation of PRAM algorithms. After introducing this technique, we will demonstrate how it can be applied to a particular problem, namely list-ranking.

4.2.1 PRAM Simulation

The efficient simulation of PRAM algorithms is a powerful technique in designing I/O efficient algorithms. In this section we demonstrate how this simulation can be performed. The fundamental idea of simulating PRAM computation in external memory is not entirely new; for example, the I/O complexity of parallel algorithms implemented using vector code has been studied [Cor92]. Our work differs in that it takes a far more direct approach to simulating parallel algorithms. The most interesting, and the most counterintuitive, of our results appears in Section 4.2.4. There we show that in order to generate I/O optimal algorithms we will, in most cases, resort to simulating PRAM algorithms that are not work-optimal. The PRAM algorithms we simulate have exponentially decreasing numbers of active processors, but very small constant factors, which is ideal for our purposes, since the I/O simulations do not need to simulate the inactive processors, and thus we get exceedingly practical I/O algorithms.

4.2.2 Generic Simulation of an $O(N)$ Space PRAM Algorithm

We begin by considering how a PRAM algorithm that uses N processors and $O(N)$ space can be simulated. In order to simulate such a PRAM algorithm, we first consider how to simulate a single PRAM step. This process requires only sorting and scanning, as is shown in the following lemma:

Lemma 4.3 *Let A be a PRAM algorithm that uses N processors and $O(N)$ space. Then a single step of A can be simulated in*

$$O\left(\frac{N}{B}\frac{\lg N/B}{\lg M/B}\right)$$

I/Os.

Proof: To simulate the PRAM memory, we keep an array of size $O(N)$ on disk in $O(N/B)$ blocks. In a single step, each PRAM processor reads $O(1)$ operands from memory, performs some computation, and then writes $O(1)$ results to memory. Let a be the maximum number of operands any one processor reads. Using $O(N/B)$ I/Os, we can make a copies of the contents of the PRAM memory. To provide the operands for each processor operation we must simulate, we sort each of the copies of the PRAM memory.

The first copy of the PRAM memory contents is sorted so that the first operand for the first processor comes first, the first operand for the second processor comes second, and so on. The second copy of the PRAM memory contents is sorted so that the second operand for the first processor comes first, the second operand for the second processor comes second, and so on. We sort all a copies of the memory in a similar manner, so that we have a lists of operands sorted by the processors to which they belong.

Note that we have implicitly assumed that each item in the PRAM memory is an operand for exactly one processor. If this is not the case, then in the scan that makes copies of the memory contents, we can duplicate the contents of some memory locations and skip over the contents of others to provide the appropriate number of copies of each memory location. Sorting is then used, as above, to move them to their appropriate locations.

Whether or not duplication is required, the total I/O complexity of generating the $a = O(1)$ operand lists is

$$O\left(\frac{N}{B}\frac{\lg N/B}{\lg M/B}\right).$$

Once the operand lists have been constructed, we scan them sequentially, proceeding processor by processor, reading the operands of the operation to take place at that processor, and writing the result of the operation the processor performs to a buffered output. This process takes $O(N/B)$ I/Os and leaves us with the results computed by each of the processors, in order by processor number. To return these results to the simulated PRAM memory, we do the reverse of the process that was used to fetch operands for the processors. We sort the results by the memory locations that they should be written to and then scan the sorted results and the simulated contents of the PRAM memory to write them back. This process has the same asymptotic complexity as sorting the operands did, giving us an overall I/O complexity of

$$O\left(\frac{N}{B}\frac{\lg N/B}{\lg M/B}\right).$$

□

To simulate an entire algorithm, we have to simulate all of its steps. Doing so gives us the following theorem as a corollary of Lemma 4.3:

Theorem 4.4 *Let A be a PRAM algorithm that uses N processors and $O(N)$ space and runs in time T. Then A can be simulated in $O(T\frac{N}{B}\log_M\frac{N}{B})$ I/Os.*

4.2.3 Simulation of Super-Linear Space PRAM Algorithms

We say that a parallel algorithm A is *dependency-aware* if, for memory cell c, we know the time step c is first accessed, and, in addition, for any memory access of c by some processor p in step i, p can determine the next step when c will need to be accessed.

Let the *carrying cost* C_i for Step i of a PRAM algorithm A be defined to be the number of memory cells referenced in Step i. Let C, the carrying cost for A, be the sum of carrying costs for all the steps in A.

Theorem 4.5 *Let A be a dependency-aware PRAM algorithm running in time T using W work with carrying cost C. Then A can be simulated sequentially in external memory using $O(sort(W+C)+T^2)$ i/o operations.*

4.2.4 Reduced Work Simulation for Geometrically Decreasing Computations

Our main approach to designing efficient I/O algorithms is to concentrate on simulating PRAM algorithms that have the "geometrically decreasing size" property, in that after a constant number of steps, the number of active processors has decreased by a constant factor. Such algorithms are typically not work-optimal in the PRAM sense, since PRAM measures of work typically count inactive processors. But for I/O purposes, inactive processors do not have to be simulated, and hence our definition of work is motivated by the I/O simulation. The main theorem we use for simulating PRAM algorithms with geometrically decreasing size is given below:

Theorem 4.6 *Let A be a PRAM algorithm that solves a problem of size N by using N processors and $O(N)$ space, and that after each of $O(\log N)$ stages, each of constant time, both the number of active processors and the number of memory cells that will be used again in the computation are reduced by a constant factor. That is, there exists a constant $0 < \alpha < 1$ such that after 1 stage, only αN processors and $O(\alpha N)$ memory locations are active, after 2 stages α^2 processors and $O(\alpha^2 N)$ memory locations are active, and so on. Then A can be simulated in external memory in $O(\frac{N}{B} \log_M \frac{N}{B})$ I/O operations.*

Proof: The first stage consists of $O(1)$ steps, each of which can, by Lemma 4.3, be simulated in $O(\frac{N}{B} \log_M \frac{N}{B})$ I/Os. Once we enter the second stage, we are essentially simulating an algorithm that runs on αN processors and uses $O(\alpha N)$ space and has the same geometrically decreasing behavior. We can thus write a recurrence for the number of I/Os needed to simulate the algorithm which looks like

$$T(N) = O\left(\frac{N}{B} \log_M \frac{N}{B}\right) + T(\alpha N).$$

The base case of this recurrence occurs when $N = o(M)$, in which case $T(N) = O(N/B)$, since we need only read the inputs into main memory, solve the problem in main memory, and write the results out. We can now prove by induction that

$$T(N) = O(\frac{N}{B} \log_M \frac{N}{B}).$$

In order to solve the recurrence, we note that by the definition of big-Oh notation, there exists some positive constant C_1 such that for all sufficiently large N,

$$T(N) \leq C_1 \frac{N}{B} \log_M \frac{N}{B} + T(\alpha N).$$

Let N^* be the smallest such N for which this inequality holds. Let

$$C_2 = \max \left\{ \frac{C_1}{1-\alpha} \right\} \cup \bigcup_{N^* \leq i \leq N^*/\alpha} \frac{T(i)}{\frac{i}{B} \log_M \frac{i}{B}}.$$

We can now prove by induction that for all $N \geq N^*$,

$$T(N) \leq C_2 \frac{N}{B} \log_M \frac{N}{B}. \tag{4.1}$$

The base cases are those between $N = N^*$ and $N = N^*/\alpha$, for which (4.1) holds by the definition of C_2. For $N > N^*/\alpha$, we rely on the inductive hypothesis that for all N between N^* and $N-1$, (4.1) holds. This means, in particular, that it holds for αN. Thus,

$$\begin{aligned} T(N) &\leq C_1 \frac{N}{B} \log_M \frac{N}{B} + T(\alpha N) \\ &\leq C_1 \frac{N}{B} \log_M \frac{N}{B} + C_2 \frac{\alpha N}{B} \log_M \frac{\alpha N}{B} \end{aligned}$$

$$= (C_1 + C_2\alpha)\frac{N}{B}\log_M \frac{N}{B} + C_2\alpha\frac{N}{B}\log_M \alpha.$$

Since α is between 0 and 1, $\log_M \alpha$ is negative, and thus

$$T(N) \leq (C_1 + C_2\alpha)\frac{N}{B}\log_M \frac{N}{B}. \tag{4.2}$$

By definition, $C_2 \geq C_1/(1-\alpha)$. Thus $C_1 \leq (1-\alpha)C_2$ and $C_1 + C_2\alpha \leq C_2$. Plugging this into (4.2), we get

$$T(N) \leq C_2\frac{N}{B}\log_M \frac{N}{B},$$

proving the inductive hypothesis for N.

Since, for all $N \geq N^*$,

$$T(N) \leq C_2\frac{N}{B}\log_M \frac{N}{B},$$

we have, by definition,

$$T(N) = O\left(\frac{N}{B}\log_M \frac{N}{B}\right).$$

□

4.3 List Ranking

In this section, we demonstrate how the PRAM simulation technique of Section 4.2.1 can be used to construct an algorithm for the problem of list ranking.

The input to list ranking is an N-node linked list L stored in external memory as an (unordered) sequence of nodes, each with a pointer *next* to the successor node in the list. Our goal is to determine, for each node v of L, the *rank* of v, i.e., the distance *rank*(v) of node v to the last node of L.

Our result is

Lemma 4.7 *List ranking of an input of N nodes can be solved in*

$$O\left(\frac{N}{B}\log_M \frac{N}{B}\right)$$

I/Os.

Our algorithmic framework is adapted from the work of Anderson and Miller [AM90]. It has also been used by Cole and Vishkin [CV86], who developed a deterministic version of Anderson and Miller's randomized algorithm.

Initially, we assign to each node v in list L $rank(v) = 1$. This can be done in $O(scan(N))$ I/Os. We then proceed recursively. First, we produce an independent set of $\Theta(N)$ nodes. Once we have a large independent set, we *bridge* each node v in the set. By bridging v, we mean that we replace pointer $next(u)$ to v with $next(v)$ and replace $rank(u)$ with $rank(u) + rank(v)$. Bridging can be done with $O(1)$ sorts and scans. We then recurse, ranking the list from which elements of the independent set have been bridged. When we return from the recursion, we reinsert each node that was bridged out, incrementing its rank by the rank of its predecessor after the recursion returned. This can also be done with $O(1)$ sorts and scans.

In order to analyze the I/O complexity of an algorithm of the type just described, we first note that all the operations performed before and after recursion use $O(\frac{N}{B}\log_M \frac{N}{B})$ I/Os. If the independent set can be found in $O(\frac{N}{B}\log_M \frac{N}{B})$ I/Os, then the overall I/O complexity of the algorithm also meets this bound. Now, since $\Theta(N)$ nodes are bridged out before recursion, the size of the recursive problem we are left with is at most a constant fraction of the size of our original problem. Thus, according to Theorem 4.6 the I/O complexity of our overall algorithm is $O(\frac{N}{B}\log_M \frac{N}{B})$. Now all that remains is to demonstrate how an independent set of size $\Theta(N)$ can be produced in $O(\frac{N}{B}\log_M \frac{N}{B})$ I/Os.

We generate the independent set using an adaptation of an algorithm developed by Anderson and Miller [AM90]. In this algorithm, we flip a fair coin for each vertex v in the chain. If v gets heads and the successor of v gets tails, then v is included in the independent set. This method of selection ensures that it is impossible for two selected nodes to be adjacent in the list. The expected size of the independent set generated this way is $(N-1)/4$.

Implementing this method of independent set construction in external memory within the required $O(\frac{N}{B}\log_M \frac{N}{B})$ I/Os is done by using an optimal sorting algorithm as discussed in Section 3.2. Before doing so, we scan through the nodes (in $O(N/B)$ I/Os) and assign each a random value 0 or 1 based on the outcome of a coin flip. We then make two copies of the list, again using $O(N/B)$ I/Os. Next, we sort the first copy of the list by the id of each node and the second copy by the *next* fields of the node. This takes $O(\frac{N}{B}\log_M \frac{N}{B})$ I/Os. Finally, by scanning the two lists, which now have opposite ends of each edge in the corresponding position, we can determine exactly which nodes are part of the independent set. This takes $O(N/B)$ I/Os. Putting all the pieces together, we have the following lemma:

Lemma 4.8 *An independent set of expected size* $(N-1)/4$ *can be constructed in* $O(\frac{N}{B}\log_M \frac{N}{B})$ *I/Os.*

Lemma 4.8 can now be combined with our earlier analysis to prove Lemma 4.7.

A more detailed analysis of the performance of list ranking, paying particular attention to the details of implementing it, is presented in Section 7.2.

4.4 Conclusions

We have demonstrated the existence of I/O-efficient algorithms for a number of important combinatorial problems. The fundamental building blocks we took advantage of are sorting and scanning, which were shown to be I/O-efficient in Chapter 3. In the next chapter, we will consider scientific computations, which are generally considered to have a much more regular structure than those we considered here. Readers interested in more details on the implementation of the algorithms discussed in this chapter may wish to skip directly to Chapter 7 where implementation is discussed in detail.

Chapter 5

Algorithms for Scientific Computation

Scientific computations often require the management of tremendous amounts of data. Often, this data is stored in matrix form. Thus, in order to support large-scale scientific computations, we must consider the I/O complexity of solutions to various fundamental problems involving this sort of data. In this chapter, we consider algorithms for these problems. Our discussion in broken into two parts; Section 5.1 discusses algorithms for sparse matrices, and Section 5.2 discussed algorithms for dense matrices. Despite the regular structure of these problems, we find that judicious permutation of input data is the key to producing efficient algorithms, just as it was for the combinatorial algorithms discussed in Chapter 4. [1]

5.1 Sparse Matrix Methods

Sparse matrix methods are widely used in scientific computations, particularly in iterative methods. A fundamental operation on sparse matrices is that of multiplying a

[1]The work presented in this chapter was originally presented in [VV95b] and [VV95a]. It is joint work with Jeffrey Vitter.

```
(1)  $z \leftarrow 0$;
(2)  foreach nonzero element $e$ of $A$ do
(3)          $z[\text{row}(e)] = z[\text{row}(e)] + \text{value}(e) \times x[\text{col}(e)]$;
(4)  end
```

Algorithm 5.1: Generic sparse matrix/vector multiplication. This is an algorithm for computing $z = Ax$ where A is a sparse $N \times N$ matrix and x and z are N-vectors.

sparse $N \times N$ matrix A by an N-vector x to produce an N-vector $z = Ax$.

Before we can work with sparse matrices in secondary memory, we need a way of representing them. In the algorithms we consider, a sparse matrix A is represented by a set of nonzero elements E. Each $e \in E$ is a triple whose components are row(e), the row index of e in A, col(e), the column index of e in A, and value(e), the value of $A[\text{row}(e), \text{col}(e)]$.

In main memory, sparse matrix-vector multiplication can be implemented using Algorithm 5.1. If the number of nonzero elements of A is N_z, then Algorithm 5.1 runs in $O(N_z)$ time on a sequential machine.

In secondary memory, we can also use Algorithm 5.1, but I/O performance depends critically on both the order in which the elements of A are processed and which of components of z and x are in main memory at any given time. In the worst case, every time we reference an object it could be swapped out. This worst-case scenario would result in $3N_z$ I/Os being performed.

In order to guarantee I/O-efficient computation, we reorder the elements of A in a preprocessing phase. In this preprocessing phase, A is divided into N/M separate $M \times N$ sub-matrices A_i, called bands. Band A_i contains all elements of A from rows iM to $(i+1)M-1$ inclusive. Although the dimensions of all the A_i are the same, the number of nonzero elements they contain may vary widely. To complete the preprocessing, the elements of each of the A_i are sorted by column.

Once A is preprocessed into bands, we can compute the output sub-vector $z[iM \ldots (i+1)M-1]$ from A_i and x using a single scan, as shown in Algorithm 5.2. If we ignore the preprocessing phase for a moment and assume that the elements of x appear in order in external memory, the I/O complexity of Algorithm 5.2 is

$$\frac{N_z}{DB} + \left\lceil \frac{N}{M} \right\rceil \frac{N}{DB} + \frac{N}{DB}.$$

The entire preprocessing phase can be implemented as a single sort on the nonzero elements of A, using band number as the primary key and column as a secondary key. This takes $2N_z/DB \left\lceil \frac{\lg(N_z/M)}{\lg(M/2DB)} \right\rceil$ I/Os, as discussed in Section 3.2. Note, however, that the preprocessing only has to be done once for a given matrix A. After that, the main phase of the algorithm can be executed repeatedly for many different vectors x. This is a common occurrence in iterative methods.

The multiplication of a banded sparse matrix and a vector is illustrated in Figure 5.1. In this illustration, $B = 5$ and $M = 35$. These numbers are much smaller than in typical systems for the sake of illustration. We have enough main memory to hold one input buffer for the banded sparse matrix, one input buffer for the vector, and five blocks full of output results. The black squares in the matrix represent non-zero entries. The shaded portions of the figure are those involved in multiplying the first band of the matrix by the vector. The first band is blocked horizontally into blocks of $B = 5$ non-zero entries. We scan across this band and down the vector, as indicated by the arrows, in order to generate results for the shaded portion of the result vector. When this is complete, we write these results out and repeat the process for each of the subsequent bands in the matrix.

The performance of this algorithm is discussed in the context of two benchmark programs in Section 8.3.

```
      // Preprocessing phase:

(1)  foreach nonzero element $e$ of $A$ do
(2)          Put $e$ into $A_{\lfloor \mathrm{row}(e)/M \rfloor}$;
(3)  endforeach
(4)  for $i \leftarrow 0, N/M$ do
(5)          Sort the elements of $A_i$ by column;
(6)  endfor

      // Main algorithm:

(7)  Allocate a main memory buffer $z_M$ of $M$ words;
(8)  for $i \leftarrow 0$ to $\lceil N/M \rceil$ do
(9)          $z_M \leftarrow 0$;
(10)         foreach nonzero element $e$ of $A_i$ do
(11)                 $z_M[\mathrm{row}(e) - iM] = z_M[\mathrm{row}(e) - iM] + \mathrm{value}(e) \times x[\mathrm{col}(e)]$;
(12)         endforeach
(13)         Write $z_M$ to $z[iM \ldots (i+1)M - 1]$;
(14) endfor
```

Algorithm 5.2: I/O-efficient sparse matrix/vector multiplication. This algorithm computes $z = Ax$ where A is a sparse $N \times N$ matrix and x and z are N-vectors.

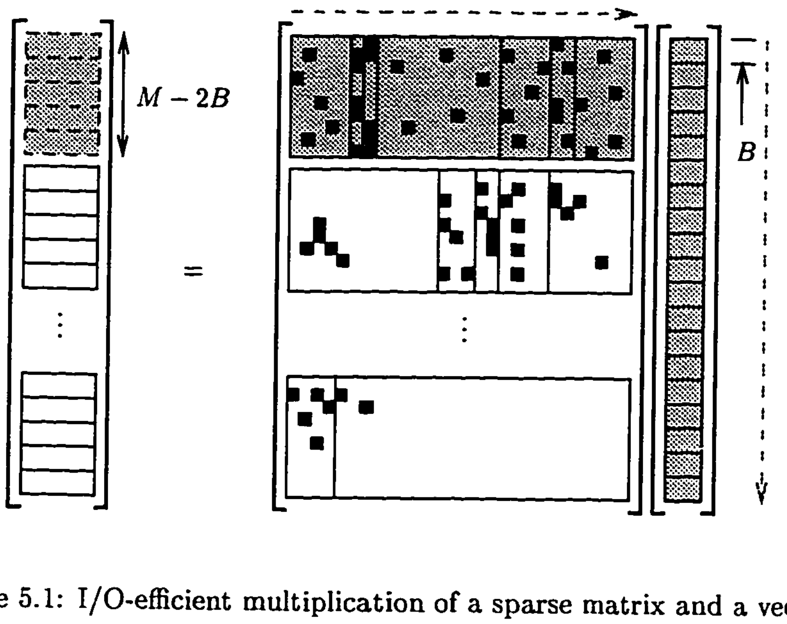

Figure 5.1: I/O-efficient multiplication of a sparse matrix and a vector.

```
(1)  if $3K^2 \le M$ then
(2)      read $A$ and $B$ into main memory;
(3)      compute $C = AB$ in main memory;
(4)      write $C$ back to disk;
(5)  else
(6)      partition $A$ at row $K/2$ and column $K/2$;
(7)      label the four quadrant sub-matrices $A_{1,1}$, $A_{1,2}$, $A_{2,1}$, and $A_{2,2}$ as shown in Figure 5.2;
(8)      partition $B$ into $B_{1,1}$, $B_{1,2}$, $B_{2,1}$, and $B_{2,2}$ in a similar manner;
(9)      permute all sub-matrices of $A$ and $B$ into row major order;
(10)     Perform the (11)-(14) using recursive invocations of this algorithm
(11)         $C_{1,1} \leftarrow A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$;
(12)         $C_{1,2} \leftarrow A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$;
(13)         $C_{2,1} \leftarrow A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$;
(14)         $C_{2,2} \leftarrow A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$;
(15)     Reconstruct $C$ from its sub-matrices $C_{1,1}$, $C_{1,2}$, $C_{2,1}$, and $C_{2,2}$;
(16)     permute $C$ back into row major order;
(17) endif
```

Algorithm 5.3: Two $K \times K$ input matrices A and B are multiplied to produce $C = AB$ using a divide and conquer approach.

5.2 Dense Matrix Methods

Dense matrices appear in a variety of computations. Like sparse matrices, they are often multiplied by vectors. In this case banding techniques similar to those discussed in the previous section can by used. Another fundamental operation is multiplication of two $K \times K$ matrices A and B to produce $C = AB$.

Asymptotically I/O-optimal multiplication of two $K \times K$ matrices over a quasiring can be done in $\Theta(K^3/\sqrt{M}DB)$ I/Os [VS94a]. The lower bound was devised by Aggarwal and Vitter [AV88]. A matching upper bound for parallel disk systems was presented by Vitter and Shriver [VS94a]. The algorithm Vitter and Shriver presented uses a recursive divide and conquer approach. It is shown as Algorithm 5.3.

Vitter and Shriver presented an asymptotic analysis of the I/O performance of Algorithm 5.3 in their work. In the following, we refine this analysis to include constant

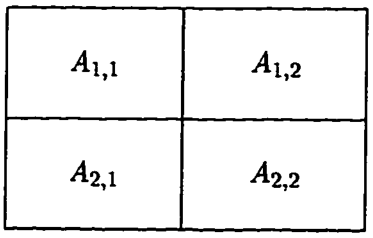

Figure 5.2: Partitioning a matrix into quadrants in order to perform matrix multiplication recursively using Algorithm 5.3. If two matrices A and B are partitioned in this manner, then the four quadrants of their product $C = AB$ can be computed by pairwise multiplication of their quadrants and addition. For example, we can compute $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$.

factors. First, we consider the base case where $3K^2 \leq M$. In this case, step (2) of the algorithm reads both A and B in their entirety, which requires $2K^2/(DB)$ I/Os. Step (4) then writes the result C back to disk, which requires $K^2/(DB)$ I/Os. A total of $3K^2/(DB)$ I/Os are thus required.

In the more general case when $3K^2 > M$, steps (6)–(9) partition and permute the two input matrices into four equal sized square submatrices. Assuming that $M > 3DB$, this step can be done in a single pass through each of the two input matrices. For the first $K/2$ rows of A, one input buffer is needed for A and two output buffers are needed, one each for $A_{1,1}$ and $A_{1,2}$. Once $A_{1,1}$ and $A_{1,2}$ have been written, we use the same input buffer for A and re-use the two output buffers for $A_{2,1}$ and $A_{2,2}$. Since the input matrices are in row major order, we scan through their elements, writing each element we read to the output buffer associated with the submatrix to which the element belongs. The submatrices are written in row major order, as required for recursive invocations of the algorithm. This is because the original input was in row major order, and thus all elements of a submatrix are already in a relative order consistent with a row major ordering of the elements of the submatrix. The total number of I/Os required to partition a matrix (A or B) in this manner is thus $2K^2/(DB)$. After four recursive invocations of the algorithm, we reconstruct a row major representation of

C from its four submatrices by reversing the process used to partition A and B in the first place.

The I/O complexity of Algorithm 5.3 is within a constant factor of optimality. Its exact I/O complexity is given by the following Lemma:

Lemma 5.1 *Assuming $M > 3DB$, the I/O complexity of Algorithm 5.3 for a problem instance involving $K \times K$ matrices is*

$$T(K) = \frac{12\sqrt{3}K^3}{DB\sqrt{M}} - \frac{9K^2}{DB}.$$

Proof: In order to prove this lemma, we construct and solve a recurrence for $T(K)$.

In the base case, when $3K^2 \leq M$, we read both inputs, perform all computation in main memory, and write the results. Since two matrices are read and one is written, with all I/O fully blocked and striped, the total I/O complexity is $3K^2/DB$.

For larger matrices, there are several different types of I/O that must be performed. First, we must partition the two input matrices into four submatrices each. If we assume that $M > 3DB$, this partitioning can be done in its entirety in a single scan through each input. One input buffer of size DB is used to buffer the input and two are used for buffering the output. While the first $K/2$ rows of A are being read, the two output buffers are used for $A_{1,1}$ and $A_{1,2}$. When the second half of the rows are being read, the output buffers are used for $A_{2,1}$ and $A_{2,2}$.[2] The total amount of I/O used in partitioning a matrix in this manner is $2K^2/DB$.

Once A and B are partitioned, we make eight recursive invocations of our algorithm, which costs us $8T(K/2)$ I/Os. We then do four element-wise additions, each of which requires $3(K/2)^2$ I/Os. Finally, we reconstruct C from its four submatrices, which has the same I/O cost as partitioning A or B did. Summing up these costs, and combining

[2]In reality, it is hard to imagine a machine in which M was not significantly larger than $3DB$. On a machine with much more main memory capacity than $3DB$, we would probably take advantage of the additional main memory to double buffer I/O or perhaps to read and write multiple contiguous blocks simultaneously. This would reduce seek overhead, as described in Section 3.2.

them with the cost of multiplying small matrices, we get the recurrence

$$T(K) = \begin{cases} \dfrac{3K^2}{DB} & \text{if } 3K^2 \leq M; \\ \\ \dfrac{9K^2}{DB} + 8T(K/2) & \text{otherwise.} \end{cases}$$

In order to solve this recurrence, let

$$S_i = \frac{DB}{M} T\left(2^i \frac{\sqrt{M}}{\sqrt{3}}\right) \tag{5.1}$$

be defined for all non-negative integers i. Then

$$S_0 = \frac{DB}{M} T\left(\frac{\sqrt{M}}{\sqrt{3}}\right) = 1.$$

For $i > 0$,

$$\begin{aligned} S_i &= \frac{DB}{M} T\left(2^i \frac{\sqrt{M}}{\sqrt{3}}\right) \\ &= \frac{DB}{M}\left(\frac{9 \cdot 4^i M}{3DB} + 8T\left(2^{i-1}\frac{\sqrt{M}}{\sqrt{3}}\right)\right) \\ &= 3 \cdot 4^i + 8S_{i-1}. \end{aligned}$$

The solution to this recurrence is

$$S_i = 4 \cdot 8^i - 3 \cdot 4^i$$

for all $i \geq 0$, which can be shown by induction.

We now have a solution to the recurrence for S_i, and thus can compute the value of $T(K)$ for values of K of the form $K = 2^i\sqrt{M}/\sqrt{3}$. We do this by solving (5.1) for

$T(K)$ to get

$$\begin{aligned} T(K) &= T\left(2^i\frac{\sqrt{M}}{\sqrt{3}}\right) \\ &= \frac{M}{DB}S_i \\ &= \frac{M}{DB}\left(4\cdot 8^i - 3\cdot 4^i\right). \end{aligned}$$

For the values of K we are interested in, $2^i = K\sqrt{3}/\sqrt{M}$, and thus $4^i = K^2\frac{3}{M}$ and $8^i = K^3\frac{3\sqrt{3}}{M\sqrt{M}}$. Substituting these into our expression for $T(K)$, we get

$$\begin{aligned} T(K) &= \frac{M}{DB}\left(4K^3\frac{3\sqrt{3}}{M\sqrt{M}} - 3K^2\frac{3}{M}\right) \\ &= \frac{12\sqrt{3}K^3}{DB\sqrt{M}} - \frac{9K^2}{DB}. \end{aligned}$$

□

Although Algorithm 5.3 is within a constant factor of optimality, it is possible to modify the algorithm to produce an iterative algorithm that is substantially more efficient for large matrices. The key idea is to permute the elements of the matrices A and B only once, rather than over and over again at each level of recursion. After this initial permutation, we have a set of submatrices that we can multiply iteratively to generate submatrices of the final solution matrix C. These submatrices can then be reassembled into a row major representation of C by a single permutation. The permutations required to generate an appropriate set of submatrices for this approach are significantly more complicated than the single pass permutation Algorithm 5.3 used to generate four submatrices at each level of recursion, but as will be seen, the overall I/O complexity of this approach is significantly less. The iterative approach is shown

```
(1)  partition $A$ into $K/\sqrt{M/3}$ rows and $K/\sqrt{M/3}$ columns of
         sub-matrices each having $\sqrt{M/3}$ rows and $\sqrt{M/3}$ columns;
     // *step (1) is shown in Figure 5.3*
(2)  partition $B$ in a manner similar to $A$;
(3)  permute all $A_{i,j}$ and $B_{i,j}$ into row major order;
(4)  **foreach** $i, j$ **do**
(5)      $C_{i,j} \leftarrow \sum_k A_{i,k}B_{k,j}$;
(6)  **endforeach**
(7)  reconstruct $C$ from all $C_{i,j}$;
(8)  permute $C$ back into row major order;
```

Algorithm 5.4: Iterative matrix multiplication. A preprocessing permutation is used to establish a representation of the matrix that is amenable to an iterative approach.

in Algorithm 5.4.

For the moment, let us ignore the permutations required by Algorithm 5.4 and analyze only the iterative portion consisting of steps (4)–(6). The index variables i, j, and k all range over $K/\sqrt{M/3}$ values. Each invocation of step (5) consists of adding $K/\sqrt{M/3}$ terms, each of which is the product of two $(\sqrt{M/3}) \times (\sqrt{M/3})$ matrices. One third of the main memory is devoted to holding each of the two matrices whose product makes up a term of the sum. The final third of main memory is used to hold a running sum of the terms as they are processed. In order to save main memory we never explicitly store the product $A_{i,k}B_{k,j}$ that makes up a term of the sum. We are able to get away with doing this because each element of such a term is the sum of a series of products of elements of $A_{i,k}$ and $B_{k,j}$. Since $C_{i,j}$ is the element-wise sum of these terms and addition is commutative, we can add each elemental product directly to the element of $C_{i,j}$ that it will eventually be added to, rather than ever explicitly representing the term $A_{i,k}B_{k,j}$.

The I/O complexity of a single invocation of step (5) is thus the I/O complexity of reading $2k/\sqrt{M/3}$ submatrices and writing one. Each submatrix can be read of

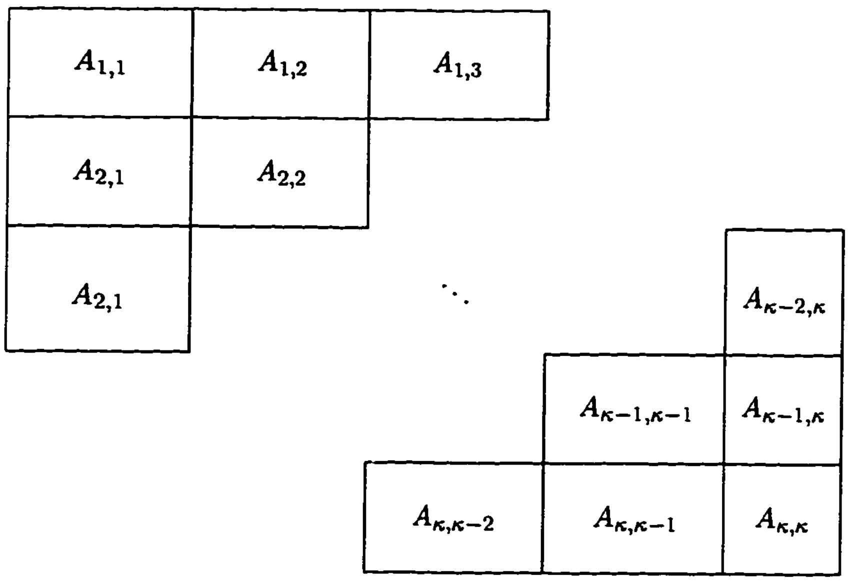

Figure 5.3: Partitioning a matrix into sub-matrices in step (1) of Algorithm 5.4. Each sub-matrix $A_{i,j}$ has $\sqrt{M/3}$ rows and $\sqrt{M/3}$ columns. The number of sub-matrices across and down A is $\kappa = K/\sqrt{M/3}$.

written in $M/(3DB)$ I/Os, giving an invocation of step (5) an overall I/O complexity of

$$\left(\frac{2k}{\sqrt{M/3}}+1\right)\frac{M}{3DB}.$$

Summing over all $(K/\sqrt{M/3})^2 = 3K^2/M$ invocations of step (5), we get an I/O complexity of

$$\frac{2\sqrt{3}K^3}{\sqrt{M}DB}+\frac{K^2}{DB}.$$

Now, let us turn our attention to the permutations required in steps (1)–(3) and (7)–(8). In special circumstances, when B, D, K, and $\sqrt{M}$ are all integral powers of two, these permutations are bit-matrix multiply complement permutations [CSW94]. In more general circumstances, however, they may have to be implemented by sorting. In either case, the complexity of sorting, as discussed in Section 3.2, provides a strong upper bound on the number of I/Os that are required to route each of the three permutations. If striped merge sorting is used, then, according to (3.1) each permutation of $K^2/3$ elements each can be done in

$$\frac{2K^2}{3DB}\left(1+\left\lceil\frac{\lg\frac{K^2}{3M}}{\lg\left(\frac{M}{2DB}-1\right)}\right\rceil\right)$$

I/Os. Thus, the total I/O complexity of all three permutations is

$$\frac{2K^2}{DB}\left(1+\left\lceil\frac{\lg\frac{K^2}{3M}}{\lg\left(\frac{M}{2DB}-1\right)}\right\rceil\right).$$

The total I/O complexity of Algorithm 5.4 is thus

$$\frac{2\sqrt{3}K^3}{\sqrt{M}DB}+\frac{3K^2}{DB}+\frac{2K^2}{DB}\left\lceil\frac{\lg\frac{K^2}{3M}}{\lg\left(\frac{M}{2DB}-1\right)}\right\rceil.$$

In the limit as N gets large, it is clear that the iterative algorithm is superior, since the leading term of its I/O complexity is a factor of 6 smaller than that of the recursive algorithm. For small matrices, however, it might be the case that lower order terms play a significant enough role to counter this fact. This is reasonable to expect in this case, since the iterative algorithm has several lower order terms with positive coefficients, whereas the recursive algorithm has a single lower order term with a negative coefficient.

What we are asking is under what circumstances is it the case that

$$\frac{12\sqrt{3}K^3}{\sqrt{M}DB}-\frac{9K^2}{DB}<\frac{2\sqrt{3}K^3}{\sqrt{M}DB}+\frac{3K^2}{DB}+\frac{2K^2}{DB}\left\lceil\frac{\lg(K^2/3M)}{\lg(M/2DB-1)}\right\rceil? \qquad (5.2)$$

When this inequality holds, the recursive algorithm is better; when it does not hold, the iterative algorithm is better. Simplifying, we find that this is precisely when

$$\frac{10\sqrt{3}K}{\sqrt{M}} < 12 + 2\left\lceil \frac{\lg(K^2/3M)}{\lg(M/2DB-1)} \right\rceil,$$

or equivalently when

$$K < \frac{\sqrt{M}}{10\sqrt{3}}\left(12 + 2\left\lceil \frac{\lg(K^2/3M)}{\lg(M/2DB)} \right\rceil\right) = \frac{6\sqrt{M}}{5\sqrt{3}} + \frac{\sqrt{M}}{5\sqrt{3}}\left\lceil \frac{\lg(K^2/3M)}{\lg(M/2DB-1)} \right\rceil. \qquad (5.3)$$

An additional fact about the two algorithms that we need to consider is that the I/O complexities we derived for them only apply for matrices with $K > \sqrt{M}/\sqrt{3}$. If the matrices are smaller than this, then the algorithms behave equivalently with respect to I/O, since both input matrices and the result fit entirely in main memory. Combining this with the upper bound (5.3), we find that the recursive algorithm is superior only for the very narrow band of matrix sizes where

$$\frac{\sqrt{M}}{\sqrt{3}} < K < \frac{6\sqrt{M}}{5\sqrt{3}} + \frac{\sqrt{M}}{5\sqrt{3}}\left\lceil \frac{\lg(K^2/3M)}{\lg(M/2DB)} \right\rceil.$$

Part III

TPIE

Chapter 6

TPIE, a Transparent Parallel I/O Environment

TPIE, a transparent parallel I/O environment, is designed to bridge the gap between the theory and practice of parallel I/O systems. TPIE is intended to demonstrate that a parallel I/O system can do all of the following simultaneously:

- Abstract away the details of how I/O is performed so that programmers need only deal with a simple high level interface.
- Implement I/O-optimal paradigms for large scale computations that are efficient not only in theory, but also in practice.
- Remain flexible, allowing programmers to specify the functional details of computation taking place within the supported paradigms. This will allow a wide variety of algorithms to be implemented within the system.
- Be portable across a variety hardware platforms.
- Be extensible, so that new features can be easily added later.

TPIE is targeted at supporting the various algorithms discussed in Chapters 3, 4, and 5.

TPIE is implemented as a set of templated classes and functions in C++. It also includes a small library and a set of test and sample applications. TPIE has been tested on a variety of hardware platforms with a variety of flavors of UNIX operating systems. Combinations that have been tested include:

- Sun Sparcstation/SunOS 4.x
- Sun Sparcstation/Solaris 5.x
- DEC Alpha/OSF/1 1.x and 2.x
- HP 9000/HP-UX
- Intel Pentium/Linux 1.x

This chapter discusses the structure of TPIE and its programmers' interface.

6.1 The Structure of TPIE

TPIE has three main components, the Block Transfer Engine (BTE), the Memory Manager (MM), and the Access Method Interface (AMI). The BTE handles block transfer for a single processor. The MM performs low level memory management across all the processors in the system. The AMI works on top of the MM and one or more BTEs, each running on a single processor, to provide a uniform interface for application programs. Applications that use this interface are portable across hardware platforms, since they never have to deal with the underlying details of how I/O is performed on a particular machine.

The BTE is intended to bridge the gap between the I/O hardware and the rest of our system. It works alongside the traditional buffer cache in a UNIX system. Unlike the buffer cache, which must support concurrent access to files from multiple address spaces, the BTE is specifically designed to support high throughput processing of data from secondary memory through a single user level address space. In order to efficiently support the merging, distribution, and scanning paradigms, the BTE provides stream oriented buffer replacement policies. To further improve performance, some implementations of the BTE move data from disk directly into user space rather than using a kernel level buffer cache. This saves both main memory space and copying time.

The MM manages random access memory on behalf of TPIE. It is the most architecture-dependent component of the system. On a single processor or multiprocessor system with a single global address space, the MM is relatively simple; its task is to allocate and manage the physical memory used by the BTE. On a distributed memory system, the MM has the additional task of coordinating communication between processors and memory modules in order to support the primitives that the AMI provides.

The AMI is a layer between the BTE and user level processes. It implements fundamental access methods, such as scanning, permutation routing, merging, distribution, and batch filtering. It also provides a consistent, object-oriented interface to application programs. The details of how these access methods are implemented depends on the hardware on which the system is running. For example, recursive distribution will be done somewhat differently on a parallel disk machine than on a single disk machine. The AMI abstracts this fact away, allowing an application program that calls a function such as `AMI_partition_and_merge()` to work correctly regardless of the underlying I/O system.

The key to keeping the AMI simple and flexible is that TPIE is a framework-oriented system, as described in Section 1.3.2. The AMI's user accessible functions serve more as templates for computation than as actual problem solving functions. The details of how a computation proceeds within the template is up to the application programmer, who is responsible for providing the functions that the template applies to data.

6.2 Streams

Conceptually, TPIE programs work with streams of data stored on external memory. A stream is an ordered collection of objects of a particular type. The current implementation of TPIE builds streams using file system meta-data as described in Section 2.4.3. Various paradigms of computation are defined on these streams, though the functional details of the computation performed within these paradigms is left to the TPIE programmer to specify. These details are specified using an operation management object, which is an object with member functions designed to work with the particular paradigm being used. Operation management objects are also known as operation managers..

Creating a stream of objects in TPIE is very much like creating any other object in C++. The only difference is that data placed in the stream, whether explicitly, or as is more commonly the case, implicitly, is stored on disk. For example, to create a stream of integers, we could use either of the following:

```
AMI_STREAM<int> stream0;

AMI_STREAM<int> *pstream0 = new AMI_STREAM<int>;
```

`AMI_STREAM` is actually a macro defined to be the name of a particular implementation of streams at the AMI level, but for most users it is safe to assume that it is simply a class. This provides portability of applications across multiple AMI implementations.

The `AMI_STREAM` constructor does not actually put anything into the stream; it simply creates the necessary data structures to keep track of the contents of the stream when data is actually put into it. Data is typically put into streams using `AMI_scan()`, which is described in the next section.

6.3 Scanning

The simplest paradigm available in TPIE is scanning, as described in Section 2.4.1. Scanning can be used to produce streams, examine the contents of streams, or transform streams.

6.3.1 Basic Scanning

The most basic task a scan can perform is to write a series of objects to a stream. In the following example, we create a stream of integers consisting of the first 10000 natural numbers.

```
class scan_count : AMI_scan_object {
private:
    int maximum;
public:
    int ii;

    scan_count(int max = 1000) : maximum(max), ii(0) {};

    AMI_err initialize(void)
    {
        ii = 0;
        return AMI_ERROR_NO_ERROR;
    };

    AMI_err operate(int *out1, AMI_SCAN_FLAG *sf)
    {
        *out1 = ++ii;
        return (*sf = (ii <= maximum)) ? AMI_SCAN_CONTINUE :
            AMI_SCAN_DONE;
    };
};

scan_count sc(10000);
AMI_STREAM<int> amis0;

void f()
{
    AMI_scan(&sc, &amis0);
}
```

The class scan_count is a class of scan management object. It has two member functions, initialize() and operate(), which TPIE calls when asked to perform a scan. The first member function, initialize() is called at the beginning of the scan. TPIE expects that a call to this member function will cause the object to initialize any internal state it may maintain in preparation for performing a scan. The second member function, operate(), is called repeatedly during the scan to create objects to go into the output stream. operate() sets the flag *sf to indicate whether it generated output or not. Only when operate() returns either an error or AMI_SCAN_DONE does TPIE stop calling it.

The call to AMI_scan behaves as the following pseudo-code:

```
AMI_err AMI_scan(scan_count &sc, AMI_STREAM<int> *pamis)
{
    int ii;
    AMI_err ae;
    AMI_SCAN_FLAG sf;

    sc.initialize();
    while ((ae = sc.operate(&ii, &sf)) == AMI_SCAN_CONTINUE) {
        if (sf) {
            write ii to *pamis;
        }
    }

    if (ae != AMI_SCAN_DONE) {
        handle error conditions;
    }

    return AMI_ERROR_NO_ERROR;
}
```

Thus, after the function f() in the original example code is called, the stream amis0 contains the integers from 1 to 10000 in order.

Now that we have produced a stream, there are a variety of operations we can perform on it. One of the simplest things we can do with a stream of objects is scan it in order to transform it. As an example, suppose we wanted to square every integer in the stream amis0. We could do so using the following code:

```
class scan_square : AMI_scan_object {
public:
    AMI_err initialize(void)
    {
        return AMI_ERROR_NO_ERROR;
    };

    AMI_err operate(const int &in, AMI_SCAN_FLAG *sfin,
                    int *out, AMI_SCAN_FLAG *sfout)
    {
        if (*sfout = *sfin) {
            *out = in * in;
            return AMI_SCAN_CONTINUE;
        } else {
            return AMI_SCAN_DONE;
        }
    };
};

scan_square ss;
AMI_STREAM<int> amis1;

void g()
{
    AMI_scan(&amis0, &ss, &amis1);
}
```

Notice that the call to AMI_scan() in g() differs from the one we used in f() in that it takes two stream pointers and a scan management object. By convention, the stream amis0 is an input stream, because it appears before the scan management object ss in the argument list. By similar convention, amis1 is an output stream. Because the call to AMI_scan has one input stream and one output stream, TPIE expects the operate() member function of ss to have one input argument (which is called in in the example above) and one output argument (called out in the example above). Note that the operate() member function of the class square_scan also takes two pointers to flags, one for input (sfin) and one for output (sfout). *sfin is set by TPIE to indicate that there is more input to be processed. *sfout is set by the scan management object to indicate when output is generated. If a scan management object has no polymorph of operate() that takes the appropriate type number of arguments for the invocation of AMI_scan() that uses it, then a compile-time error is generated.

A call to AMI_scan with one input stream and one output stream behaves as the following pseudo-code:

```
AMI_err AMI_scan(AMI_STREAM<int> *instream, scan_square &ss,
                 AMI_STREAM<int> *outstream)
{
    int in, out;
    AMI_err ae;
    AMI_SCAN_FLAG sfin, sfout;

    sc.initialize();

    while (1) {
        {
             read in from *instream;
             sfin = (read succeeded);
        }
        if ((ae = ss.operate(in, &sfin, &out, &sf)) ==
            AMI_SCAN_CONTINUE) {
            if (sfout) {
                write out to *outstream;
            }
            if (ae == AMI_SCAN_DONE) {
                return AMI_ERROR_NO_ERROR;
            }
            if (ae != AMI_SCAN_CONTINUE) {
                handle error conditions;
            }
        }
    }
}
```

More complicated invocations of AMI_scan() can operate on up to four input streams and four output streams. Here is an example that takes two input streams of values, x and y, and produces four output streams, one consisting of the running sum of the x values, one consisting of the running sum of the y values, one consisting of the running sum of the squares of the x values, and the last consisting of the running sum of the squares of the y values.

```
class scan_sum : AMI_scan_object {
private:
    double sumx, sumx2, sumy, sumy2;
public:
    AMI_err initialize(void)
    {
        sumx = sumy = sumx2 = sumy2 = 0.0;
        return AMI_ERROR_NO_ERROR;
    };

    AMI_err operate(const double &x, const double &y,
                    AMI_SCAN_FLAG *sfin,
                    double *sx, double *sy,
                    double *sx2, double *sy2,
                    AMI_SCAN_FLAG *sfout)
    {
        if (sfout[0] = sfout[2] = sfin[0]) {
            *sx = (sumx += x);
            *sx2 = (sumx2 += x * x);
        }
        if (sfout[1] = sfout[3] = sfin[1]) {
            *sy = (sumx += y);
            *sy2 = (sumy2 += y * y);
        }
        return (sfin[0] || sfin[1]) ? AMI_SCAN_CONTINUE :
            AMI_SCAN_DONE;
    };
};

AMI_STREAM<double> xstream, ystream;

AMI_STREAM<double> sum_xstream, sum_ystream;
```

```
AMI_STREAM<double> sum_x2stream, sum_y2stream;

scan_sum ss;

void h()
{
    AMI_scan(&xstream, &ystream, &ss,
             &sum_xstream, &sum_ystream, &sum_x2stream, &sum_y2stream);
}
```

6.3.2 ASCII Input/Output

TPIE provides a number of predefined scan management objects. Among the most useful are instances of the template classes cxx_ostream_scan<T> and cxx_ostream_scan<T>, which are used for reading ASCII data into streams and writing the contents of streams in ASCII respectively. ASCII I/O is performed through the iostream facilities provided in the standard C++ library. Any class T for which the operators ostream &operator<<(ostream &s, T &t) and istream &operator>>(T &t) are defined can be used with this mechanism.

As an example, suppose we have a file called input_nums.txt containing one integer per line, such as

```
17
289
4195835
3145727
.
.
.
```

To read this file into a TPIE stream of integers, square each integer, and write them out to the file output_nums.txt we could use the following code:

```
void f()
```

```
{
    ifstream in_ascii("input_nums.txt");
    ofstream out_ascii("input_nums.txt");
    cxx_istream_scan<int> in_scan(in_ascii);
    cxx_ostream_scan<int> out_scan(out_ascii);
    AMI_STREAM<int> in_ami, out_ami;
    scan_square ss;

    // Read them.
    AMI_scan(&in_scan, &in_ami);

    // Square them.
    AMI_scan(&in_ami, &ss, &out_scan);

    // Write them.
    AMI_scan(&out_ami, out_scan);

}
```

In order to read from an input file using the scan object `in_scan`, `AMI_scan()` repeatedly calls `in_scan->operate()`, just as it would for any scan object. Each time `in_scan->operate()` is called, it uses the >> operator to read a single integer from the input file. When the input file is exhausted, `in_scan->operate()` returns `AMI_SCAN_DONE`, and `AMI_scan()` returns to its caller. The behavior of `out_scan` is similar to that of `in_scan`, except that it writes to an output file instead of reading from an input file.

6.3.3 Multi-Type Scanning

In all of the examples presented up to this point, scanning has been done on streams of objects that are all of the same type. `AMI_scan()` is not limited to such scans, however. In the following example, we have a scan management class that takes two streams of `double`s and returns a stream of complex numbers.

```
class complex {
public:
    complex(double real_part, imaginary_part);
    ...
};

class scan_build_complex : AMI_scan_object {
public:
    AMI_err initialize(void) {};
    AMI_err operate(const double &r, const double &i,
                    AMI_SCAN_FLAG *sfin,
                    complex *out, AMI_SCAN_FLAG *sfout)
    {
        if (*sfout = (sfin[0] || sfin[1])) {
            *out = complex((sfin[0] ? r : 0.0), (sfin[1] ? i : 0.0));
            return AMI_SCAN_CONTINUE;
        } else {
            return AMI_SCAN_DONE;
        }
    };
};
```

6.3.4 Out-of-Step Scanning

Up to this point, we have considered operate functions that always process all the inputs they are given. In this section, we introduce the concept of out-of-step scanning, which allows a scan management object to reject certain inputs and ask that they be resubmitted in subsequent calls to the `operate()` member function.

Suppose we have two streams of integers, each of which we know is sorted in ascending order. We would like to merge the two streams into a single output stream consisting of all the integers in the two input streams, in sorted order. In order to do this with a scan, we must have the ability to look at the next integer from each stream, choose the smaller of the two and write it to the output stream, and then ask for the next number from the stream from which it was taken. Luckily, there is a simple mechanism for performing this task. The same flags that TPIE uses to tell the scan management object which inputs are available can be used by the scan management object to indicate which inputs were used and which inputs should be presented again.

Consider the following example of a scan management object class which performs exactly the sort of binary merge described in the preceding paragraph:

```
class scan_binary_merge : AMI_scan_object {
public:
    AMI_err initialize(void) {};

    AMI_err operate(const int &in0, const int &in1,
                    AMI_SCAN_FLAG *sfin, int *out,
                    AMI_SCAN_FLAG *sfout)
    {
        if (sfin[0] && sfin[1]) {
            if (in0 < in1) {
                sfin[1] = false;
                *out = in0;
            } else {
                sfin[0] = 0;
                *out = in1;
            }
        } else if (!sfin[0]) {
            if (!sfin[1]) {
                *sfout = 0;
                return AMI_SCAN_DONE;
            } else {
                *out = in1;
            }
        } else {
            *out = in0;
        }
        *sfout = 1;
        return AMI_SCAN_CONTINUE;
    }
};
```

In the operate method, we first check that both inputs are valid by looking at the flags pointed to by sfin. If both are valid, then we select the smaller of the inputs and copy it to the output. We then clear the other input flag to let TPIE know that we did not use that input, but we will need it later and it should be resubmitted on the next call to operate. The remainder of the function handles the cases when one of more of the input streams is empty.

Note that in earlier scanning examples we did not write to the input flags at all. This indicated to AMI_scan that all inputs it presented to the scan management object were consumed by the invocation of the operate() member function and thus none needed to be presented again in subsequent calls.

6.4 Merging

The binary merging scan management class presented in the previous section could be used recursively to implement a merge sorting algorithm. We could simply divide the input stream into sub-streams small enough to fit into main memory, read each sub-stream into memory and sort it, and then merge pairs of streams, then pairs of merged pairs of streams, and so on, until we had merged all the input back into one completely sorted stream. While this approach would correctly sort the input, it would not be nearly as efficient as possible on most machines. The reason is that we typically have enough main memory available to merge many streams together at one time.

We can take advantage of the available main memory to reduce the number of passes required to merge a set of streams into a single stream, as described in Section 2.4.2. This will allow us to reduce the I/O complexity of merging by a factor of $O(\lg(M/B))$.

Unfortunately, taking advantage of all available main memory can be difficult, since we must explicitly keep track of the space needed for input blocks from each of the streams being merged, as well as the overhead of any data structures needed for the merge. Luckily, TPIE provides a mechanism that does most of the work for us. The function AMI_partition_and_merge() divides an input stream into sub-streams just small enough to fit into main memory, operates on each in main memory, then merges them back into a single output stream, using intermediate streams if memory constraints dictate. As was the case with AMI_scan(), the functional details of AMI_partition_and_merge() are specified via an operation management object, as shown in the following example:

```
class my_merger : AMI_merge_manager {
public:
    AMI_err initialize(arity_t arity, const T * const *in,
                       AMI_merge_flag *taken_flags,
                       int &taken_index);
    AMI_err operate(const T * const *in, AMI_merge_flag *taken_flags,
                    int &taken_index, T *out);
    AMI_err main_mem_operate(T* mm_stream, size_t len);
    size_t space_usage_overhead(void);
    size_t space_usage_per_stream(void);
};

AMI_STREAM<T> instream, outstream;

void f()
{
    my_merger mm;
    AMI_partition_and_merge(&instream, &outstream, &mm);
}
```

The class members are as follows:

`initialize()` Tells the object how many streams it should merge (`arity`) and what the first item from each stream is (`in`). `taken_flags` and `taken_index` provide two mechanisms for the merge manager to tell TPIE what objects it took from the input streams. These members are discussed in more detail in the context of a merge sorting example in Section 6.4.1.

`operate()` Just as in scanning, this member function is called repeatedly to process input objects.

`main_mem_operate()` Operates on an array of data in main memory when a sub-stream is small enough to fit entirely in main memory.

`space_usage_overhead()` TPIE calls this prior to initialization to asses how much main memory this object will use.

space_usage_per_item() TPIE calls this prior to initialization to asses how much main memory may be used per input stream. Merge management objects are allowed to use main memory space linear in the number of input streams.

The AMI access method AMI_partition_and_merge() behaves as indicated by the following pseudo-code. Note that for simplicity of presentation, boundary conditions are not covered.

```
AMI_err AMI_partition_and_merge(instream, outstream, mm)
{
    max_ss = max # of items that can fit in main memory;
    partition instream into num_substreams substreams of size max_ss;

    foreach substream[i] {
        read substream[i] into main memory;
        mm->main_mem_operate(substream[i]);
        write substream[i];
    }

    call mm->space_usage_overhead() and mm->space_usage_per_stream;

    compute merge_arity; // Maximum # of streams we can merge.

    while (num_substreams > 1) {
        for (i = 0; i < num_substreams; i += merge_arity) {
            merge substream[i] .. substream[i+merge_arity-1];
        }
        num_substreams /= merge_arity;
        max_ss *= merge_arity;
    }

    write single remaining substream to outstream;

    return AMI_ERROR_NO_ERROR;
}
```

6.4.1 Implementing Mergesort: An Extended Example

In order to examine some additional features of merge management objects, let us consider a more complete collection of code that implements and uses a merge management object to sorting integers. First, we declare the class:

```
class s_merge_manager : public AMI_merge_base<int> {
private:
    arity_t input_arity;
    pqueue *pq;
public:
    s_merge_manager(void);
    virtual ~s_merge_manager(void);
    AMI_err initialize(arity_t arity, const int * const *in,
                       AMI_merge_flag *taken_flags,
                       int &taken_index);
    AMI_err operate(const int * const *in, AMI_merge_flag *taken_flags,
                    int &taken_index, int *out);
    AMI_err main_mem_operate(int* mm_stream, size_t len);
    size_t space_usage_overhead(void);
    size_t space_usage_per_stream(void);
};
```

In addition to the standard class members for a merge management object, we have the following:

`input_arity` The number of input streams the merge management object must handle.

pq A priority queue into which items will be placed.

`s_merge_manger()` A constructor.

`s̃_merge_manger()` A destructor.

Construction and destruction are fairly straightforward. At construction time, we have no priority queue because we do not yet know how big the priority queue should be. pq will be set up when `initialize` is called. The destructor checks whether pq has been allocated, and deletes it if it has. The constructor and destructor are implemented as follows:

```
s_merge_manager::s_merge_manager(void)
{
    pq = NULL;
}

s_merge_manager::~s_merge_manager(void)
{
    if (pq != NULL) {
        delete pq;
    }
}
```

When AMI_merge() is called with a merge management object of type s_merge_manager, the first member functions called are space_usage_overhead() and space_usage_per_stream(). These functions return the number of bytes of main memory that the merge management object will allocate when initialized. space_usage_overhead()'s return value indicates that space will be needed for a priority queue. space_usage_per_stream()'s return value indicates that for each input stream, space (which is to be allocated when the priority queue is constructed) will be needed for an integer and an arity type.

```
size_t s_merge_manager::space_usage_overhead(void)
{
    return sizeof(pqueue<arity_t,int>);
}


size_t s_merge_manager::space_usage_per_stream(void)
{
    return sizeof(arity_t) + sizeof(int);
}
```

The next member function called by AMI_merge() is main_mem_operate(), which is called to handle the initial substreams that are small enough to fit in main memory. Since we are sorting, we will simply use quicksort.

```
AMI_err s_merge_manager::main_mem_operate(int* mm_stream, size_t len)
{
    qsort(mm_stream, len, sizeof(int), c_int_cmp);
    return AMI_ERROR_NO_ERROR;
}
```

Having sorted all of the initial substreams, `AMI_merge()` begins to merge them. Before merging a set of substreams, the merge management object's member function `initialize()` is called to inform the merge management object of the number of streams it should be prepared to merge. The object is also provided with the first object from each of the streams to be merged. For objects of the class `s_merge_manager`, the `initialize()` member function is as follows:

```
AMI_err s_merge_manager::initialize(arity_t arity,
                                    const int * const *in,
                                    AMI_merge_flag *taken_flags,
                                    int &taken_index)
{
    arity_t ii;

    input_arity = arity;

    if (pq != NULL) {
        delete pq;
    }

    // Construct a priority queue that can hold arity items.
    pq = new pqueue_heap_op(arity);

    for (ii = arity; ii--; ) {
        if (in[ii] != NULL) {
            taken_flags[ii] = 1;
            pq->insert(ii,*in[ii]);
        } else {
            taken_flags[ii] = 0;
        }
    }

    taken_index = -1;
    return AMI_MERGE_READ_MULTIPLE;
}
```

Note the use of the return value AMI_MERGE_READ_MULTIPLE. This indicates that the flags pointed to by *taken_flags are set to indicate which of the inputs were used and should not be presented again. This is very similar to the use of input flags to indicate which inputs were used by a scan management object as described in Section 6.3.4. The reason that we have a special return value to indicate when these flags are used is to increase performance. In order for AMI_scan() to determine which inputs were taken, it must examine all the flags. In a many-way merge, this could be quite time consuming. In the common case where only one item is taken, its index can be returned in taken_index in order to save the time that would be spent scanning the flags. This technique is used in the operate() member function, whose implementation is as follows:

```
AMI_err s_merge_manager::operate(CONST int * CONST *in,
                                 AMI_merge_flag *taken_flags,
                                 int &taken_index,
                                 int *out)
{
    // If the queue is empty, we are done.  There should be no more
    // inputs.
    if (!pq->num_elts()) {
        return AMI_MERGE_DONE;
    } else {
        arity_t min_source;
        int min_t;

        pq->extract_min(min_source,min_t);
        *out = min_t;
        if (in[min_source] != NULL) {
            pq->insert(min_source,*in[min_source]);
            taken_index = min_source;
        } else {
            taken_index = -1;
        }
        return AMI_MERGE_OUTPUT;
    }
}
```

6.5 Distribution

Distribution, as described in Section 2.4.2, is similar to merging, except that instead of combining a large number of input streams into a single output stream, the reverse is done, and the contents of a single input stream are distributed into many output streams. As was the case with merging, scan management objects and `AMI_scan()` can be used when distribution is done on a small scale. In circumstances where we wish to take advantage of all available main memory, however, `AMI_scan()` is inadequate.

Currently, TPIE supports distribution based on key values of the items begin distributed. This is done through the entry point `AMI_kb_dist()`. `AMI_kb_dist()` takes an input stream and a maximum and minimum key value. Inputs are distributed to as many output streams as can be buffered in main memory, based on a set of medians uniformly distributed over the range of keys. An additional output stream is written which contains the names of each of the streams that input was distributed to, so that these streams can be accessed again, either for further distribution or for additional processing.

6.6 Permutation

6.6.1 General Permutation

Permutation is a basic building block for many I/O algorithms. Routing a general permutation in the I/O model is asymptotically as complex as sorting, though for some important classes of permutations, such as BMMC permutations (See Section 6.6.2) faster algorithms are possible. In this section, we discuss `AMI_general_permute()`, which routes arbitrary permutations, but always takes as long as sorting, regardless of whether the particular permutation can be done more quickly or not.

General permutations are routed using the function `AMI_general_permute()`. Like other AMI functions, `AMI_general_permute()` relies on an operation management object to determine its precise behavior. Unlike functions covered up to now, however, the type of the operation management object need not depend on the type of object in the stream being permuted.

A general permutation management object must provide two member functions, `initialize()` and `destination()`. `initialize()` is called to inform the general permutation object of the length of the stream to be permuted. `destination()` is then called repeatedly to determine the destination for each object in the stream based on its initial position.

Here is an example of using general permutation to reverse the order of the items in a stream. The member function `initialize()` records the length of the stream in a private member `total\_size`. Subsequent calls to `destination()` subtract their input position from the total length to get the destination position.

```
class reverse_order : public AMI_gen_perm_object {
private:
    off_t total_size;
public:
    AMI_error initialize(off_t ts) {
        total_size = ts;
        return AMI_ERROR_NO_ERROR;
    };
    off_t destination(off_t source) {
        return total_size - 1 - source;
    };
};
```

```
AMI_STREAM<int> amis0, amis1;

void f()
{
    reverse_order ro;

    AMI_general_permute(&amis0, &amis1, (AMI_gen_perm_object *)&ro);
}
```

6.6.2 Bit Permutation

Bit permuting is a permutation technique in which the destination address of a given item is computed by manipulating the bits of its source address. The particular class of bit permutations that TPIE supports is the set of bit matrix multiply complement (BMMC) permutations. These permutations are defined on sets of objects whose size is a power of 2.

To illustrate, suppose we have an input consisting of $N = 2^n$ objects. A BMMC permutation on the input is defined by a nonsingular $n \times n$ bit matrix A and an n element column vector c of bits. Source and destination addresses are interpreted as column vectors of bits, with the low order bit of the address at the top. The destination address x' corresponding to a given source address x is computed as

$$x' = Ax + c$$

where addition and multiplication of matrix elements is done over GF(2). For a detailed description of BMMC permutations, see [CSW94].

Routing BMMC permutations in TPIE is done using the `AMI_BMMC_permute()` entry point, which takes an input stream, and output stream, and a pointer to a bit permutation management object. In the following example, we route a permutation that simply reverses the order of the source address bits to produce the destination address.

First, we construct the bit matrices we will use in the permutation.

```
bit_matrix A(n,n);
bit_matrix c(n,1);

{
    unsigned int ii,jj;

    for (ii = n; ii--; ) {
        c[ii][0] = 0;
        for (jj = n; jj--; ) {
            A[n-1-ii][jj] = (ii == jj);
        }
    }
}
```

Now we construct a permutation management object from the matrices and perform the permutation.

```
AMI_bit_perm_object bpo(A,c);

ae = AMI_BMMC_permute(&amis0, &amis1, (AMI_bit_perm_object *)&bpo);
```

This is all that is required to perform the permutation. Appropriate algorithms are chosen by the AMI, and the permutation is performed.

6.7 Sorting

6.7.1 Comparison Sorting

Sorting is a common primitive operation in many algorithms. Two examples of algorithms that rely on sorting are distribution sweeping, as described in Section 4.1.1, and PRAM simulation, as described in Section 4.2.1. Sorting is also often useful in and of itself.

I/O-efficient sorting can be done in a variety of ways, such as by merging (See Section 6.4), distribution (See Section 6.5), and Sharesort [AP94a], which combines elements of both along with simple bit permutations (See Section 6.6.2). It was shown in Section 3.3 that different algorithms are appropriate for different I/O system configurations. Because of this, TPIE provides a single function `AMI_sort()`, which selects an appropriate algorithm based on the underlying hardware characteristics.

`AMI_sort()` has two polymorphs. The first works on streams of objects for which the operator < is defined. It is invoked as follows:

```
AMI_STREAM<int> instream;
AMI_STREAM<int> outstream;

void f()
{
    AMI_sort(&instream, &outstream);
}
```

The second polymorph of `AMI_sort()` uses an explicit function to determine the relative order of two objects in the input stream. This is useful in cases where we may want to sort a stream of objects in several different ways. For example, the following code sorts a stream of complex numbers in two ways, by their real parts and by their imaginary parts.

```
class complex {
public:
    complex(double real_part, imaginary_part);
    double re(void);
    double im(void);
    ...
};

int compare_re(const complex &c1, const complex &c2)
{
    return (c1.re() < c2.re()) ? -1 :
            ((c1.re() > c2.re()) ? 1 : 0);
}

int compare_im(const complex &c1, const complex &c2)
{
    return (c1.im() < c2.im()) ? -1 :
            ((c1.im() > c2.im()) ? 1 : 0);
}

AMI_STREAM<complex> instream;
AMI_STREAM<complex> outstream_re;
AMI_STREAM<complex> outstream_im;

void f()
{
    AMI_sort(&instream, &outstream_re, compare_re);
    AMI_sort(&instream, &outstream_im, compare_im);
}
```

6.7.2 Key Bucket Sorting

TPIE also supports sorting based on the distribution of objects into buckets of ranges of keys. This can be done using the entry point `AMI_kb_sort()` whose syntax and functionality are similar to that of `AMI_sort()` as discussed above.

6.8 Matrix Operations

In addition to streams, which are linearly ordered collections of objects, the AMI provides a mechanism for storing large matrices in external memory. Matrices are a subclass of streams, and thus can be used with any of the stream operations discussed above. When a matrix is treated as a stream, its elements appear in row major order. In addition to stream operations, matrices support three simple arithmetic operations, addition, subtraction, and multiplication.

It is assumed that the class `T` of the elements in a matrix forms a quasiring with the operators + and *. Furthermore, the object `T((int)0)` is assumed to be an identity for +. At the moment, it is not assumed that the operator - in an inverse of +, and therefore no reduced complexity matrix multiplication algorithms analogous to Strassen's algorithm [Str69] are used.

TPIE provides support for two different classes of matrices: dense matrices, which are described in Section 6.8.1; and sparse matrices, which are described in Section 6.8.2.

6.8.1 Dense Matrix Operations

Dense matrices are implemented by the templated class `AMI_matrix`, which is a subclass of `AMI_STREAM`.

Dense matrices can be filled using `AMI_scan()`, though typically they are filled using the function `AMI_matrix_fill()`, which uses a scan management object that is given the row and column of each element of the matrix and asked to fill them in. In the following example, we create a 1000 by 1000 upper triangular matrix of ones and zeroes:

```
template<class T>
class fill_upper_tri : public AMI_matrix_filler<T> {
    AMI_err initialize(unsigned int rows, unsigned int cols)
    {
        return AMI_ERROR_NO_ERROR;
    };
    T element(unsigned int row, unsigned int col)
    {
        return (row <= col) ? T(1) : T(0);
    };
};

AMI_matrix<double> m(1000, 1000);

void f()
{
    fill_upper_tri<double> fut;

    AMI_matrix_fill(&m, (AMI_matrix_filler<T> *)&fut);
}
```

Arithmetic on dense matrices is performed in a straightforward way using the functions AMI_matrix_add(), AMI_matrix_subtract(), and AMI_matrix_multiply(), as in the following example:

```
AMI_matrix m0(1000, 500), m1(500, 2000), m2(1000, 2000);
AMI_matrix m3(1000, 500), m4(1000, 500);

void f()
{
    // Add m3 to m4 and put the result in m0.
    AMI_matrix_add(em3, em4, em0);

    // Multiply m0 by em1 to get m2.
    AMI_matrix_mult(em0, em1, em2);

    // Subtract m4 from m3 and put the result in m0.
    AMI_matrix_subtract(em3, em4, em0);
}
```

6.8.2 Sparse Matrices

External memory sparse matrices are implemented as a templated class AMI_sparse_matrix<T>. This class contains a stream of sparse matrix elements of the class AMI_sm_elem<T>, which is defined as follows:

```
template <class T>
class AMI_sm_elem {
public:
    unsigned int er;     // Element's row in the matrix.
    unsigned int ec;     // Element's column in the matrix.
    T val;               // Value in the element.
};
```

There is exactly one element in the stream for each non-zero element in a matrix. The order in which the non-zero elements appear in the stream can be arbitrary, although one might typically expect to see them in some canonical order, such as row-major order. In some cases, it is necessary for TPIE to permute the elements into some order that is dependent on the run-time environment, as in the sparse matrix by vector multiplication algorithm described in Section 5.1.

6.8.3 Element-wise Arithmetic

The functions `AMI_matrix_add()` and `AMI_matrix_subtract()`, defined in Section 6.8.1, perform elementwise arithmetic on matrices. At times, we might also wish to perform elementwise multiplication or division, or perform a scalar arithmetic operation on all elements of a matrix. TPIE provides mechanisms for performing these operations not only on matrices, but also on arbitrary streams, so long as they are of objects for which the appropriate arithmetic operators (e.g. +, -, *, /) are defined.

Elementwise arithmetic is done with scan management objects of the classes `AMI_scan_add`, `AMI_scan_sub`, `AMI_scan_mult` and `AMI_scan_div`, which are defined in the header file `AMI_stream_arith.H`. Here is an example that performs elementwise division on the elements of two streams.

```
#include <ami_stream_arith.H>

void foo()
{
    AMI_STREAM<int> amis0;
    AMI_STREAM<int> amis1;
    AMI_STREAM<int> amis2;

    // Divide each element of amis0 by the corresponding element of
    // amis1 and put the result in amis2.
    AMI_scan(&amis0, &amis1, &sd, &amis2);
}
```

6.8.4 Sparse Matrix by Vector Multiplication

TPIE performs multiplication of sparse matrices and vectors using the algorithm described in Section 5.1.

External-memory vectors are implemented as matrices with only a single column. For example

```
AMI_matrix<double> x(n, 1);
```

defines a vector with n elements.

We recall from Section 5.1 that the algorithm for multiplying sparse matrices by vectors has two parts. The first part pre-processes the sparse matrix into bands and the second part multiplies the banded sparse matrix by the vector. In TPIE, these two parts of the algorithm are decoupled, since it is often desirable to pre-process a matrix once and then multiply it by a large number of different vectors, as is done in the sparse matrix benchmarks in Section 8.3. Banded and non-banded matrices are of the same type as far as the compiler is concerned. The only difference is that the matrix elements they contained are either permuted into bands or in arbitrary order.

The following code fragment illustrates the steps one would go through to band a matrix and then multiply it by a vector.

```
AMI_matrix<double> p(n, 1), q(n, 1);

// n x n sparse matrices of doubles.  A_raw is the initial version,
// and A is the version that has been permuted into bands.

AMI_sparse_matrix<double> A(n,n), A_raw(n,n);

// Rows per band and total bands.
unsigned int rpb, tb;

void f()
{
    // Determine how many rows per band and how many total bands
    // will be created when the matrix is banded.

    AMI_sparse_band_info(A_raw, rpb, tb);

    // Band the matrix.

    AMI_sparse_bandify(A_raw, A, rpb);

    // Do the multiplication.

    AMI_sparse_mult_scan_banded(A, p, q, n, n, rpb);
}
```

Note that once we have permuted A into its banded form A_raw, we can perform arbitrary numbers of banded multiplications on it by calling AMI_sparse_mult_scan_banded() many times.

6.9 Conclusions

TPIE uses a stream abstraction to support a large number of I/O efficient operations. These include many variations of scanning, sorting, permuting, bit permuting, dense and sparse matrix operations. These primitives can be composed to implement many of the algorithms discussed in Chapters 4 and 5.

In the next chapter, we will see additional examples of how TPIE can be used to solve problems in an I/O-efficient manner.

Chapter 7

Additional Examples of TPIE Implementations

In the previous chapter, we discussed the basic structure of TPIE and how it can be used to solve a number of fundamental problems in an I/O-efficient manner. This chapter contains some additional annotated examples of TPIE application code for more advanced problems. The two applications we consider are convex hulls and list ranking.

7.1 Convex Hull

The convex hull of a set of points in the plane is the smallest convex polygon which encloses all of the points. Graham's scan is a simple algorithm for computing convex hulls. It is discussed in most introductory books on computational geometry, such as [PS85]. Although Graham's scan was not originally designed for external memory, it can be implemented optimally in this setting. What is interesting about this implementation is that external memory stacks are used within the implementation of a scan management object.

First, we need a data type for storing points. We use the following simple class, which is templated to handle any numeric type.

```
template<class T>
class point {
public:
    T x;
    T y;
    point() {};
    point(const T &rx, const T &ry) : x(rx), y(ry) {};
    ~point() {};

    inline int operator==(const point<T> &rhs) const {
        return (x == rhs.x) && (y == rhs.y);
    }
    inline int operator!=(const point<T> &rhs) const {
        return (x != rhs.x) || (y != rhs.y);
    }

    // Comparison is done by x.
    int operator<(const point<T> &rhs) const {
        return (x < rhs.x);
    }

    int operator>(const point<T> &rhs) const {
        return (x > rhs.x);
    }

    friend ostream& operator<<(ostream& s, const point<T> &p);
    friend istream& operator>>(istream& s, point<T> &p);
};
```

Once the points are sorted by their x values, we simply scan them to produce the upper and lower hulls, each of which are stored as a stack pointed to by the scan management object. We then concatenate the stacks to produce the final hull. The stacks are implemented using the AMI_stack class, which is a class derived from AMI_STREAM. This stack class supports all the normal operations that streams support, and also has push() and pop() member functions that add an object to the end of the stream and delete the last object in the stream respectively.

The code for computing the convex hull of a set of points using Graham's scan with external memory data and external memory stacks is as follows.

```
template<class T>
AMI_err convex_hull(AMI_STREAM< point<T> > *instream,
                     AMI_STREAM< point<T> > *outstream)
{
    AMI_err ae;

    point<T> *pt;

    AMI_stack< point<T> > uh((unsigned int)0, instream->stream_len());
    AMI_stack< point<T> > lh((unsigned int)0, instream->stream_len());

    AMI_STREAM< point<T> > in_sort;

    // Sort the points by x.

    ae = AMI_sort(instream, &in_sort);

    // Compute the upper hull and lower hull in a single scan.

    scan_ul_hull<T> sulh;

    sulh.uh_stack = &uh;
    sulh.lh_stack = &lh;

    ae = AMI_scan(&in_sort, &sulh);

    // Copy the upper hull to the output.

    uh.seek(0);

    while (1) {
        ae = uh.read_item(&pt);
        if (ae == AMI_ERROR_END_OF_STREAM) {
            break;
        } else if (ae != AMI_ERROR_NO_ERROR) {
            return ae;
        }

        ae = outstream->write_item(*pt);
        if (ae != AMI_ERROR_NO_ERROR) {
            return ae;
        }
    }

    // Reverse the lower hull, concatenating it onto the upper hull.

    while (lh.pop(&pt) == AMI_ERROR_NO_ERROR) {
        ae = outstream->write_item(*pt);
        if (ae != AMI_ERROR_NO_ERROR) {
            return ae;
        }
    }

    return AMI_ERROR_NO_ERROR;
}
```

The only task that remains is to define a scan management object that is capable of producing the upper and lower hulls by scanning the points. According to Graham's scan algorithm, we produce the upper hull by moving forward in the x direction, adding each point we encounter to the upper hull, until we find a point that induces a concave turn on the surface of the hull. We then move backwards through the list of points that have been added to the hull, eliminating points until a convex path is reestablished. This process is made efficient by storing the points on the hull so far in a stack. The code for the scan management object, which relies on the function ccw() to actually determine whether a corner is convex or not, is as follows:

```
template<class T>
class scan_ul_hull : AMI_scan_object {
public:
    AMI_stack< point <T> > *uh_stack, *lh_stack;

    scan_ul_hull(void);
    virtual ~scan_ul_hull(void);
    AMI_err initialize(void);
    AMI_err operate(const point<T> &in, AMI_SCAN_FLAG *sfin);
};
```

```
template<class T>
scan_ul_hull<T>::scan_ul_hull(void) : uh_stack(NULL), lh_stack(NULL)
{
}

template<class T>
scan_ul_hull<T>::~scan_ul_hull(void)
{
}

template<class T>
AMI_err scan_ul_hull<T>::initialize(void)
{
    return AMI_ERROR_NO_ERROR;
}


template<class T>
AMI_err scan_ul_hull<T>::operate(const point<T> &in,
                                 AMI_SCAN_FLAG *sfin)
{
    AMI_err ae;

    // If there is no more input we are done.
    if (!*sfin) {
        return AMI_SCAN_DONE;
    }

    if (!uh_stack->stream_len()) {

        // If there is nothing on the stacks then put the first point
        // on them.
        ae = uh_stack->push(in);
        if (ae != AMI_ERROR_NO_ERROR) {
            return ae;
        }

        ae = lh_stack->push(in);
        if (ae != AMI_ERROR_NO_ERROR) {
            return ae;
        }

    } else {
```

```
// Add to the upper hull.

{
    // Pop the last two points off.

    point<T> *p1, *p2;

    tp_assert(uh_stack->stream_len() >= 1, "Stack is empty.");

    uh_stack->pop(&p2);

    // If the point just popped is equal to the input, then we
    // are done.  There is no need to have both on the stack.

    if (*p2 == in) {
        uh_stack->push(*p2);
        return AMI_SCAN_CONTINUE;
    }

    if (uh_stack->stream_len() >= 1) {
        uh_stack->pop(&p1);
    } else {
        p1 = p2;
    }

    // While the turn is counter clockwise and the stack is
    // not empty pop another point.

    while (1) {
        if (ccw(*p1,*p2,in) >= 0) {
            // It does not turn the right way.  The points may
            // be colinear.
            if (uh_stack->stream_len() >= 1) {
                // Move backwards to check another point.
                p2 = p1;
                uh_stack->pop(&p1);
            } else {
                // Nothing left to pop, so we can't move
                // backwards.  We're done.
                uh_stack->push(*p1);
                if (in != *p1) {
                    uh_stack->push(in);
                }
                break;
            }
        } else {
            // It turns the right way.  We're done.
            uh_stack->push(*p1);
            uh_stack->push(*p2);
            uh_stack->push(in);
            break;
        }
    }
}

// Add to the lower hull.

{
    // Pop the last two points off.

    point<T> *p1, *p2;

    tp_assert(lh_stack->stream_len() >= 1, "Stack is empty.");

    lh_stack->pop(&p2);

    // If the point just popped is equal to the input, then we
    // are done.  There is no need to have both on the stack.

    if (*p2 == in) {
        lh_stack->push(*p2);
        return AMI_SCAN_CONTINUE;
    }

    if (lh_stack->stream_len() >= 1) {
        lh_stack->pop(&p1);
    } else {
        p1 = p2;
    }

    // While the turn is clockwise and the stack is
    // not empty pop another point.

    while (1) {
        if (ccw(*p1,*p2,in) <= 0) {
            // It does not turn the right way.  The points may
            // be colinear.
```

```
                    if (lh_stack->stream_len() >= 1) {
                        // Move backwards to check another point.
                        p2 = p1;
                        lh_stack->pop(&p1);
                    } else {
                        // Nothing left to pop, so we can't move
                        // backwards.  We're done.
                        lh_stack->push(*p1);
                        if (in != *p1) {
                            lh_stack->push(in);
                        }
                        break;
                    }
                } else {
                    // It turns the right way.  We're done.
                    lh_stack->push(*p1);
                    lh_stack->push(*p2);
                    lh_stack->push(in);
                    break;
                }
            }
        }
    }

    return AMI_SCAN_CONTINUE;
}
```

By encapsulating the external memory stacks in the scan management object that performs the scan, we are able to run the entire algorithm in an I/O-efficient manner.

The only thing left to be written is the function ccw(), which computes twice the signed area of a triangle in the plane by evaluating a 3 by 3 determinant. The result is positive if and only if the the three points in order form a counterclockwise cycle. The interested reader is directed to [PS85] for more details on the role of ccw() in geometric computations.

```
template<class T>
T ccw(const point<T> &p1, const point<T> &p2, const point<T> &p3)
{
    T sa;

    sa = ((p1.x * p2.y - p2.x * p1.y) -
          (p1.x * p3.y - p3.x * p1.y) +
          (p2.x * p3.y - p3.x * p2.y));

    return sa;
}
```

This completes the implementation of Graham's scan.

7.2 List-Ranking

List ranking, which was discussed in Section 4.3, is a fundamental problem in graph theory. The problem is as follows: We are given the directed edges of a linked list in some arbitrary order. Each edge is an ordered pair of node identifiers. The first is the source of the edge and the second is the destination of the edge. Our goal is to assign a weight to each edge corresponding to the number of edges that would have to be traversed to get from the head of the list to that edge.

The code given below solves the list ranking problem using the randomized algorithm presented in Section 4.3. As was the case in the code examples in Chapter 6, `#include` statements for header files and definitions of some classes and functions as well as some error and consistency checking code are left out so that the reader can concentrate on the more important details of how TPIE is used. A completely ready to compile version of this code is included in the TPIE source distribution.

To begin with, we need a class to represent edges. Because the algorithm will set a flag for each edge and then assign weights to the edges, we include fields for these values.

```
class edge {
public:
    unsigned long int from;        // Node it is from
    unsigned long int to;          // Node it is to
    unsigned long int weight;      // Position when ranked.
    bool flag;            // A flag used to randomly select some edges.

    friend ostream& operator<<(ostream& s, const edge &e);
};
```

As the algorithm runs, it will sort the edges. At times this will be done by their sources and at times by their destinations. The following simple functions are used to compare these values:

```
int edgefromcmp(const edge &s, const edge &t)
{
    return (s.from < t.from) ? -1 : ((s.from > t.from) ? 1 : 0);
}

int edgetocmp(const edge &s, const edge &t)
{
    return (s.to < t.to) ? -1 : ((s.to > t.to) ? 1 : 0);
}
```

The first step of the algorithm is to assign a randomly chosen flag, whose value is 0 or 1 with equal probability, to each edge. This is done using `AMI_scan()` with a scan management object of the class `random_flag_scan`, which is defined as follows:

```
class random_flag_scan : AMI_scan_object {
public:
    AMI_err initialize(void);
    AMI_err operate(const edge &in, AMI_SCAN_FLAG *sfin,
                    edge *out, AMI_SCAN_FLAG *sfout);
};

AMI_err random_flag_scan::initialize(void) {
    return AMI_ERROR_NO_ERROR;
}

AMI_err random_flag_scan::operate(const edge &in, AMI_SCAN_FLAG *sfin,
                                         edge *out, AMI_SCAN_FLAG *sfout)
{
    if (!(sfout[0] = *sfin)) {
        return AMI_SCAN_DONE;
    }
    *out = in;
    out->flag = (random() & 1);

    return AMI_SCAN_CONTINUE;
}
```

The next step of the algorithm is to separate the edges into an active list and a cancel list. In order to do this, we sort one copy of the edges by their sources (using edgefromcmp) and sort another copy by their destinations (using edgetocmp). We then call AMI_scan() to scan the two lists and produce an active list and a cancel list. A scan management object of class separate_active_from_cancel, which is defined below, is used.

```
//////////////////////////////////////////////////////////////////////
// separate_active_from_cancel
//
// A class of scan object that takes two edges, one to a node and one
// from it, and writes an active edge and possibly a canceled edge.
//
// Let e1 = (x,y,w1,f1) be the first edge and e2 = (y,z,w2,f2) the
// second.  If e1's flag (f1) is set and e2's (f2) is not, then we
// write (x,z,w1+w2,?) to the active list and e2 to the cancel list.
// The effect of this is to bridge over the node y with the new active
// edge.  f2, which was the second half of the bridge, is saved in the
// cancelation list so that it can be ranked later after the active
// list is recursively ranked.
//
// Since all the flags should have been set randomly before this
// function is called, the expected size of the active list is 3/4 the
// size of the original list.
//////////////////////////////////////////////////////////////////////
class separate_active_from_cancel : AMI_scan_object {
public:
    AMI_err initialize(void);
    AMI_err operate(const edge &e1, const edge &e2,
                    AMI_SCAN_FLAG *sfin, edge *active, edge *cancel,
                    AMI_SCAN_FLAG *sfout);
};

AMI_err separate_active_from_cancel::initialize(void)
{
    return AMI_ERROR_NO_ERROR;
}

// e1 is from the list of edges sorted by where they are from.
// e2 is from the list of edges sorted by where they are to.
AMI_err separate_active_from_cancel::operate(const edge &e1,
                                             const edge &e2,
                                             AMI_SCAN_FLAG *sfin,
                                             edge *active,
                                             edge *cancel,
                                             AMI_SCAN_FLAG *sfout)
{
    // If we have both inputs.
    if (sfin[0] && sfin[1]) {
        // If they have a node in common we may be in a bridging
        // situation.
        if (e2.to == e1.from) {
            // We will write to the active list no matter what.
            sfout[0] = 1;
            *active = e2;
            if (sfout[1] = (e2.flag && !e1.flag)) {
                // Bridge.  Put e1 on the cancel list and add its
                // weight to the active output.
                active->to = e1.to;
                active->weight += e1.weight;
                *cancel = e1;
                sfout[1] = 1;
            } else {
                // No bridge.
                sfout[1] = 0;
            }
        } else {
            // They don't have a node in common, so one of them needs
            // to catch up with the other.  What happened is that
            // either e2 is the very last edge in the list or e1 is
            // the very first or we just missed a bridge because of
            // flags.
```

```
                sfout[1] = 0;
                if (e2.to > e1.from) {
                    // e1 is behind, so just skip it.
                    sfin[1] = 0;
                    sfout[0] = 0;
                } else {
                    // e2 is behind, so put it on the active list.
                    sfin[0] = 0;
                    sfout[0] = 1;
                    *active = e2;
                }
            }
            return AMI_SCAN_CONTINUE;
        } else {
            // If we only have one input, either just leave it active.
            if (sfin[0]) {
                *active = e1;
                sfout[0] = 1;
                sfout[1] = 0;
                return AMI_SCAN_CONTINUE;
            } else if (sfin[1]) {
                *active = e2;
                sfout[0] = 1;
                sfout[1] = 0;
                return AMI_SCAN_CONTINUE;
            } else {
                // We have no inputs, so we're done.
                sfout[0] = sfout[1] = 0;
                return AMI_SCAN_DONE;
            }
        }
}
```

The next step of the algorithm is to strip the cancelled edges away from the list of all edges. The remaining active edges will form a recursive subproblem. Again, we use a scan management object, this time of the class `strip_active_from_cancel`, which is defined as follows:

```
////////////////////////////////////////////////////////////////////////
//
```

```
// strip_cancel_from_active
//
// A scan management object to take an active list and remove the
// smaller weighted edge of each pair of consecutive edges with the
// same destination.  The purpose of this is to strip edges out of the
// active list that were sent to the cancel list.
//
//////////////////////////////////////////////////////////////////////
class strip_cancel_from_active : AMI_scan_object {
private:
    bool holding;
    edge hold;
public:
    AMI_err initialize(void);
    AMI_err operate(const edge &active, AMI_SCAN_FLAG *sfin,
                    edge *out, AMI_SCAN_FLAG *sfout);
};

AMI_err strip_cancel_from_active::initialize(void) {
    holding = false;
    return AMI_ERROR_NO_ERROR;
}

// Edges should be sorted by destination before being processed by
// this object.
AMI_err strip_cancel_from_active::operate(const edge &active,
                                          AMI_SCAN_FLAG *sfin,
                                          edge *out, AMI_SCAN_FLAG *sfout)
{
    // If no input then we're done, except that we might still be
    // holding one.
    if (!*sfin) {
        if (holding) {
            *sfout = 1;
            *out = hold;
            holding = false;
            return AMI_SCAN_CONTINUE;
        } else {
            *sfout = 0;
            return AMI_SCAN_DONE;
        }
    }

    if (!holding) {
            // If we are not holding anything, then just hold the current
            // input.
            hold = active;
            holding = true;
            *sfout = 0;
        } else {
            *sfout = 1;

            if (active.to == hold.to) {
                if (active.weight > hold.weight) {
                    *out = active;
                } else {
                    *out = hold;
                }

                holding = false;
            } else {
                *out = hold;
                hold = active;
            }
        }

        return AMI_SCAN_CONTINUE;
}
```

After recursion, we must patch the cancelled edges back into the recursively ranked list of active edges. This patching is done using a scan with a scan management object of the class `interleave_active_cancel`, which is implemented as follows:

```
//////////////////////////////////////////////////////////////////////
// interleave_active_cancel
//
// This is a class of merge object that merges two lists of edges
// based on their to fields.  The first list of edges should be active
// edges, while the second should be cancelled edges.  When we see two
// edges with the same to field, we know that the second was cancelled
// when the first was made active.  We then fix up the weights and
// output the two of them, one in the current call and one in the next
// call.
//
// The streams this operates on should be sorted by their terminal
// (to) nodes before AMI_scan() is called.
//
////////////////////////////////////////////////////////////////////////

class patch_active_cancel : AMI_scan_object {
private:
    bool holding;
    edge hold;
public:
    AMI_err initialize(void);
    AMI_err operate(const edge &active, const edge &cancel,
                    AMI_SCAN_FLAG *sfin,
                    edge *patch, AMI_SCAN_FLAG *sfout);
};

AMI_err patch_active_cancel::initialize(void)
{
    holding = false;
    return AMI_ERROR_NO_ERROR;
}

AMI_err patch_active_cancel::operate(const edge &active,
                                     const edge &cancel,
                                     AMI_SCAN_FLAG *sfin,
                                     edge *patch,
                                     AMI_SCAN_FLAG *sfout)
{
    // Handle the special cases that occur when holding an edge and/or
    // completely out of input.
    if (holding) {
        sfin[0] = sfin[1] = 0;
        *patch = hold;
        holding = false;
        *sfout = 1;
        return AMI_SCAN_CONTINUE;
    } else if (!sfin[0]) {
        *sfout = 0;
        return AMI_SCAN_DONE;
    }

    if (!sfin[1]) {
        // If there is no cancel edge (i.e. all have been processed)
        // then just pass the active edge through.
        *patch = active;
    } else {
        if (holding = (active.to == cancel.to)) {
            patch->from = active.from;
            patch->to = cancel.from;
            patch->weight = active.weight - cancel.weight;
            hold.from = cancel.from;
            hold.to = active.to;
            hold.weight = active.weight;
        } else {
            *patch = active;
            sfin[1] = 0;
        }
    }

    *sfout = 1;
    return AMI_SCAN_CONTINUE;

}
```

Finally, here is the actual function to rank the list.

```
//////////////////////////////////////////////////////////////////////
// list_rank()
//
// This is the actual recursive function that gets the job done.
// We assume that all weights are 1 when the initial call is made to
// this function.
//
// Returns 0 on success, nonzero otherwise.
//////////////////////////////////////////////////////////////////////

int list_rank(AMI_STREAM<edge> *istream, AMI_STREAM<edge> *ostream)
{
    AMI_err ae;

    off_t stream_len = istream->stream_len();

    AMI_STREAM<edge> *edges_rand;
    AMI_STREAM<edge> *active;
    AMI_STREAM<edge> *active_2;
    AMI_STREAM<edge> *cancel;
    AMI_STREAM<edge> *ranked_active;
```

```
 AMI_STREAM<edge> *edges_from_s;
 AMI_STREAM<edge> *cancel_s;
 AMI_STREAM<edge> *active_s;
 AMI_STREAM<edge> *ranked_active_s;

 // Scan/merge management objects.
 random_flag_scan my_random_flag_scan;
 separate_active_from_cancel my_separate_active_from_cancel;
 strip_cancel_from_active my_strip_cancel_from_active;
 patch_active_cancel my_patch_active_cancel;

 // Check if the recursion has bottomed out.  If so, then read in
 // the array and rank it.

 {
     size_t mm_avail;

     MM_manager.available(&mm_avail);

     if (stream_len * sizeof(edge) < mm_avail / 2) {
         edge *mm_buf = new edge[stream_len];
         istream->seek(0);
         istream->read_array(mm_buf,&stream_len);
         main_mem_list_rank(mm_buf,stream_len);
         ostream->write_array(mm_buf,stream_len);
         return 0;
     }
 }

 // Flip coins for each node, setting the flag to 0 or 1 with equal
 // probability.

 edges_rand = new AMI_STREAM<edge>;

 AMI_scan(istream, &my_random_flag_scan, edges_rand);

 // Sort one stream by source.  The original input was sorted by
 // destination, so we don't need to sort it again.

 edges_from_s = new AMI_STREAM<edge>;

 ae = AMI_sort(edges_rand, edges_from_s, edgefromcmp);
```

```
// Scan to produce and active list and a cancel list.

active = new AMI_STREAM<edge>;
cancel = new AMI_STREAM<edge>;

ae = AMI_scan(edges_from_s, edges_rand,
              &my_separate_active_from_cancel,
              active, cancel);

delete edges_from_s;
delete edges_rand;

// Strip the edges that went to the cancel list out of the
// active list.

active_s = new AMI_STREAM<edge>;

ae = AMI_sort(active, active_s, edgetocmp);

delete active;

active_2 = new AMI_STREAM<edge>;

ae = AMI_scan(active_s,
              &my_strip_cancel_from_active,
              active_2);

delete active_s;

// Recurse on the active list.  The list we pass in is sorted by
// destination.  The recursion will return a list sorted by
// source.

ranked_active = new AMI_STREAM<edge>;

list_rank(active_2, ranked_active);

delete active_2;

cancel_s = new AMI_STREAM<edge>;

AMI_sort(cancel, cancel_s, edgetocmp);

delete cancel;
```

```
    // The output of the recursive call is not necessarily sorted by
    // destination.  We'll make it so before we try to merge in the
    // cancel list.

    ranked_active_s = new AMI_STREAM<edge>;

    AMI_sort(ranked_active, ranked_active_s, edgetocmp);

    delete ranked_active;

    // Now merge the recursively ranked active list and the sorted
    // cancel list.

    ae = AMI_scan(ranked_active_s, cancel_s,
                  &my_patch_active_cancel, ostream);

    delete ranked_active_s;
    delete cancel_s;

    return 0;
}
```

Our recursion bottoms out when the problem is small enough to fit entirely in main memory, in which case we read it in and call a function to rank a list in main memory. The details of this function are omitted here.

```
////////////////////////////////////////////////////////////////////////
// main_mem_list_rank()
//
// This function ranks a list that can fit in main memory.  It is used
// when the recursion bottoms out.
//
////////////////////////////////////////////////////////////////////////

int main_mem_list_rank(edge *edges, size_t count)
{
    // Rank the list in main memory

    ...
    return 0;
}
```

7.3 Conclusions

The two examples given in this chapter illustrate how TPIE can be used to solve real application problems. Using TPIE's framework-oriented approach to I/O, these algorithms were implemented without having to resort to any low-level I/O programming.

Convex hulls and list-ranking are not the only algorithms we have implemented in TPIE. A number of other algorithms are discussed in the next chapter, although we will focus more on the performance of TPIE based implementations of these algorithms than on the details of the implementations themselves.

Chapter 8

TPIE Prototype Performance

In this chapter we present a number of performance results illustrating the performance of TPIE implementations of a number of algorithms discussed in this thesis. Some of these algorithms were implemented in the context of NAS parallel benchmarks [BBB$^+$94], while others were implemented as a part of our own benchmark programs. Our results indicate that TPIE solutions to these benchmark problems are extremely efficient, both in terms of the amount of I/O they perform and the amount of real time they consume.

8.1 Scanning – The NAS EP Benchmark

Because scanning is such a generic operation, we could have chosen any of a wide variety of problems as a benchmark. We chose the NAS benchmark NAS EP [BBB$^+$94] for two reasons: it was designed to model computations that actually occur in large-scale scientific computation; and it can be used to illustrate an important class of scan optimizations called scan combinations.

The NAS EP benchmark generates a sequence of independent pairs of Gaussian deviates. It first generates a sequence of $2N$ independent uniformly distributed deviates using the liner congruential method [Knu81]. Then, it uses the polar method [Knu81] to

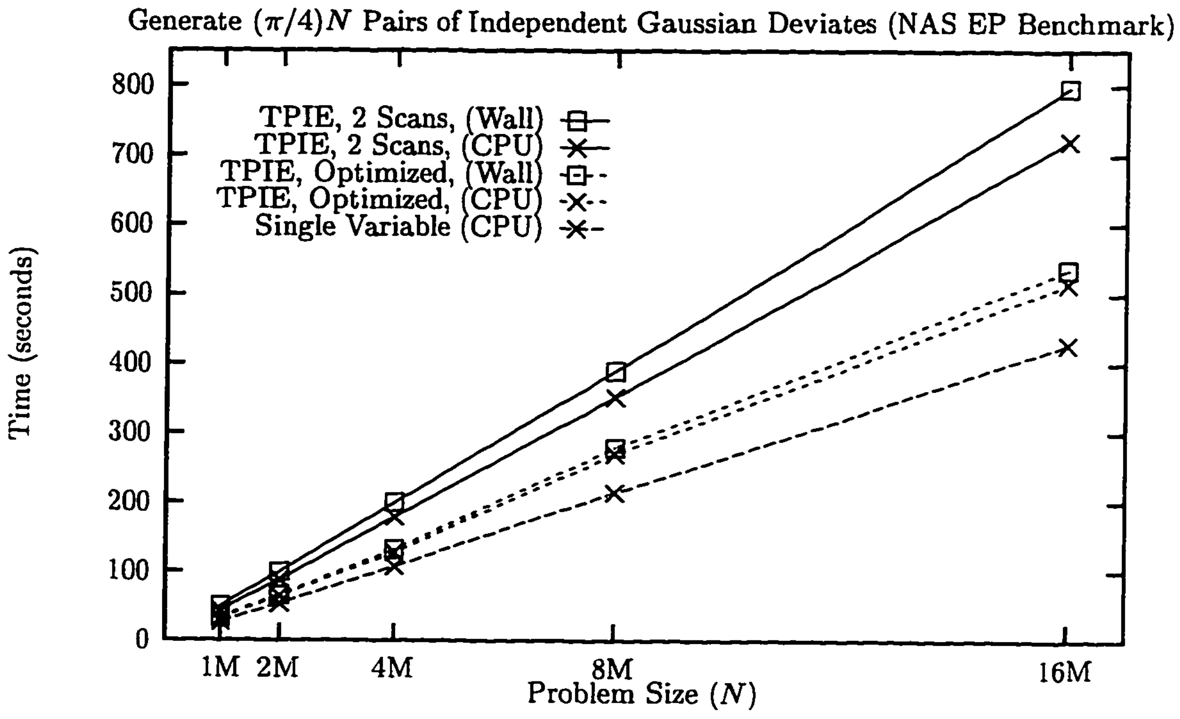

Figure 8.1: NAS EP Benchmark

generate approximately $(\pi/4)N$ pairs of Gaussian deviates from the original sequence of uniform deviates.

Performance of our TPIE implementation of NAS EP is shown in Figure 8.1. There are three sets of curves, labeled "TPIE, 2 Scans," "TPIE, Optimized," and "Single Variable."

The difference between the 2 scan TPIE curves and the optimized TPIE curves lies in the number of scans performed. In the the former, two separate scans are performed, one to write the uniformly distributed random variates and the other to read the uniformly distributed random variates and write the Gaussian pairs. In the latter, the two steps are combined into a single scan. As expected, the optimized code outperforms the unoptimized code.

This significance of this difference lies not so much in the fact that it tells programmers they should combine scans, as in the fact that scan combination is a relatively straightforward optimization that can be automated by a preprocessor. Such a preprocessor would parse the C++ text of a program and, where possible, construct hybrid scan management objects. The scans would then be replaced by a single scan using the hybrid object. Additionally, scans can often be piggy-backed on many other types of operations, such as merges, distributions, sorts, and permutations.

Returning to Figure 8.1, the single variable curve plots the CPU performance of a C++ program that does not perform any I/O at all, using TPIE or any other system. Instead, each pair of random variates is simply written over the previously generated pair in main memory. The purpose of this curve is to illustrate a fundamental lower bound on the CPU complexity of generating the variates. By comparing this lower bound to the CPU curves of the TPIE implementations, we can see that the CPU overhead associated with scheduling and performing I/O, communicating between the components of TPIE, and interacting with the user supplied scan management object is quite small. In the optimized case it amounts to approximately 20%.

8.2 Sorting

The NAS IS benchmark is designed to model key ranking [BBB+94]. We are given an array of integer keys $K_0, K_1, \dots K_{N-1}$ chosen from a key universe $[0, U)$, where $U \ll N$. Our goal is to produce, for each i, the rank $R(K_i)$, which is the position K_i would appear in if the keys were sorted. The benchmark does not technically require that the keys be sorted at any time, only that their ranks be computed. As an additional caveat, each key is the average of four random variates chosen independently from a uniform probability distribution over $[0, U)$. The distribution is thus approximately normal. Ten iterations of ranking are to be performed, and at the beginning of each iteration an extra key is added in each distant tail of the distribution.

In order to rank the keys, we sort them, scan the sorted list to assign ranks, and then re-sort based on the original indices of the keys. In the first sort, we do not have a uniform distribution of keys, but we do have a distribution whose probabilistic structure is known. Given any probabilistic distribution of keys with cumulative distribution function (c.d.f.) F_K, we can replace each key value k_i by $k_i' = F_K(k_i)$ in order to get keys that appear as if chosen at random from a uniform distribution on $[0, 1]$. Because the keys of the NAS IS benchmark are sums of four independent uniformly distributed random variates, their c.d.f. is a relatively easy to compute piecewise fourth degree polynomial.

For the sake of comparison, we implemented this first sort in four ways, using both merge sort and three variations of distribution sorting. One distribution sort, called CDF1, assumed that the keys were uniformly distributed. The next CDF4, used the fourth degree c.d.f. mentioned above to make the keys more uniform. Finally, as a compromise, CDF2 used a quadratic approximation to the 4th degree c.d.f. based on the c.d.f. of the sum of two independent uniform random variables.

In the second sort, the indices are the integers in the range $[0, N)$, so we used a distribution sort in all cases. The rationale behind this was that distribution and merging should use the same amount of I/O in this case, but distribution should require less CPU time because it has no need for the main-memory priority queue that merge sorting requires.

The performance of the the various approaches is shown in Figure 8.2. As we expected, merge sort used more CPU time than any of the distribution sorts and the more complicated the c.d.f. we computed the more CPU time we used. When total time is considered, merge sort came out ahead of the distribution sorts. This appears to be the result of imperfect balance when the keys are distributed, which causes an extra level of recursion for a portion of the data. Interestingly, the quality of our c.d.f. approximation had little effect on the time spent doing I/O. We conjecture that this would not be the case with more skewed distributions, such as exponential distributions. The jump in the total time for the merge sort that occurs between 8M and 10M is due

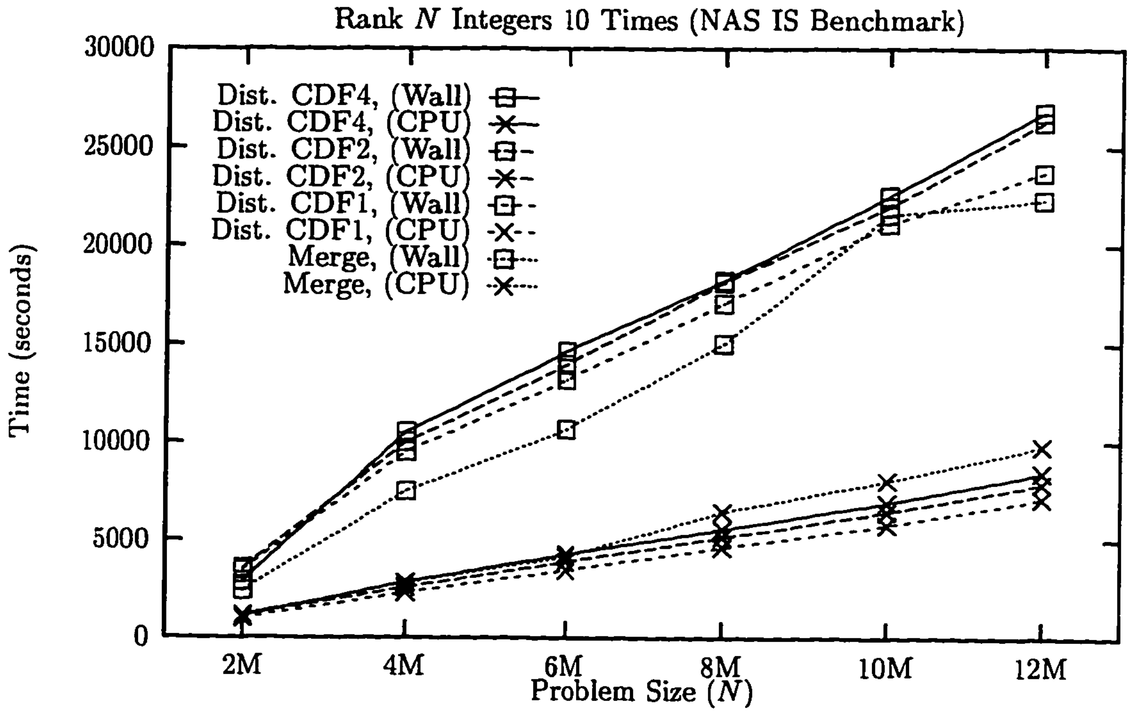

Figure 8.2: NAS IS benchmark performance

to a step being taken in the logarithmic term in that range.

8.3 Sparse Matrix Benchmarks

In order to test the performance of TPIE sparse matrices, we implemented two benchmarks. The first is the NAS CG benchmark [BBB$^+$94], which solves an unstructured sparse linear system by the conjugate gradient method. The second, which we call SMOOTH, which models a finite element computation on a 3-D mesh.

8.3.1 The NAS CG Benchmark.

NAS CG, which is fully specified in [BBB$^+$94], uses the inverse power method to find an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of non-zeroes. The benchmark consists of 15 iterations of an outer

Problem Class	Matrix Dimension	Nonzero Entries	Time (CPU)	Time (Wall)
Sample	1,400	78,148	187	189
Class A	14,000	1,853,104	3271	8353

Table 8.1: NAS CG Benchmark Timings

loop, which solves a sparse linear system $Az = x$. The sparse linear solver consists of 25 iterations, each of which does a sparse matrix-vector multiplication followed by a small number of element-wise vector additions and inner product computations.

The input to NAS CG is generated by a sequential FORTRAN program supplied by NAS [BBB+94]. Unfortunately, we were unable to get this program to run for systems as large as we would have liked to test. For $N = 28,000$, the program crashed at run-time, and for $N = 75,000$, the f77 compiler ran out of memory. The TPIE results for the problem sizes we were able to generate and solve are summarized in Table 8.1.

For the smaller of the two problem instances ($N = 1,400$), there is essentially no I/O overhead. This is a direct consequence of the fact that the entire problem fits in main memory, and thus TPIE never writes any intermediate data to the disk. The larger problem ($N = 14,000$), more accurately reflects the relative I/O and CPU cost we expect to see in larger problem instances. Once the sparse matrix has been pre-processed, all matrix-vector multiplications are essentially scans, with at most two floating point operations performed per scanned value.

The performance of our implementation is summarized in the graph shown in Figure 8.3.

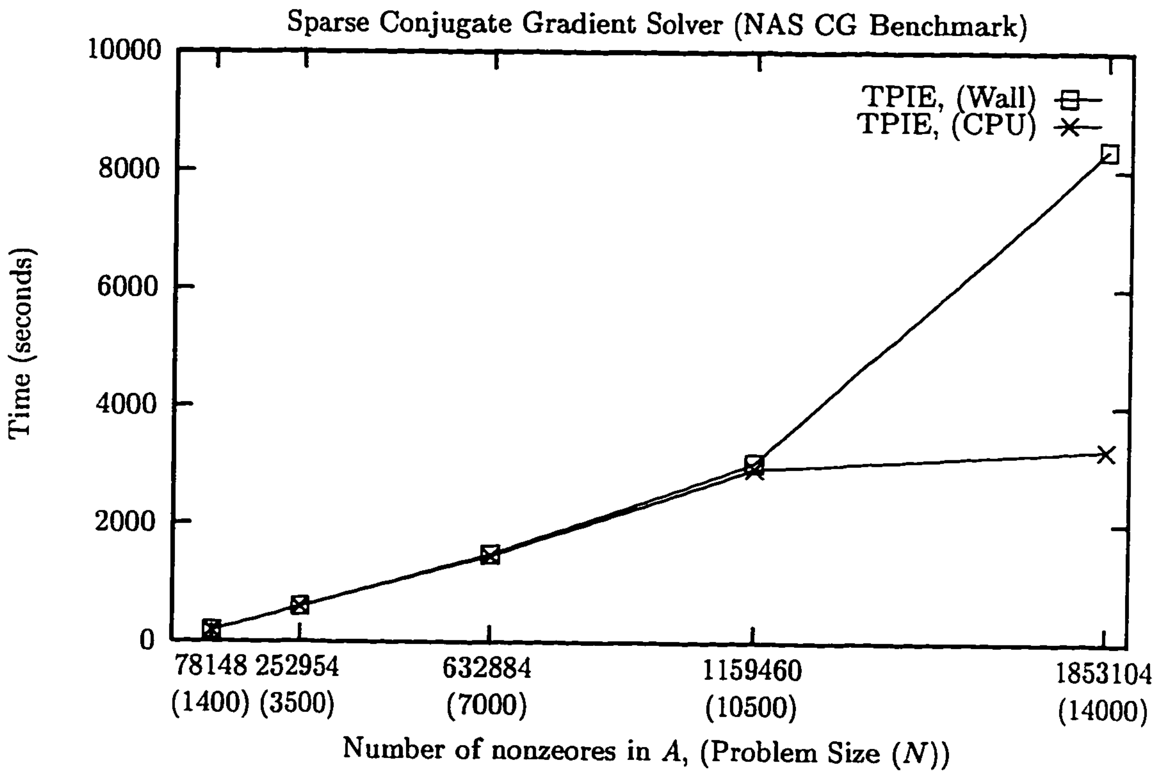

Figure 8.3: NAS CG Benchmark

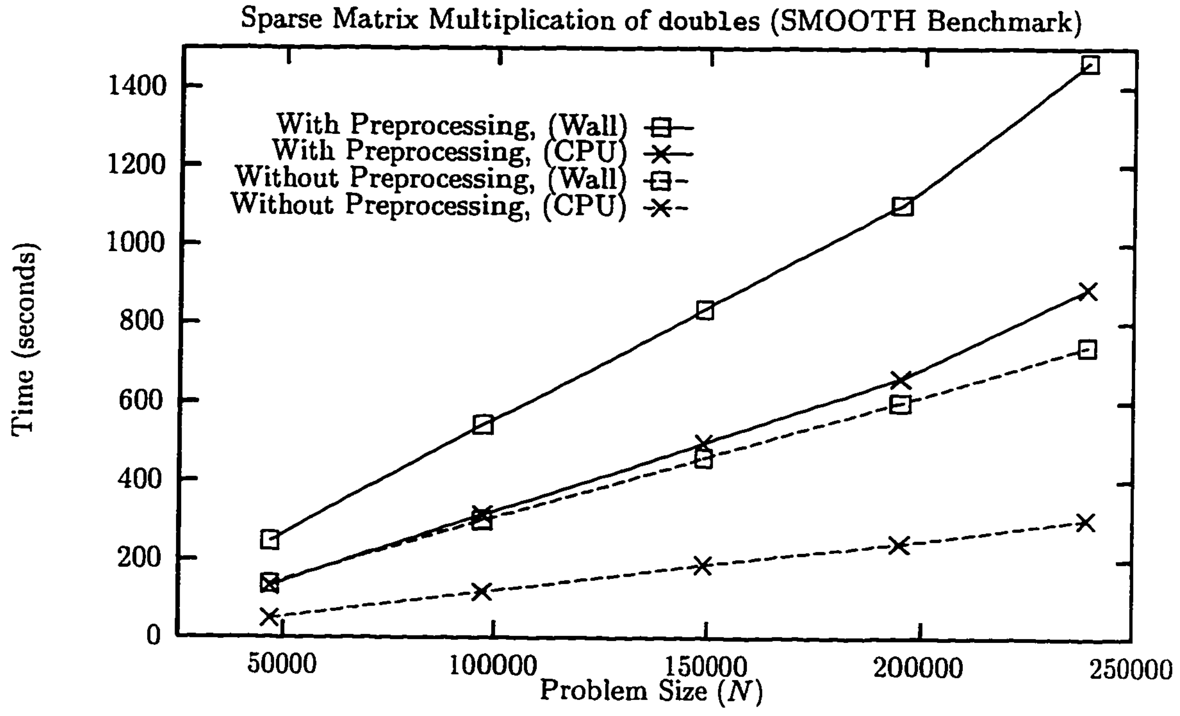

Figure 8.4: SMOOTH Benchmark

8.3.2 The SMOOTH Benchmark

The SMOOTH benchmark implements sparse matrix-vector multiplication between a $N \times N$ matrix with $27N$ nonzero elements and a dense N-vector. The result is then multiplied by the matrix again. Ten iterations are performed.

The performance of SMOOTH is shown in Figure 8.4. Although we do ten iterations of multiplication, and only pre-process once, the total time with preprocessing is significantly higher than that of the multiplication iterations alone. As expected, I/O is not a major contributor to this difference, because sorting only requires a small number of linear passes through the data. The big difference is in CPU time. The additional CPU time used in preprocessing the sparse matrix is roughly twice the CPU time used in all ten iterations of the multiplication.

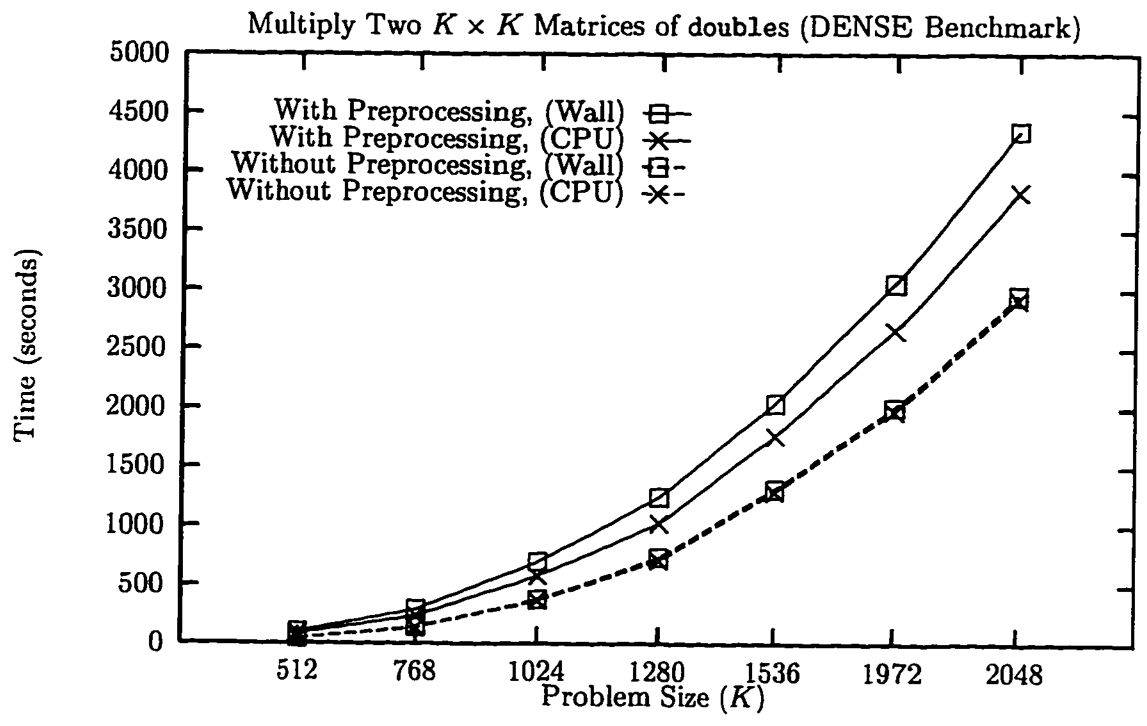

Figure 8.5: DENSE Benchmark

8.4 Dense Matrix Benchmark

We implemented a benchmark, called DENSE, which constructs two $K \times K$ matrices, pre-processes them, and then multiplies them. Times were recorded for both the total benchmark and for the multiplication only. The results are shown in Figure 8.5. As expected, the CPU time required to multiply the matrices follows a cubic path. Because of read-ahead, I/O is almost fully overlapped with computation, making the CPU and total time curves virtually indistinguishable. The cost of preprocessing the matrices is approximately one third of the cost of multiplying them. If several multiplications are done with the same matrix the preprocessing cost per multiplication will go down.

8.5 Conclusions

In this chapter we have shown empirically that a number of well-known benchmarks can be implemented in an I/O-efficient manner using TPIE. In a way this completes our journey, which began with the study of abstract models and algorithms, proceeded through the design of a system to implement these algorithms and the implementation of these algorithms using the system, and finally measured the real-world performance of the implementations. In the remaining chapter we will reflect on this journey and outline some of the research issues that lie on the road ahead of us.

Chapter 9

Conclusions

In this thesis we have approached the problem of performing I/O-efficient computations from a number of different angles. We began by discussing models of computation and basic algorithmic techniques. From there we moved on to discussions of specific algorithms for both combinatorial and scientific problems. Once we had discussed these algorithms, we turned to the task of designing systems to support their implementation. We discussed methods of designing such systems and why we chose a framework-oriented approach. We then introduced TPIE, and discussed its design and use, including extended examples. Finally, we evaluated the performance of code implemented in TPIE for several benchmark problems. Our contributions are summarized in Sections 9.1 through 9.3.

9.1 Algorithm Design and Analysis

In this thesis we developed a number of new algorithmic techniques. These include distribution sweeping and batch filtering for solving problems in computational geometry, a three-dimensional convex hull algorithm, a PRAM simulation technique, a new analysis and comparison of sorting algorithms, a new analysis of an existing dense matrix multiplication algorithm and a design for a more efficient one, and a new data structuring technique and algorithm for sparse matrix by vector multiplication.

9.2 TPIE: A Framework-Oriented I/O System

In order to implement the algorithms we designed and studied, we proposed the notion of a framework-oriented I/O system. We also described how this class of system fits into a taxonomy of existing systems. Existing systems were categorized into two types, access-oriented and array-oriented. Framework-oriented systems are designed to be easier to use and more efficient than either of these two types of systems. Framework-oriented systems allow programmers to deal with a simple abstract model of data on secondary storage devices without ever having to do explicit I/O. They also allow programmers to concentrate on the functional details of the applications they are writing within well-established I/O-efficient computation paradigms.

In addition to proposing the concept of a framework-oriented system, we developed the first such system, which we named TPIE. TPIE is designed to allow programmers to simply and flexibly implement a variety of I/O-efficient computations. It supports a number of paradigms of I/O-efficient computation, including scanning, permuting, sorting, and matrix operations. These paradigms, along with user-supplied functional details, can be used to construct a wide variety of algorithms.

9.3 Experimental Analysis of I/O-Efficient Algorithms

Using TPIE, we implemented a number of algorithms and collected experimental data on their performance. Our results represent a comprehensive cross section of I/O-efficient computations implemented using a framework-oriented approach. The results, on the whole, indicate that I/O-efficient computation can be made a reality in practice. By using the paradigms that TPIE provides us with, we are often able to remove the I/O bottleneck that might otherwise be seen in these applications. In many cases computations that we would expect to be I/O bound become CPU bound because the amount of I/O required when using one of our efficient algorithms is so small. Furthermore, we demonstrated that the CPU overhead of managing the levels of TPIE is a relatively small fraction of the overall CPU requirements of most of our applications.

9.4 Extending Our Work in I/O-Efficient Computation

Although we covered a great deal of ground, there is even more still open for exploration. In particular, we feel that the biggest obstacle to continued development of I/O-efficient computation systems is the current lack of support from operating systems. As was seen in Chapter 8, TPIE performance is quite good, but more support from the operating system could make it even better.

Integrating support for I/O-efficient computation into operating systems is a two way street; the operating system must provide appropriate mechanisms to support I/O-efficiency and systems such as TPIE must be augmented to make use of these mechanisms.

The two services that operating systems provide which are most closely related to I/O-efficient computation are demand-paged virtual memory and file systems. Both manage data on disks and move it back and forth between main memory and disks, but neither does so in a manner particularly suited to I/O-efficient computation.

One of the problems with both services is that it can be difficult or impossible for programs to specify which blocks of data should be replaced when new data is to be brought into an already full main memory. File systems make this difficult because of their use of a main memory buffer cache [Bac86, LMKQ89] which is, in almost all cases, completely inaccessible to user programs. Another drawback of buffer caches with respect to I/O-efficient computation is that they take away main memory that a program might otherwise be able to make use of.

Demand-paged virtual memory systems reduce the ability of programs to control what blocks are released by implementing page replacement policies that are designed to work for programs that exhibit considerable locality of reference. Typically these are based on some variant of a least recently used heuristic [Bac86, LMKQ89]. Some micro-kernels, such as Mach [ABB+86, BKLL93] and NT [Cus93], have taken steps towards allowing processes to manage their own virtual memory through user-level page fault handlers. These systems do not, however, allow processes to decide which of their pages should be evicted when main memory is used up. This task is still left to the kernel. More recently, some work has been done on kernels that allow processes to determine which of their pages to evict [CFL94]. The kernel now just decides which process should be forced to evict a page. These mechanisms were not specifically designed to support I/O-efficient computation, but they appear to offer a good starting point for the those wishing to build systems that are.

By combining TPIE or a similar framework-oriented system with an operating system that allowed more direct application level management of storage resources we believe we could construct a system that would be both easy for programmers to use and even more efficient than our current TPIE implementation.

9.5 Final Thoughts

Our work has addressed both the theory and practice of I/O-efficient computation. More importantly, we have attempted to pursue theory with an eye towards the problems of practice, and practice as informed by theory. We believe that by pursuing research in this manner the whole of our contribution to the field of I/O-efficient computation will necessarily be greater still than the sum of our contributions to its theory or practice. We have found this approach both challenging and fulfilling, and encourage other researchers to pursue I/O-efficient computation in a similar manner.

Bibliography

[AACS89] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. Technical Report RC 15118, IBM Watson Research Center, October 1989. An earlier version appeared in *Proceedings of Nineteenth Annual ACM Symposium on Theory of Computing*, pages 305–314, New York, NY, May 1987.

[ABB+86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *USENIX Conference Proceedings*, pages 93–112, Atlanta, GA, Summer 1986. USENIX.

[ACF90] B. Alpern, L. Carter, and E. Feig. Uniform memory hierarchies. In *Proceedings of 31st Annual IEEE Symposium on Foundations of Computer Science*, St. Louis, MO, October 1990.

[ACF93] B. Alpern, L. Carter, and J. Ferrante. Modeling parallel computers as memory hierarchies. In W. K. Giloi, S. Jahnichen, and B. D. Shriver, editors, *Working Conference on Massively Parallel Programming Models*, pages 116–123, Berlin, Germany, September 1993.

[ACFS94] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2/3):72–109, August and September 1994.

[ACG+88] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, 3:293–327, 1988.

[ACS87] A. Aggarwal, A. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proceedings of 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 204–216, Los Angeles, CA, October 1987.

[AHU74] Alfred V. Aho, John E. Hopcraft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[AL93] S. G. Akl and K. A. Lyons. *Parallel Computational Geometry.* Prentice-Hall, 1993.

[AM90] R. J. Anderson and G. L. Miller. A simple randomized parallel algorithm for list-ranking. *Info. Processing Letters*, 33(5):269–273, 1990.

[ANS90] ANSI X3J3/S8.115. Fortran 90, 1990.

[AP94a] A. Aggarwal and G. Plaxton. Optimal parallel sorting in multi-level storage. In *Proc. 4th Annual ACM-SIAM Symp. on Discrete Algorithms*, Arlington, VA, 1994.

[AP94b] Alok Aggarwal and C. Greg Plaxton. Optimal parallel sorting in multi-level storage. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 659–668, Arlington, VA, January 1994.

[Arg94] Lars Arge. The buffer tree: A new technique for optimal I/O-algorithms. Technical Report RS-94-16, BRICS, Univ. of Aarhus, Denmark, 1994.

[AV88] Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[AVV95] Lars Arge, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory algorithms for processing line segments in georgraphic information systems. In *Proceedings of the Third European Symposium on Algorithms (ESA)*, Corfu, Greece, September 1995.

[Bac86] Maurice J. Bach. *The Design of the Unix Operating System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[Bay72] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[BBB$^+$94] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Fredrickson, T. Lansinki, R. Schrieber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report RNR-94-007, RNR, March 1994.

[BGV97] Rakesh D. Barve, Edward F. Grove, and Jeffrey S. Vitter. Simple randomized mergesort on parallel disks. special issue on parallel I/O in *Parallel Computing*, 1997. to appear.

[BGvN89] Burks, Goldstine, and von Neumann. Preliminary discussion of the logical design of an electronic computing instrument (1946). In Zenon W. Pylyshyn and Liam J. Bannon, editors, *Perspectives on the Computer Revolution, Second Edition.* Ablex Publishing Corporation, 1989.

[BKLL93] Joseph Boykin, David Kirschan, Alan Langerman, and Susan LoVerso. *Programming under Mach.* Addison-Wesley, 1993.

[BM72] R. Bayer and E. McCreight. Organization of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[Bol79] B. Bollobas. *Graph Theory: An Introductory Course.* Springer–Verlag, New York, 1979.

[CBH$^+$94] Alok Choudhary, Rajesh Bordawekar, Michael Harry, Rakesh Krishnaiyer, Ravi Ponnusamy, Tarvinder Singh, and Rajeev Thakur. PASSION: parallel and scalable software for input-output. Technical Report SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, September 1994.

[CC94] Thomas H. Cormen and Alex Colvin. ViC*: A preprocessor for virtual-memory C*. Technical Report PCS-TR94-243, Dept. of Computer Science, Dartmouth College, November 1994.

[CFH$^+$95] Peter Corbett, Dror Feitelson, Yarson Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernard Traversat, and Parkson Wong. MPI-IO: A parallel file I/O interface for MPI. Technical Report NAS-95-002, NASA Ames Research Center, January 1995. Version 0.3.

[CFL94] Pei Cao, Edward W. Felten, and Kai Li. Implementation and performance of application-controlled file caching. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–177, 1994.

[CG86a] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1:133–162, 1986.

[CG86b] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: Ii. applications. *Algorithmica*, 1:163–191, 1986.

[CGG$^+$95] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 139–149, San Francisco, CA, 1995.

[Chi95] Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. In *Proc. 1995 Workshop on Algs. and Data Structures. (WADS)*, 1995.

[CK94] Thomas H. Cormen and David Kotz. Integrating theory and practice in parallel file systems. Technical Report TR94-223, Dept. of Math and Computer Science, Dartmouth College, 1994. Revised 9/20/94.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, 1990.

[Com79] D. Comer. The ubiquitous b-tree. *Computing Surveys*, 11(2):121–137, 1979.

[Cor92] Thomas H. Cormen. *Virtual Memory for Data Parallel Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992.

[Cor93] Thomas H. Cormen. Fast permuting in disk arrays. *Journal of Parallel and Distributed Computing*, 17(1–2):41–57, Jan./Feb. 1993.

[CSW94] Thomas H. Cormen, Thomas Sundquist, and Leonard F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. Technical Report PCS-TR94-223, Dartmouth College Dept. of Computer Science, July 1994.

[Cus93] Helen Custer. *Inside Windows NT.* Microsoft Press, 1993.

[CV86] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal list-ranking. *Information and Control*, 70(1):32–53, 1986.

[Ell90] T. M. R. Ellis. *Fortran 77 Programming: With an Introduction to Fortran 90 Standard.* Addison Wesley, second edition, 1990.

[ES90] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA, 1990.

[Flo72] R. W. Floyd. Permuting information in idealized two-level storage. In R. Miller and J. Thatcher, editors, *Complexity of Computer Calculations*, pages 105–109. Plenum Press, New York, 1972.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th Annual ACM Symp. of Theory of Computation (STOC)*, pages 114–118, 1978.

[Gib92] Garth A. Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage.* ACM Distinguished Dissertations. MIT Press, 1992.

[GLS94] W. D. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* Scientific and Engineering Computation. MIT Press, Cambridge, MA, 1994.

[Gra72] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132–133, 1972.

[GS78] Leo J. Guibas and Robert Sedgewick. A diochromatic framework for balanced trees. In *Proceedings of 19th Annual IEEE Symposium on Foundations of Computer Science*, pages 8–21. IEEE Computer Society, 1978.

[GS85] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graphics*, 4:74–123, 1985.

[GTVV93] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *IEEE Foundations of Comp. Sci.*, pages 714–723, 1993.

[Hoa62] C. A. R. Hoare. Quicksort. *Computer Journal*, 5:10, 1962.

[Joh91] Mike Johnson. *Superscalar Microporcessor Design.* Prentice Hall, 1991.

[JW75] K. Jensen and N. Wirth. *PASCAL: user manual and report.* Springer-Verlag, 1975.

[Kar91] R. M. Karp. Probabilistic recurrence relations. In *Proc. 23rd Annual Symposium on Thery of Computation*, pages 190–197, 1991.

[Kim86] Michelle Y. Kim. Synchronized disk interleaving. *IEEE Transactions on Computers*, C-35(11):978–988, November 1986.

[Knu73] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison Wesley, 1973.

[Knu81] Donald E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison Wesley, 2d. ed. edition, 1981.

[KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall Software Series. Prentice-Hall, Englewood Cliffs, NJ, 1978.

[KS86] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal of Computing*, 15:287–299, 1986.

[LKB87] Mivan Livny, Setrag Khoshafian, and Haran Boral. Multi-disk management algorithms. In *Proceedings of the 1987 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 69–77, May 1987.

[LMKQ89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System.* Addison-Wesley, Reading, MA, 1989.

[LRT93] Charles E. Leiserson, Satich Rao, and Sivan Toledo. Efficient out-of-core algorithms for linear relaxation using blocking covers. In *IEEE Foundations of Comp. Sci.*, pages 704–713, 1993.

[NV90] M. H. Nodine and J. S. Vitter. Large-scale sorting in parallel memories. In *Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures*, pages 29–39, 1990.

[NV92] Mark H. Nodine and Jeffrey Scott Vitter. Optimal deterministic sorting in parallel memory hierarchies. Technical Report CS–92–38, Brown University, August 1992.

[NV93a] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, June 1993.

[NV93b] M. H. Nodine and J. S. Vitter. Paradigms for optimal sorting with multiple disks. In *Proc. of the 26th Hawaii Int. Conf. on Systems Sciences*, January 1993.

[NV93c] Mark H. Nodine and Jeffrey Scott Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 120–129, Velen, Germany, June 1993.

[Pat94] Yale N. Patt. The I/O subsystem—a candidate for improvement. *IEEE Computer*, 27(3):15–16, March 1994.

[PCGK89] D. A. Patterson, P. Chen, G. Gibson, and R. H. Katz. Introduction to redundant arrays of inexpensive disks (RAID). In *IEEE COMPCON 89, San Francisco, Feb.-*, March 1989.

[PGK88] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. ACM SIGMOD Conf.*, page 109, Chicago, IL, June 1988.

[PH77] F. P. Preparata and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Communications of the ACM*, 20(2):87–93, 1977.

[PS85] Franco P Preparata and Michel Ian Shamos. *Computational geometry : an Introduction.* Springer-Verlag, 1985.

[RS92] John R. Reif and Sandeep Sen. Optimal parallel randomized algorithms for three-dimensional convex hulls and related problems. *SIAM J. Computing*, 21(3):466–485, 1992.

[RW94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[Sed75] R. Sedgewick. *Quicksort.* PhD thesis, Stanford Univ., Stanford, CA, May 1975.

[Sed78] R. Sedgewick. Implementing quicksort programs. *Comm. of the ACM*, 21:847–856, 1978.

[SGM86] Kenneth Salem and Hector Garcia-Molina. Disk striping. In *Proceedings of the 2^{nd} International Conference on Data Engineering*, pages 336–342. ACM, February 1986.

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

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[SW] Elizabeth A. M. Shriver and Leonard F. Wisniewski. Choreographed disk access using the whiptail file system: Applications. manuscript.

[SW94] Kent E. Seamons and Marianne Winslett. An efficient abstract interface for multidimensional array I/O. In *Supercomputing '94*. IEEE Computer Society, 1994.

[SW95] Elizabeth A. M. Shriver and Leonard F. Wisniewski. An API for choreographing data accesses. Technical Report PCS–TR95–267, Dartmouth College Department of Computer Science, October 1995.

[SWC+95] Elizabeth A. M. Shriver, Leonard F. Wisniewski, Bruce G. Calder, David Greenberg, Ryan Moore, and David Womble. Parallel disk access using the Whiptail File System: Design and implementation. Manuscript, 1995.

[TV90] R. Tamassia and J. S. Vitter. Optimal cooperative search in fractional cascaded data structures. In *Proceedings 2nd ACM Symosium on Parallel Algorithms and Architectures*, pages 307–316, 1990.

[Ven94] Darren Erik Vengroff. A transparent parallel I/O environment. In *Proc. 1994 DAGS Symposium on Parallel Computation*, July 1994.

[Ven95] Darren Erik Vengroff. *TPIE User Manual and Reference*. Duke University, 1995. Available via WWW at `http://www.cs.duke.edu:~dev/tpie.html`.

[VS94a] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2):110–147, 1994.

[VS94b] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(3):148–169, 1994.

[VV95a] Darren Erik Vengroff and Jeffrey Scott Vitter. I/O-efficient scientific computation using TPIE. Technical Report CS–1995–18, Duke University Dept. of Computer Science, 1995.

[VV95b] Darren Erik Vengroff and Jeffrey Scott Vitter. Supporting I/O-efficient scientific computation in TPIE. In *Proc. Seventh IEEE Symposium on Parallel and Distributed Processing (SPDP '95)*, 1995.

[Wir71] Niklaus Wirth. The programming language pascal. *Acta Informatica*, 1(1):35–63, 1971.

[X3J78] X3J3 Subcommittee. *American National Standard Programming Language Fortran (X3.9-1978)*. American National Standards Institute, 1978.

[Zhu94] B. Zhu. Further computational geometry in secondary memory. In *Proc. Int. Symp. on Algorithms and Computation*, 1994.