Minimizing the Input/Output Bottleneck

Mark Howard Nodine

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-92-48
October 1992

Minimizing the Input/Output Bottleneck

by
Mark Howard Nodine
B. S., Tulane University, 1978
B. A., Tulane University, 1978
S. M., Massachusetts Institute of Technology, 1982
S. M., Harvard University, 1986
Sc. M., Brown University, 1988

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 1993

This dissertation by Mark Howard Nodine
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 _____ _____

Jeffrey Vitter

Recommended to the Graduate Council

Date _____ _____

Daniel P. Lopresti

Date _____ _____

Alok Aggarwal

Approved by the Graduate Council

Date _____ _____

Vita

The author was reared in the suburbs of Philadelphia and in 1974 was graduated from the Conestoga Senior High School in Berwyn, PA. He has computed that at his twentieth high school reunion, he will have spent $13\frac{1}{2}$ of those years in school full-time and another year part-time in his seemingly never-ending quest to become educated. He spent the first four of those years at the Tulane University of Louisiana, where he got two degrees (B.A. and B.S.) and three majors (Mathematics, Physics, and Chemistry). He subsequently went to graduate school at MIT in Physical Chemistry, where he received the S.M. degree in 1982. After working for a year and a half at Applicon (Schlumberger) and during the middle of four years at Bolt Beranek and Newman, he matriculated in 1985 at Harvard University and was awarded the S.M. degree in Computer Science in 1986. Since 1987, he has been working on his doctorate in Computer Science at Brown University, and with the completion of this thesis hopes to find a Real Life.

The author has a wife and two children, enjoys playing the piano and softball, and is an elder and church planter in the East Providence Fellowship of House Churches.

Preface

In this thesis, we examine ways of ameliorating the input/output (I/O) bottleneck through developing special-purpose algorithms tailored to specific I/O environments. We give provably optimal deterministic algorithms for sorting on parallel disks and in parallel memory hierarchies. The memory hierarchies considered are the Hierarchical Memory Model (HMM), the Block Transfer model (BT), and the Uniform Memory Hierarchy (UMH). We give an algorithm for sorting that is simultaneously optimal in terms of I/Os and internal processing time in a model where we have D disks and P processors, as long as P does not exceed $M \log \min\{M/B, \log M\}/\log M$. We give the first known algorithms for sorting efficiently in single Uniform Memory Hierarchies.

We show how to achieved optimal I/O performance of VLSI implementations of lattice computations by transferring less information, and give matching upper and lower bounds. Finally, we show how to use blocking efficiently for external graph searching.

Acknowledgements

There are so many who have helped me to get to the point of writing this thesis, that it almost seems that I have a debt to everyone. I would like to thank my advisor Jeff Vitter first and foremost, who always believed in me and was able push me towards extending my results into new areas. He was patient in teaching me and was excellent at sifting the good ideas I had from the bad ones.

I want to thank IBM and the Brown Center for the Advancement of College teaching for providing fellowships to support me during my tenure here. Thanks are also due to Jeff Vitter, Andy van Dam, Philip Klein, and Dan Lopresti for writing me letters of recommendation that resulted in my getting the IBM fellowship for two years.

There were many people with whom I had helpful discussions. The computer science faculty have been quite helpful, and at one time or another I have spoken with Andy van Dam, Tom Doeppner, John Hughes, Paris Kanellakis, Philip Klein, Dan Lopresti, Franco Preparata, Steve Reiss, John Savage, and Roberto Tamassia about topics related to my thesis. Of these, I would like to single out Andy van Dam, John Hughes, Philip Klein, and Franco Preparata for special thanks. Professor Tom Banchoff in the mathematics department also provided helpful information about tessellations in arbitrary dimensions. Students who have had some input into this thesis are Ajit Agrawal, Bob Cohen, P. Krishnan, Marian Nodine, Ravi Ramamurthy, S. Sairam, Solomon Shimony, Liddy Shriver, and Darren Vengroff. I would especially like to thank P. Krishnan for his friendship and helpful comments. Finally, thanks are due to Professor Mike Goodrich in the computer science department at Johns Hopkins University, with whom I collaborated on the graph searching work.

Additionally, there were many who provided non-technical support. Thanks to Ted Camus, who graciously provided a refrigerator in my office where I could store my lunch without its being snitched. Thanks to Vishwanath Ramachandran for keeping things light around the office. Thanks to Bob Zeleznik, Phillip Hubbard, Oren Tversky and many others for giving me a softball team to play on. Thanks to Mary Andrade for handling the UPS paperwork for me countless times as well as sending out requested papers to many. Thanks to Kathy Kirman and Tina Cantor for their cheerful smiles.

There are many folks outside the department I would also like to thank. My wife has been the best friend to me that anyone could wish for. She picked me up time after time when I became discouraged, and gave me the support with the home and the children without which I would not have been able to accomplish my research. I am also appreciative of my children Timothy and Anna, without whom I would have

finished much sooner, but not had nearly as much fun. I owe thanks to my church as well for praying for me and teaching me how to become more mature as a person. Special thanks go to Ted Leung, who was a bridge between the department and the church, and Dick Scoggins, who spent much time with me one-on-one.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A major aim of computer science has been to develop algorithms that operate as efficiently as possible. In particular, computer scientists have been interested in the performance of algorithms when the size of the problem N gets large. For sufficiently small values of N, there might not be much difference between, say, an algorithm that runs in time kN^3 and one that runs in time $k'N^4$, where k and k' are constants. In fact, if $k' < k$, the latter algorithm may be faster for small values of N. However, no matter what k and k' are, there is some value N_0 such that for any $N > N_0$, the former algorithm will run faster. For this reason, designers of algorithms concentrate on the *asymptotic* performance of the algorithm, and have a tendency to ignore the constants involved. The former algorithm would be characterized as an $O(N^3)$ algorithm and the latter as an $O(N^4)$ algorithm, where the big-O swallows up any constants of proportionality.

Researchers have been primarily interested in the part of the running time of an algorithm involved in doing the computation; the fact that the central processor is only a small part of the overall computer system has been traditionally ignored in the design of algorithms. One exception to this trend has been an attempt to model communication delays in some of the architectures for parallel and distributed processing. A modern computer system typically has much more involved, for example, graphics input/output devices, caches, main memory, secondary storage, archival storage, keyboard input devices, clocks, network interfaces, special-purpose processors (e.g., for serial communication or graphics), and printers. The actual performance of the computer system for a given task is in general related to the amount of time needed in each of the subcomponents of the system that are involved in the task, plus the communication time needed. For example, in printing a simple text document, the total speed is usually determined more by the speed of the printer or the speed of the communication between the processor and the printer, than by the amount of computation time needed by the processor.

In this thesis, we consider a different component of the computer system from the processor, one that is often the bottleneck in large-scale computations, and that is the input/output (I/O) subsystem. In interactive applications, the bottleneck in

the computer's performance is usually the user, and while there is a rich and diverse literature on improving the effectiveness of the communication between the user and the computer, those issues are not the ones that we have chosen to address in this thesis. We will therefore concentrate on non-interactive applications, and I/O between devices.

The advances in Very Large-Scale Integrated circuit (VLSI) technology has resulted in an abundance of special-purpose processors. VLSI has obvious uses for things like serial communications and graphics. People are also implementing chips for such things as solving the Poisson equation in two dimensions [Man] and neural network computations [PLK, TML, WHA]. The advantage of using VLSI is that it is possible to create very computation-efficient special-purpose processors. However, an area that is not studied as diligently is the I/O needs of VLSI chips. In particular, it is possible for the I/O to become the bottleneck in the use of special-purpose processors. Cypher has examined some of the theoretical aspects of VLSI pin limitations [Cyp].

Even in general-purpose computer systems, any algorithm is going to require I/O when the size of the problem gets to be too large for internal memory. These difficulties are not merely of theoretical interest, since accessing secondary storage such as disks is slower by orders of magnitude than accessing internal memory. The I/O time typically dominates the total running time. Developing efficient I/O strategies can have a significant impact on the performance for problems that are too large for internal memory.

The usual way of examining I/O complexity has been to think about the communication time needed to move items from main memory to secondary storage, most often a disk in modern computer systems, as discussed above. This is an example of the simplest I/O model, the *two-level memory hierarchy*, where memory is partitioned into two levels: the small, fast *internal memory* and the much larger, but slower *external memory* (or secondary storage). The internal memory can be thought of as the main memory in a computer and the external memory as the disk subsystem, but the model applies equally well to a fast cache vs. main memory, a disk vs. tape archives, or in VLSI, on-chip vs. off-chip memory. Floyd considered an idealized two-level storage where the slow memory was separated into pages of size p, and looked at the problem of permuting [Flo].

In the two-level memory hierarchy, the data elements reside initially in the external memory and the goal of the computation is to write the answer to the problem back into external memory. In order for the algorithm to use any data element, however, it must move it into internal memory, incurring a cost of one I/O. If the intermediate results grow too numerous to be stored in internal memory, they can also be written to the external memory at a cost of one I/O. Writing an element of the answer to the external memory also costs one I/O. The sum of all the I/O costs is the measure of the I/O efficiency of the algorithm. Of course, this quantity is only of interest if the sum of the problem size and the intermediate results is larger than the internal memory; otherwise the I/O complexity is trivially no more than $2N$, where N is the size of the problem: N to read the problem and N more to write it out again.

There are several elaborations to the two-level memory model. The first elaboration

Figure 1.1: An idealized view of a disk, showing the sources of the various delays.

arises from the fact that the two-level memory hierarchy has three components to the delay caused in communicating between external and internal memory when the external memory is a disk. These components are caused by the physical construction of a disk. An ideal disk can be viewed much like a record player, as shown in Figure 1.1. Unlike a record player, the disk is not "played" in order. The usable area on the disk is broken up into concentric circles called *tracks* (or cylinders), and each of the tracks is broken into a series of smaller sections called *blocks* (or sectors). The disk has a read/write head that can extend or retract as needed to get to the right track, and the whole disk is spinning rapidly so that every sector of the current track will eventually pass under the read/write head. This construction of a disk gives rise to three components of the delay.

1. *Seek time.* The seek time is the amount of delay needed to retract or extend the read/write head to get to the right track.

2. *Latency time.* The latency is the amount of rotational delay until the read/write head reaches the proper spot to begin reading the sector.

3. *Transmission delay.* The transmission delay is the rotational delay while the data

passes under the read/write head.

In practice, the seek time dominates the total delay, with the transmission delay for an element of data being almost negligible. Accordingly, an elaboration of the two-disk model is to consider that transferring a block of information, rather than a single element, takes unit time. So one of the areas of research in two-level memory hierarchies deals with how to use blocking efficiently to take maximal advantage of the fact that transmission time is inconsequential.

A second elaboration to the two-level memory hierarchy is motivated because information processing is currently facing what several researchers have called an input/output (I/O) crisis [CGK, Che, PGK]. CPU speeds have been increasing approximately exponentially over the last decade, as have memory capacities [PGK]. Unfortunately, disk speeds have seen little improvement in rotational latency and only a halving of seek times [CGK]. Even if we consider the improvements in disk technology to be exponential, it is of a lower order than that seen by processors. The recent trend toward parallel computers aggravates the I/O bottleneck further, since improving the performance of one part of a subsystem while leaving other parts unchanged does not take full advantage of those improvements [Amd]. It seems that the only way around the I/O bottleneck is to use multiple disks in parallel. This approach is taken, for example, in the TWA reservation system [GiS]. So the second elaboration is being able to handle disk parallelism.

There are two traditional ways of tackling the I/O bottleneck: general-purpose algorithms and special-purpose ones. With general-purpose algorithms, the operating system takes care of handling the I/Os efficiency. For example, in a two-level memory, the operating system would allow the algorithm designer to view all of memory as being of unit cost to access, and would take care of paging data in on an as-needed basis. When a datum is referenced that is not in memory, this reference causes a *page fault*, which is serviced by reading in the block containing that datum. The operating system implements a *caching* algorithm that keeps blocks of data in the fast memory as long as possible, flushing a block out only when it needs to make room for a new block to be read from the disk. The decision of which block to remove is the source of a number of different algorithms, and this topic has been widely studied [BKT, CKP, FKL, McS, RaS, SlT]. Other expedients such as prefetching and clustering [PeS] have been proposed to improve performance. The difficulty with all of these schemes is that the operating system must make some assumptions about the ways that data will be accessed (such as "locality of reference"), and those assumptions may not always hold.

The second approach to tackling the I/O bottleneck is by using special-purpose algorithms designed with the specific problem and architecture in mind. Experience has shown that even a small amount of cognizance of the I/O environment in designing an algorithm can reap huge rewards in the I/O performance, and hence in the overall performance of the algorithm. For example, in programming on a computer that stores its arrays in row-major order, simply changing the order of a loop to access the array in row-major rather than column-major order can make an enormous difference in the number of page faults caused in a computation, no matter what reasonable paging

algorithm is used. This increase comes because the paging algorithm cannot know the needs of the specific application, and can only hope to be generally applicable to a broad range of applications. This thesis focuses specifically on developing special-purpose algorithms that are optimal for the problem and architecture.

There is a long history of developing special-purpose algorithms with a view towards I/O efficiency. As mentioned above, Floyd considered an algorithm for permuting on two-level memory hierarchies [Flo]. Knuth has 132 pages devoted to the I/O efficiency of sorting, all but 17 of which deal with the problems of using tapes [Knu]. Even considered the problem of sorting using parallel tapes with parallel processors [Eve], but his algorithms are hampered a bit by the fact that he assumes that each of the P processors has exactly 4 tape drives attached to it. Hong and Kung gave the first general techniques for proving lower bounds on I/O complexity, considering a number of different problems [HoK]. Bitton *et al.* consider a special case of parallel disk sorting where the processors comprise a tree, with a disk attached to each of the leaf nodes [BDH, Fri]. Savage and Vitter examined the effects of block size on a number of problems, including FFT and matrix multiplication [SaV]. Aggarwal and Vitter introduced a parallel disk model and gave algorithms for matrix multiplication, permuting, FFT, and sorting in their model [AgV]. Vitter and Shriver gave a more realistic parallel disk model and gave optimal algorithms for the same suite of problems as considered by Aggarwal and Vitter [ViSa]. Ullman and Yannakakis investigated the input/output complexity of the transitive closure problem [UlY]. Carlson showed how to use local memory judiciously in order to compute the FFT efficiently on a CRAY-2 computer [Car]. Cormen gave fast algorithms for permuting in the Vitter-Shriver parallel disk model [Cor].

It is sometimes unrealistic to deal with a computer's memory as comprising only two layers. Modern computer systems often have many levels of hierarchy such as CPU registers, fast CPU cache, main memory, hardware cache, local fast disk, slower network file servers, and finally archival storage. In progressing down this list, the memory modules become successively slower and much larger. Accordingly, several researchers have dealt with memory hierarchies as a way of modeling the entire I/O subsystem pertaining to memory. These hierarchical memory models can also be elaborated by considering several of them operating in parallel.

A number of authors have investigated memory hierarchies. Mattson *et al.* gave evaluation techniques for paging algorithms in linear memory hierarchies, proposing a class of paging algorithms called "stack algorithms" from which they derive an optimal page replacement algorithm [MGS]. Silberman proposed an interesting memory hierarchy in which the CPU is directly connected to the lowest k levels of the hierarchy, so that a page "miss" only occurs if the needed page is on level $k + 1$ or greater [Sil]. Aggarwal *et al.* advanced the first model of memory hierarchies, HMM, on which they considered special-purpose algorithms, including matrix multiplication, FFT, and sorting [AAC]. Aggarwal *et al.* also proposed a follow-up model called BT in which they added blocking considerations to the HMM memory model [ACSa]. Alpern *et al.* gave a somewhat more constrained model of memory hierarchies, the UMH model, with interesting blocking issues and parallel transfer between levels [ACF]. Luccio considered

a model called LPM based on pipelined access to memory that contains many of the same considerations as are present in memory hierarchies [LuP]. Bailey investigated the performance of FFT in external or hierarchical memory, implementing a version of FFT that requires as few as two passes through the data to compute a 2^n-point FFT [Bai].

In this thesis, we will occasionally need to consider processor parallelism as well as memory parallelism. Processor parallelism is generally accomplished in the real world by connecting the processors, each with its own private memory, using some kind of network topology such as a hypercube, butterfly switch, or cube-connected cycles. Theoreticians prefer to be more abstract, and so developed a model of parallel computation called the Parallel Random Access Machine (PRAM). In the PRAM model, each processor is connected directly to a global memory (or alternatively, directly to the local memory of all the other processors as well as its own). This model is further distinguished according to what kinds of simultaneous accesses are allowed to any particular memory location. If at all times the processors are accessing different memory locations — note that this is a property of the algorithm — we say that the PRAM needed is of the Exclusive Read/Exclusive Write (EREW) variety. This model is the weakest PRAM model. The model can be made somewhat stronger by allowing Concurrent Read along with Exclusive Write (CREW), and stronger still by allowing Concurrent Read and Concurrent Write (CRCW). The algorithms in this thesis that use the PRAM model all require the CRCW-PRAM.

Chapters 2 and 3 both address the issue of finding an optimal deterministic algorithm for large-scale sorting on parallel arrays of disks with blocking. The Greed Sort algorithm presented in Chapter 2 was the first to demonstrate that it is possible to achieve optimal performance for this problem deterministically. Unfortunately, Greed Sort has the disadvantages that the algorithm can not be directly programmed on some current disk arrays because it requires being able to write to different tracks of each of the parallel disks simultaneously and that we do not know how to extend it to parallel memory hierarchies. Chapter 3 presents a new optimal deterministic algorithm for this large-scale sorting problem, called Balance Sort. Balance Sort has none of the deficiencies pointed out for Greed Sort, and it has an especially small constant factor in its running time. There is a simple version of Balance Sort that we discovered during our quest for it; it appeared that this simple version would be optimal for the problem. However, after much time spent, we were unable to complete the proof of optimality of the simple algorithm. For the interested researcher, we present in Appendix B what progress we made towards the proof of optimality of the simple algorithm. Continuing the theme of sorting in different memory models, we present in Chapter 4 the first known completely deterministic results for sorting in parallel memory hierarchies comprising HMM and BT hierarchies. Finally, in Chapter 5 we investigate for the first time sorting in UMH along with it parallelization and other variants.

In Chapter 6, we talk about the communication required in implementing a special-purpose processor in VLSI for lattice computations. We show that the extent to which I/O is a bottleneck in such a special-purpose processor relates to the amount of information we need to get from the processor, although ironically, a certain minimum

amount of information needs to be gotten in order to achieve optimal I/O performance; further decreasing the amount of information gleaned from the chip results in a worse I/O bottleneck. The computation of the optimal amount of information to be retained resulted in an interesting discrete minimization formula, for which we were able to prove that rounding to the nearest integer always produced the discrete minimum. The proof of this surprising fact is presented in Appendix A.

Chapter 7 considers the problem of blocking efficiently to permit large-scale graph searching. Our fundamental conclusion is that a certain amount of redundancy of information is required in order to permit efficient searching of graphs.

Chapter 2

Sorting on Parallel Disks: Greed Sort

2.1 Introduction

Sorting consumes roughly 20 percent of computing resources in large-scale installations [Knu, LiV]. Of particular importance is external sorting, in which the records to be sorted are too numerous to fit in the processor's main memory and instead are stored in secondary storage, typically made up of one or more magnetic disks. The bottleneck in external sorts is the input/output (I/O) operations. Typically data are transferred in units of *blocks*, which may consist of several kilobytes. This approach takes advantage of the fact that the seek time is usually much longer than the time needed for transferring a record of data once the disk read/write head is positioned. An increasingly popular way to get further speedup is to use many disk drives working in parallel [GHK, GiS, Jil, Mag, PGK, Uni].

Initial work in the use of parallel block transfer for sorting was done by Aggarwal and Vitter [AgV]. In their model, they considered the parameters

$$\begin{aligned} N &= \#\text{ records in the file} \\ M &= \#\text{ records that can fit in internal memory} \\ B &= \#\text{ records per block} \\ D &= \#\text{ blocks transferred per I/O} \end{aligned}$$

where $M < N$, and $1 \leq DB \leq M/2$. In each I/O, D blocks of B records can be transferred simultaneously, as illustrated in Figure 2.1. This model generalized the initial work on I/O of Floyd [Flo] and Hong and Kung [HoK]. Aggarwal and Vitter proved that the average-case and worst-case number of I/Os required for sorting is[1]

$$\Theta\left(\frac{N}{DB}\frac{\log(N/B)}{\log(M/B)}\right). \tag{2.1}$$

[1] We use the notation $\log x$ to denote the quantity $\max\{1, \log_2 x\}$.

Figure 2.2: A more realistic D-parallel two-level memory model.

Their lower bound is based solely on routing arguments, except for the pathological case in which M and B are extremely small, in which case the comparison model is used. They gave two algorithms, a modified merge sort and a distribution sort, that each achieved the optimal I/O bounds. Likewise, they showed that the number of I/Os required to transpose a $p \times q$ matrix is

$$\Theta\left(\frac{N}{DB}\left(1+\frac{\log \min\{M, \min\{p,q\}, N/B\}}{\log(M/B)}\right)\right). \tag{2.2}$$

Vitter and Shriver considered a more realistic *D-disk model*, in which the secondary storage is partitioned into D physically distinct disk drives [ViSa, ViSb], as in Figure 2.2. (Note that each head of a multi-head drive could count as a distinct disk in this definition, as long as each could operate independently of the other heads on the drive.) In each I/O operation, each of the D disks can simultaneously transfer one block of B records. Thus, D blocks can be transferred per I/O, as in the [AgV] model, but only if no two blocks access the same disk. This assumption is very reasonable in light of the way real systems are constructed.

The measure of performance is the number of parallel I/Os required; internal computation time is ignored. In practice, though, the algorithms dealt with are also very efficient in terms of internal processing. This model also applies to the case in which

each of the D disks is controlled by a separate CPU with internal memory capable of storing M/D records, and the D CPUs are connected by a network that allows some basic operations (like sorting of the M records in the internal memories) to be performed quickly in parallel. The bottleneck can be expected to be the I/O.

Vitter and Shriver presented a randomized version of distribution sort using two complementary partitioning techniques. Their algorithm meets the I/O bound (2.1) for the more lenient model of [AgV], and thus the algorithm is optimal. They posed as an open problem the question of whether there is an optimal algorithm that is deterministic. They also showed that matrix transposition can be done optimally in the D-disk model, achieving the bound (2.2).

Disk striping is a commonly used technique in which the D disks are synchronized, so that the D blocks accessed during an I/O are located at the same relative position on each disk. This technique effectively transforms the disks into a single disk with larger block size $B' = DB$. Merge sort combined with disk striping is deterministic, but the number of I/Os used can be much larger than optimal, by a multiplicative factor of $\log(M/B)$.

In the area of parallel computing, Leighton introduced the columnsort algorithm as an optimal sorting algorithm on linear-sized sorting networks [Lei]. It was pointed out in [AgV] that columnsort can be used for optimal sorting with respect to I/O if N and B are not too large. Since columnsort is a (non-adaptive) sorting network algorithm, its I/O schedule can be pre-specified independently of the data to take full advantage of parallel block transfer.

In this chapter, we answer the open question posed in [ViSa] and present an optimal deterministic sorting algorithm. The algorithm, which we call *Greed Sort*, is an interesting variant of merge sort. In each merge pass, a priority scheme guarantees that each record ends up sufficiently close to its correct position, so that another pass of columnsort can complete the merging.

2.2 The Greed Sort Algorithm

We start by defining an order on the locations on the disks. Let $B_{i,j}$ denote the ith block stored on disk j, and let $R_{i,j,k}$ denote the kth record in $B_{i,j}$. We order the locations so that

- Record $R_{i,j,k}$ precedes $R_{i,j,k+1}$ for all i, j, k.
- Block $B_{i,j}$ precedes block $B_{i,j+1}$ for all i, j.
- Block $B_{i,D}$ precedes block $B_{i+1,1}$ for all i.

Now we can describe the sorting problem.

Problem instance: A series of N records, each containing a key field, located in the first N locations on the disk.

Figure 2.4: Assume that run 1 has the block with the smallest minimum and run 2 has the block with the smallest maximum. Then this figure depicts the state of the sort after the blocks have been processed.

Goal: To permute the records so that the key fields are in non-decreasing order in the first N locations on the disk.

Figure 2.3 gives an example of a sorted list with $B = 1$.

Our Greed Sort algorithm is a variant of merge sort. We create initial input runs (sorted lists) of size N/M by repeatedly reading in a memoryload, sorting it, and writing it back to the disks. In each subsequent pass, we merge $R = \sqrt{M/B}/2$ input runs at a time to form larger runs, which become the input runs for the next pass.

The basic idea behind Greed Sort is the following: Let us assume that each of the R runs to be merged is stored consecutively on disk, beginning on disk 1 and cycling through the D disks. In each parallel read operation, the one or two "best" available blocks from each disk are read into primary memory: the block with the smallest

Figure 2.6: The shift operation used in Steps 6 and 8 of columnsort.

minimum key value and the block with the smallest maximum key value. Treating each of the D disks independently, we sort its $2B$ records in primary memory and write the smallest B to the output list that we're forming; we put the largest B records back at the front of the run from which the smallest minimum was taken (note that this run remains sorted after this operation). See Figure 2.4. Ties are broken arbitrarily. If the blocks with the smallest minimum and the smallest maximum are the same block, we read only the one block and write it to the output list.

This operation results in an "approximately merged" list. The crucial observation is that the records are within $RDB = D\sqrt{MB}/2$ positions of their correct sorted locations. By an appropriate use of clustering throughout the course of the algorithm, this approximately merged list can be completely merged by a single pass consisting of several applications of the columnsort algorithm of Leighton [Lei] to subfiles of size $D\sqrt{MB}$. Then the next merge begins.

Columnsort is easiest to visualize as sorting into column-major order a matrix with r rows and c columns, with the requirement that c divides r and $r > 2(c-1)^2$. It has eight steps, of which the odd-numbered steps are all the same: sort all the records in each column. Steps 2 and 4 are a transpose operation as shown in Figure 2.5. Steps 6 and 8 amount to a cyclical shift by $r/2$, and are shown in Figure 2.6.

The pseudocode for the Greed Sort algorithm is given in Algorithm 1. Since all the records within a run are sorted, consecutive blocks on the same disk from a run are in non-decreasing order. Thus, the best available block on a given disk must be the next unread block on that disk from some run. The collection of next unread blocks, one for each (run, disk) pair, is called the *candidate set* of blocks and is stored in a priority queue. There are D disks and R runs, so the candidate set has cardinality DR.

We assume for convenience that the runs are separated on each disk by blocks containing the key value $+\infty$ larger than any key. The algorithm uses $mark[i,j]$ to keep track of the next block of run i to be read from disk j. The set of all blocks $mark[i,j]$ comprises the candidate set. The maximum and minimum key fields of block $mark[i,j]$ are stored in $biggest[i,j]$ and $smallest[i,j]$, respectively. We do our I/O operations into the buffers $b1$ and $b2$, which each consist of D smaller buffers. Buffers $b1[j]$ and $b2[j]$, for $1 \leq j \leq D$, each hold B records from disk j; we denote their maximum and minimum keys by $\max(b1[j])$, $\min(b1[j])$, $\max(b2[j])$, and $\min(b2[j])$.

2.2.1 Proof of correctness

The correctness of Greed Sort depends on showing that each merge pass correctly merges the $R = \sqrt{M/B}/2$ runs into a single sorted run.

Theorem 1 *Each stage of merging in the Greed Sort algorithm correctly merges R runs.*

Theorem 2 below shows that each record in the approximately merged output list formed from the R runs is at most $L = RDB$ locations from its correct sorted location. The columnsorts are done on successive overlapping subfiles of size $2L$ to complete the sorting as shown in Theorem 3. This proves Theorem 1 (conditionally, of course, on proving Theorems 2 and 3).

Theorem 2 *In the approximately merged output list formed from R runs, each record is at most RDB locations from its correct sorted location.*

This theorem is proved using Lemmas 1–3. First we start with a definition.

Definition 1 A sequence is called *L-regressive* if for any two elements $x < y$, y does not precede x by more than L elements in the sequence.

Lemma 1 *For any two records $x < y$ written to disk j in the approximately merged output list, x will be located at most $R - 1$ blocks later than y on disk j.*

Proof: Let record y be written to disk j of the output list at step t. (See Figure 2.4.) (By step t, we mean the tth time DB records are written to the output list being formed.) We claim that (a) there is no run i such that $biggest[i,j] < y$, and (b) there is at least one run i such that $smallest[i,j] \geq y$. Claim (a) is true since every record written to the output has a value no bigger than the block that was chosen as having the smallest maximum. To show claim (b), we consider the two cases: (1) step t reads in two distinct blocks from disk j, and (2) step t reads in only one block from disk j. In the former case, the run that gets records put back onto it has no record on disk j that is less than y by construction since only the smallest B records were written to the output list; in the latter case the run from which the block was read has no records on disk j that are less than y since it was a sorted run. So in either case, there is a run whose minimum is at least y. From claim (a), no run has more than one block containing records less than y. We can now invoke claim (b) to show that any record

Algorithm 1 [Greed_Sort]

{ Create the initial runs }
repeat N/M times
read the next M records into primary memory, DB at a time
sort the M records internally
write the M records back onto disk, DB at a time

repeat until only 1 run is left
{ Merge $R = \sqrt{M/B}/2$ runs at a time }
repeat until all runs have been processed
pick the next $R = \sqrt{M/B}/2$ runs to process
for $i := 1$ **to** R **do**
read in parallel the first D blocks of run i into buffer $b1$
for $j := 1$ **to** D **do**
$mark[i, j] := 1$
$biggest[i, j] := \max(b1[j])$
$smallest[i, j] := \min(b1[j])$

{ Do an approximate merge of the R runs }
repeat until all records of the R runs have been processed
for $j := 1$ **to** D **do**
{ Determine the best one or two blocks to read from each disk }
$bestrun1[j] := i$ such that $biggest[i, j]$ is a minimum
$bestrun2[j] := i$ such that $smallest[i, j]$ is a minimum
for $j := 1$ **to** D **do in parallel**
{ Note: $b1$ and $b2$ are buffers }
read block $mark[bestrun1[j], j]$ of run $bestrun1[j]$ from disk j into $b1[j]$
if $bestrun1[j] \neq bestrun2[j]$ **then**
read block $mark[bestrun2[j], j]$ of run $bestrun2[j]$ from disk j into $b2[j]$
merge $b1[j]$ and $b2[j]$ in place internally
write $b2[j]$ to block $mark[bestrun2[j], j]$ of run $bestrun2[j]$ on disk j
write $b1[j]$ to disk j in the output list
read block $mark[bestrun1[j], j] + 1$ of run $bestrun1[j]$ from disk j into $b1[j]$
for $j := 1$ **to** D **do**
$mark[bestrun1[j], j] := mark[bestrun1[j], j] + 1$
$biggest[bestrun1[j], j] := \max(b1[j])$
$smallest[bestrun1[j], j] := \min(b1[j])$
if $bestrun1[j] \neq bestrun2[j]$ **then**
$smallest[bestrun2[j], j] := \min(b2[j])$

{ Do a restorative pass to turn the approximate merge into a complete merge }
$L := D\sqrt{MB}/2$
$T :=$ total # of records in the R runs
for $n := 0$ **to** $T/L - 2$ **do**
use columnsort to sort records $nL + 1, nL + 2, \ldots, nL + 2L$ of the run

Figure 2.7: All the records output on disk j at this step are guaranteed to have keys ≤ 7 (in fact the largest key will be 4). Any disk prior to disk j has at most one block per run containing elements less than any key output on disk j, all of which will be written no more than $R-1$ blocks after this one. Any disk after disk j has already output all keys less than those written on disk j. Thus, a smaller key can follow a larger one by at most RDB locations in the approximately merged output list.

smaller than y will be written to the output list within the next $R-1$ blocks. The reason is that any block containing a value less than y will have a smaller minimum than the block whose minimum is at least y. Each of the $R-1$ steps after step t is guaranteed to reduce the number of blocks containing values less than y by at least one. Since at most $R-1$ blocks contained values less than y (combining claims (a) and (b)), all values less than y will be written to the output list in the $R-1$ blocks following the one containing y. □

Lemma 2 *The approximately sorted output list is RDB-regressive.*

Proof: Let y be output on disk j. By Lemma 1, any block containing a value less than y on disk j will be output in the next $R-1$ time steps. Any block on a disk $i < j$ containing a value less than y will also either be output at the same time or within the next $R-1$ time steps as suggested by Figure 2.7, since no run has more than one block containing values less than y. Furthermore, any disk $i > j$ has already written all records with keys less than y. Since each "stripe" on the collection of disks contains DB records, the maximum regression is RDB. □

Figure 2.8: If an element y precedes its correct location by more than L, then there is not enough space (shaded region) for all those elements less than y without violating the L-regressive condition.

Lemma 3 *If a list is L-regressive, then every record is at most L locations from its correct sorted location.*

Proof: For simplicity, let all the records have unique keys. Assume that y is the jth smallest record and occurs at position i where $i < j - L$, as in Figure 2.8. We derive a contradiction by showing that there exists an $x < y$ that succeeds y by more than L records. There are $j-1$ records less than y. In order to meet the L-regressive condition, they must all occur in the range $1, \ldots, i + L$. There are $i + L - 1 < j - 1$ locations that can contain values less than y without violating the L-regressive condition. Thus, at least one record less than y must be out of the desired range. This same argument also shows that the jth record cannot be at any location $i > j + L$. When we allow duplicate keys, the argument gets a little more complicated, since we cannot talk about the jth record exactly. The fact that remains true, however, is that every record is within L of the closest place it can acceptably go. □

Lemmas 2 and 3 directly prove Theorem 2. Finally, we demonstrate that the final pass of columnsorts finishes the merge.

Theorem 3 *If every element in a list is within a distance of L of its sorted location, then a series of sorts of size $2L$, beginning at every Lth location, will suffice to complete the sort.*

Proof: Let there be N total records, so that we perform a total of $N/L - 2$ sorts of size $2L$. The proof proceeds by induction on the number of sorts performed. During step t we sort records $tL+1, tL+2, \ldots, tL+2L$, as in Figure 2.9. We show the following invariants at the beginning of step t by induction:

1. All the records $1, \ldots, tL$ are completely sorted
2. No record in $tL + 1, \ldots, N$ is more than $2L$ before or L after its final position.

The invariants are clearly met at the beginning since (1) refers to the null set and we have assumed something stronger than (2).

Now we can demonstrate that each step in the range $t = 1, \ldots, N/L-2$ preserves the invariants. To preserve invariant (1) we need to demonstrate that the sort moves records

Figure 2.9: At each step in sorting $2L$ elements, the part before the beginning of the sort is completely sorted and the part starting halfway through the sort is L-unsorted. The remaining portion has no element more than L after nor $2L$ before where it should be.

that belong in locations $tL+1, \ldots, tL+L$ to their final locations. By invariant (2), every record in this range must fall within the $2L$ records we are sorting: the record belonging at $tL+L$ can be off by at most L, placing it at $tL+2L$, which is within the sorting range. Thus, invariant (1) is preserved. Similarly, invariant (2) is preserved, since none of the records moved to the range $tL+L+1, \ldots, tL+2L$ belonged in the block $tL+1, \ldots, tL+L$ (by preservation of invariant (1)), so that they must be at most $2L$ before or L after their intended location.

Step $N/L-2$ by definition sorts the last $2L$ records, so that at its conclusion, the whole series is sorted. □

2.2.2 Analysis of the algorithm

We first show by a clustering technique that the amount of storage space required for the data structures is small enough. To provide for the data structures, we need to strengthen the condition imposed by [AgV] so that $DB \leq \frac{1}{2}\lfloor M - M^{1/2+\beta} \rfloor$, for $0 < \beta < 1/2$.

Theorem 4 *For $0 < \beta < 1/2$, the amount of primary memory space needed for the data structures of Greed Sort is $O(M^{1/2+\beta})$.*

Proof: The number of runs that we must merge in order to obtain optimal performance is $\sqrt{M/B}/2$. As we pointed out in Section 2.2, the candidate set requires a total of $D\sqrt{M/B}$ key fields to be kept in primary memory to decide what blocks to read next. At first glance, it seems if $D = \Omega(M)$ and $B = 1$, we will require $\Omega(M^{3/2})$ storage space in primary memory, which is clearly impossible. However, we can use a partial disk striping method throughout the course of the algorithm. Assume that D grows faster than M^{β}, where $0 < \beta < 1/2$. We can cluster our D disks into clusters of $D' = M^{\beta}$ clusters of $B' = BD/D'$ disks synchronized together. Each of the D' clusters acts like a logical disk with block size B'. Thus, the number of primary storage locations we need is at most

$$D'\sqrt{M/B'} \leq M^{\beta}\sqrt{M/B'} = O(M^{1/2+\beta}).$$

The expression for the number of I/Os remains the same, namely,

$$k\frac{N}{D'B'}\frac{\log\frac{N}{B'}}{\log\frac{M}{B'}} = O\left(\frac{N}{DB}\frac{\log\frac{N}{B}}{\log\frac{M}{B}}\right).$$

□

In order to demonstrate that Greed Sort has an optimal I/O bound, we need to analyze the I/O efficiency of the columnsort subroutine.

Theorem 5 *Columnsort sorts $N \leq D\sqrt{MB}$ records with $O(N/DB)$ parallel I/Os.*

Proof: First we show that columnsort produces a correctly sorted sequence when $N \leq D\sqrt{MB}$. We define the number of rows in the matrix to be $r = M$, so that we can sort each column internally. We have

$$N \leq D\sqrt{MB} \leq DB\sqrt{M} \leq \frac{M^{3/2}}{2} < \frac{M^{3/2}}{\sqrt{2}},$$

making use of our assumption that $2DB \leq M$. Thus, the number of columns in our application of columnsort is $c = N/r \leq \sqrt{M/2}$, meeting the columnsort requirement that $r \geq 2(c-1)^2$. This observation implies that columnsort works correctly.

Steps 1, 3, 5, 6, 7, and 8 can be done easily with $O(N/DB)$ I/Os. The transpose-like operation in Steps 2 and 4 can be done with $O(N/DB)$ I/Os by a variant of the $p \times q$ matrix transpose algorithm of [ViSb], for $p = M$ and $q = N/M \leq D\sqrt{B/M}$, which we omit for brevity. The resulting number of I/Os is

$$\begin{aligned} O\left(\frac{N}{DB}\frac{\log\min\left\{M, D\sqrt{B/M}, D\sqrt{MB}/B\right\}}{\log(M/B)}\right) &= O\left(\frac{N}{DB}\frac{\log\left(D\sqrt{M/B}\right)}{\log(M/B)}\right) \\ &= O\left(\frac{N}{DB}\right). \end{aligned}$$

□

The greedy merge reads each record at most three times (once for updating *biggest* and *smallest* and up to twice for merging) and writes each record at most twice, taking full advantage of parallel block transfer. The columnsort routine is called $2N/k$ times, each time using $O(k/DB)$ I/Os, for a value of k that differs from pass to pass. Thus the total number of parallel I/Os is $O(N/DB)$:

Corollary 1 *Each merging step requires $O(N/DB)$ parallel I/Os.*

We are now ready for our main theorem.

Theorem 6 *Greed Sort sorts $N \geq M$ records with $O(\frac{N}{DB}\log\frac{N}{B}/\log\frac{M}{B})$ parallel I/Os deterministically, which is optimal, as long as $2DB \leq \lfloor M - M^{1/2+\beta} \rfloor$.*

Proof: Theorem 1 showed the correctness of the algorithm. The fact that the algorithm can remain within the memory bounds was shown in Theorem 4. Finally, Corollary 1 shows that each pass takes $O(N/DB)$. Since we started with N/M runs and at each pass merged together $\sqrt{M/B}/2$ of them, the total I/O bound is

$$O\left(\frac{N}{DB}\left(1+\log_{\sqrt{M/B}/2}\frac{N}{M}\right)\right) = O\left(\frac{N}{DB}\frac{\log\frac{N}{B}}{\log\frac{M}{B}}\right),$$

which is optimal, by the lower bound of [AgV]. □

2.3 Conclusions

Greed sort is an optimal deterministic algorithm for external sorting applications that is easy to implement in practice. The advantages of greed sort are:

- It exhibits optimal I/O performance under realistic high-performance storage systems with multiple disks.
- It is deterministic with a worst-case guarantee on performance. Its performance can be substantially better than that of the commonly used technique of disk striping.
- It is simple and straightforward to implement.

Chapter 3

Sorting on Parallel Disks: Balance Sort

3.1 How to Balance

Distribution Sort is an algorithm that chooses a set of numbers that partition the elements to be sorted into ranges of values called *buckets*. Having the partition elements, we can then process the elements one at a time, placing each into the appropriate bucket, and sort each bucket. Finally, we concatenate the buckets in the right order to produce a completely sorted list. For example, if you needed to sort the checks from your bank statement, and the checks fall in the range 350–399, you might choose as your partition elements 360, 370, 380, and 390. You could then sort the checks into stacks (buckets). Once each stack is sorted, you then put the 350s first, followed by the 360s, etc., and the checks are fully sorted.

This chapter describes a new algorithm called Balance Sort that is based on Distribution Sort. In Balance Sort, after finding the partitioning elements, we read D records into memory in parallel, partition them, and write them back in parallel, then read, partition, and write the next D records, and so on, until the whole file has been partitioned into buckets. We want to minimize the number of parallel reads that we will need during the next level of recursion to re-read each bucket. The heart of Balance Sort is in balancing all of the buckets evenly among the disks. This section describes the Balance routine, while the next section describes how to use Balance to achieve the optimal number of I/Os. For the time being, we will assume that the block size B is 1. Nothing in this section is dependent upon that assumption, and we show in Section 3.2.4 how to accommodate arbitrary block sizes.

Let x_{bd} be the number of records written to disk d from bucket b; $X = (x_{bd})$ is called the *histogram matrix*. The number of parallel reads that are needed to read bucket b during the next level of recursion is then $r_b = \max_{\text{disks } d}\{x_{bd}\}$. So we want to minimize $\sum_{\text{buckets } b} r_b$. We process the file a track at a time (all of the disks are synchronized for reading) and use a greedy strategy that attempts to minimize the increase of this sum. We do this by computing a matrix Y, called the *skew matrix*,

Algorithm 2 [Simple_Balance(P)]

```
{ P contains the partition elements }
{ Initialize }
for b := 1 to S
    for d := 1 to D
        x_bd := 0
        y_bd := 0

{ Process the file }
for t := 1 to ⌈N/D⌉
    read D records in parallel into array T
    { Compute the matching matrix Z }
    for i := 1 to D
        τ[i] := bucket(T[i], P)
        for d := 1 to D
            z_{i,d} := y_{τ[i],d}
    find a permutation π of T that is a min-cost matching of matching matrix Z
    for d := 1 to D
        increment x_{τ[π(d)],d}
        increment y_{τ[π(d)],d}
    decrement any row of Y that has no zeros
    write out T in the permuted order
```

that gives a measure of how unbalanced each bucket is. We define $y_{bd} = x_{bd} - x_b^{\min}$, where $x_b^{\min} = \min_{\text{disks } d}\{x_{bd}\}$. Intuitively, the skew matrix element y_{bd} indicates how many records of bucket b will be on disk d after all the records in the bucket have been read on some other disk. We call y_{bd} the *skew* of bucket b on disk d. We attempt to minimize the elements of the skew matrix by computing a min-cost bipartite matching between blocks and disks. In particular, a block from bucket b will have cost y_{bd} to be assigned to disk d. Consider Algorithm 2.

In this algorithm, we assume that bucket(a, P) returns the number of the bucket to which a belongs based on the set of partitioning elements P. Note that this algorithm is an on-line algorithm in that it makes its decisions on where to place things based only on the records it has already seen (as codified in the arrays X and Y). The complexity of finding a permutation for the min-cost matching is $O(D^3)$, but for many architectures, this computation will be able to be done in parallel with I/O operations, and so will not result in any slowdown.

There is an obvious fact about the disk sums in the histogram matrix.

Lemma 4 *The disk sums in the histogram matrix, $x_d = \sum_b x_{bd}$, are all equal.*

Proof: By definition, $\sum_b x_{bd}$ is the number of tracks written on disk d. Since we are

reading and writing whole tracks at a time, the sum must be constant over all disks. □

It is not necessary from an algorithmic standpoint to have a skew matrix Y, since the min-cost matching will give exactly the same results if it consists of rows from the histogram matrix X. However, the skew matrix gives some insight into how unbalanced the buckets are. Intuitively, the skew matrix tells what will be left over after all the buckets have been read with all the full parallelism available to them. Assigning a bucket b to a disk d for which y_{bd} is zero is a good thing; it will never increase the number of reads required to process the entire bucket beyond the minimum needed. Alternatively, assigning a bucket b to a disk d for which y_{bd} is larger than $y_{bd'}$ for any other disk d' will always increase the total number of reads needed to process the bucket beyond what is strictly needed. In doing a min-cost matching, we are attempting to balance all of the buckets represented in the track simultaneously. The skew matrix has some useful properties, as evidenced by the following simple lemmas.

Lemma 5 *Every bucket b contains at least one zero in the skew matrix.*

Proof: By definition of the skew matrix, those disks d for which $x_{bd} = \min_d \{x_{bd}\}$ will have $y_{bd} = 0$. □

Lemma 6 $y_{bd} \geq 0$ *for all b and d.*

Proof: By definition of constructing the skew matrix, we subtracted off the minimum value for each bucket; hence all the other values in the bucket were at least as large as the one subtracted. □

Lemma 7 *The disk sums in the skew matrix, $y_d = \sum_b y_{bd}$, are all equal.*

Proof: This fact is a simple consequence of Lemma 4 and the fact that forming the skew matrix from the histogram matrix subtracts the same number from each disk for each bucket. □

We need a little information about how the skew matrix evolves as a function of the number of tracks processed.

Definition 2 Let Y be a skew matrix and let Y' be the same skew matrix after a track has been processed using the min-cost matching to assign buckets to disks and after any rows that have no zeroes have been decremented. If $y'_{bd} > y_{bd}$ then we say that bucket b has been *promoted* on disk d.

Lemma 8 *At most one bucket promotes on any given disk on any track. Furthermore, that bucket can increase its skew by at most one on the disk.*

Proof: Since exactly one bucket has its value in the histogram matrix incremented on the given disk, it is the only one that could have its value increase in the skew matrix on that disk. Since the value in the histogram matrix was incremented by one, one is also the maximum possible increase in its skew. □

The following lemma about changes to the skew matrix is pivotal in proving the optimal behavior of our algorithm.

Lemma 9 *Assume that the largest element of skew matrix Y is* max *and that some min-cost matching for a matching matrix composed of rows of Y results in a skew matrix Y' for which several buckets have a skew of* max $+1$*. Then all the buckets that reached the larger skew must have had a skew of* max *in Y on every disk for which the skew increased to* max $+1$*.*

Proof: Assume that bucket b reaches a skew of max $+1$ on disk d and bucket b' reaches a skew of max $+1$ on disk d'. Then in the partial matching, we have

	d	d'
b	$\underline{\max}$	i
b'	j	$\underline{\max}$

where underlines indicate that the elements are part of a min-cost matching. Hence, it must be the case that $i + j \geq 2\,\max$. Since no element of Y is greater than max, then it must be the case that $i = j = \max$. Since this fact is true for every pair of buckets that achieve a new maximum skew, the result follows. □

The upshot of Lemma 9 is that if several buckets are promoted to a new maximum skew value, there must have been a "block of maxes" in the matching matrix. That is, the matching matrix must in part (up to permutations of the rows and columns) have looked like this:

$$\begin{array}{cccc} \underline{\max} & \max & \ldots & \max \\ \max & \underline{\max} & \ldots & \max \\ \vdots & \vdots & \ddots & \vdots \\ \max & \max & \ldots & \underline{\max} \end{array}$$

The difficulty with Algorithm 2 is that, although there is much empirical evidence to suggest that it is always within a factor of 2 of the optimal time for reading all the buckets back in, it has been difficult to prove this conjecture in the general cases. Intuitively, the algorithm always tries to place the buckets on the disk that conflicts least with what has been previously placed. If it could be shown that the maximum skew grows sufficiently slowly (for example, that it takes at least a quadratic number of tracks to attain any given skew value) or that the maximum skew is bounded linearly by the number of buckets available, then it would be possible to show that Algorithm 2 could obtain optimal results for balancing.

Our alternative and provably optimal algorithm makes use of two key ideas:

1. We rebalance occasionally if the min-cost operation skews the balance too badly.

2. We only need to use some (preferably large) constant fraction of the disks efficiently.

The point behind rebalancing is that when some bucket b gets too skewed, that is, when $x_{bd} - x_{bd'} > c$ for some disks d and d', where $c > 0$ is some skew threshold, we can read a block from bucket b on disk d and write it to disk d'. In fact, any number of buckets (up to $\lfloor D/2 \rfloor$) can be rebalanced in a single parallel I/O operation, as long as the swaps all take place on distinct disks.

Saying that we need to use only some constant fraction α of the disks efficiently for any given bucket means that the number of tracks needed to read that bucket will expand by a factor of only about $1/\alpha$ above the optimal. Our algorithm will use $\alpha = \frac{1}{2}$. Our algorithm is given in Algorithm 3. In addition to keeping a histogram matrix X, the algorithm keeps an auxiliary matrix A to ensure that after processing each track, no bucket is too far out of balance. Specifically, if m_b is the median number of blocks that bucket b has on all the disks (m_b is the median of $x_{b1}, \ldots, x_{bD}$),[1] we define $a_{bd} := \max\{0, x_{bd} - m_b\}$. Invariant 1 about A below will guarantee that $X_{bd} \leq m_b + 1$ for all $1 \leq d \leq D$.

The differences between Algorithm 2 and Algorithm 3 are underlined. This algorithm requires two subroutines to complete its work: ComputeAux and Rebalance, given in Algorithms 4 and 5, respectively. We define these routines so as to maintain two invariants on the matrix A at step (1) in the algorithm:

1. A is a binary matrix; that is, all of its elements are either 0 or 1.

2. Every row of A has at least $\lceil D/2 \rceil$ 0s in it (or equivalently, every row has no more than $\lfloor D/2 \rfloor$ 1s in it).

It is easy to see that Invariant 2 is maintained, since the smallest $\lceil D/2 \rceil$ elements all become zero, by our definition of the median.

To show Invariant 1, it is necessary to understand a bit more about how the min-cost matching affects the auxiliary matrix and how effective the Rebalance subroutine is. We use induction to show that the variant is maintained. Auxiliary matrix A is clearly binary initially (it is all 0s).

To show that the invariant is maintained, we first need a preliminary lemma. Next we will show a couple of lemmas that assume the invariant holds at line (1) of Algorithm 3 and show what happens to A at later points in the iteration. After this, we show some lemmas needed to demonstrate that Rebalance is able to remove any 2s introduced as a result of the min-cost matching. Finally, Theorem 7 establishes the invariant.

Lemma 10 *The only entries of A that can be larger at line (3) of Algorithm 3 than they were at line (1) of the same iteration are those matched in the min-cost matching. Furthermore, any element of A that increases can increase by only one.*

Proof: Certainly, any elements of X not involved in the min-cost matching are the same at line (3) as they were at line (1), so the crux of the proof is to show that the

[1] We use the convention that the median is always the $\lceil D/2 \rceil$th smallest element, rather than the convention in statistics that it is the average of the two middle elements when D is even.

Algorithm 3 [Balance(P)]

```
    { P contains the partition elements }
    { Initialize }
    for b := 1 to S
        for d := 1 to D
            x_bd := 0
            a_bd := 0

    { Process the file }
    for t := 1 to ⌈N/D⌉
        read D records in parallel into array T
        { Compute the matching matrix Z }
        for i := 1 to D
            τ[i] := bucket(T[i], P)
            for d := 1 to D
(1)             z_{i,d} := a_{τ[i],d}
        find a permutation π of T that is a min-cost matching with matching matrix Z
        write out T in the permuted order
        for d := 1 to D
            increment x_{τ[π(d)],d}
(2)     A := ComputeAux(X)
(3)     if there is a 2 in A
(4)         Rebalance(A, X)
            A := ComputeAux(X)
```

Algorithm 4 [ComputeAux(X) returns A]

```
    for b := 1 to S
        m := median of x_b1, ..., x_bD (the ⌈D/2⌉th smallest element)
        for d := 1 to D
            a_bd := max(0, x_bd − m)
```

call to ComputeAux in line (2) maintains this property of A. But since the median for any bucket can only go up as a result of the matching, the difference between any element not matched and the median can only go down. The elements of A involved in the match increase by only one, and again, since the median can only increase, the difference between them and the median can go up by at most one. □

Now we can show that lemmas similar to Lemmas 8 and 9 for skew matrices will

Algorithm 5 [Rebalance(A, X)]

```
    { Create a bipartite matching problem }
    U := {d | a_bd = 2 for some b ∈ 1, ..., S}
    V := {1, ..., D} − U
    E := ∅
    for i := 1 to |U|
        d := U_i
(1)     b := (unique) bucket such that a_bd = 2
        for j := 1 to |V|
            d' := V_j
            if a_bd' = 0
                E := E ∪ (U_i, V_j)
    { We will swap a pair of blocks for every edge in the following match }
(2) Find a maximum bipartite matching on the graph G = (U ∪ V, E)
    { The array R will tell us what buckets to read from each disk }
    { The array W will tell us on what disk to write the block }
    for d := 1 to D
        r_d := 0
        w_d := 0
    for each match (U_i, V_j)
        d := U_i
        r_d := b
        w_d := V_j
        { Update X to reflect the swap }
        x_bd := x_bd − 1
        x_bd' := x_bd' + 1
    Read the last block of bucket r_d on disk d if r_d ≠ 0
    Write the block read from disk d onto disk w_d if r_d ≠ 0
```

hold for the auxiliary matrix.

Lemma 11 *Assuming that A is binary at line (1) of Algorithm 3, at most one bucket will have a 2 on any given disk at line (3) of the algorithm during the same iteration. No buckets will have any elements larger than 2.*

Proof: By Lemma 10, only those elements involved in a match could have increased and even then, only by one. Since A was binary prior to the match, the largest element is no greater than 2 at line (3). Furthermore, by definition of the match, exactly one element of X is incremented on each disk; that element is the only one that could become a 2 on that disk. □

Lemma 12 *Assume that A is binary at line (1) of Algorithm 3 and let $\mathcal{D}$ be the set of disks having a 2 in the auxiliary matrix at line (3) during some iteration. Then every bucket that has a 2 in must have had a 1 on every disk in $\mathcal{D}$ back in line (1) of the algorithm during this iteration.*

Proof: The argument used in the proof of Lemma 9 works here as well, except that in this case, max = 1 and some of the 2s may have reverted to 1s during the call to ComputeAux at line (2) of the algorithm, if the median for that bucket happened to increase. However, the "block of 1s" is still a necessary (though not sufficient) condition for getting 2s on several buckets. □

Every bucket in A at line (3) of Algorithm 3 has at least $\lceil D/2 \rceil$ zeroes in it. The call to ComputeAux at line (2) guarantees this fact. We need one more fact about A before we can understand the Rebalance routine.

Lemma 13 *Assuming that A is binary at line (1) of Algorithm 3, at most $\lfloor D/2 \rfloor$ disks will have 2s in A at line (3) on the same iteration.*

Proof: By Lemma 12, a necessary condition for the introduction of 2s on k different disks is to have a "block of 1's" in the matching matrix. But each bucket has at least $\lceil D/2 \rceil$ zeroes, which means that the maximum width of the block of 1s is $\lfloor D/2 \rfloor$. □

Now that we know what the matrix A looks like at the call to Rebalance at line (4) of Algorithm 3, we need to show some facts about the Rebalance subroutine. First, note that Lemma 11 establishes that the bucket at line (1) of Algorithm 5 is unique. Also, note that the following lemmas are assuming that the induction hypothesis holds (i.e., that A was binary back on line (1) of Algorithm 3), even though we do not state that assumption as part of the lemmas.

Lemma 14 *The maximum matching at line (2) of Algorithm 5 will include all vertices in U.*

Proof: By Lemma 13, $|U| \leq \lfloor D/2 \rfloor$. Also, each vertex in U has at least $\lceil D/2 \rceil$ edges coming out of it. Each edge that is chosen for the match can invalidate at most one possible edge for every other vertex of U. Since the minimum number of edges coming out of any vertex in U exceeds $|U|$, then the simple greedy algorithm of just choosing the first edge for the first vertex of U, the first remaining edge for the second vertex of U, and so on, will always result in a match that includes all the vertices of U. □

At long last, we are ready to prove that A is always binary at line (1) of Algorithm 3.

Theorem 7 *Invariant 1 (A is a binary matrix) is maintained at line (1) of Algorithm 3.*

Proof: The proof proceeds by induction on the number of tracks processed. Before any tracks have been processed, it is trivially true, since all the elements of A are zero.

Assume that the invariant holds at the beginning of an iteration. We now show that the invariant will hold at the end of the iteration to be carried back to the beginning

of the next iteration. If there is no 2 in A at line (3) of Algorithm 3, then the invariant is clearly maintained. Assume that $a_{bd} = 2$ at line (3) of Algorithm 3. We know from Lemma 14 that the vertex corresponding to disk d in U will be matched to some disk d' for which $a_{bd'} = 0$. The rebalancing algorithm will read the last block of bucket b on disk d and write it to disk d'. This operation is guaranteed to make it so that the call to ComputeAux in line (5) of Algorithm 3 will result in $a_{bd} \leq 1$ and $a_{bd'} \leq 1$, since the median element could not have existed on disk d, so there is no chance of lowering the median. By Lemma 14, we can accommodate all of the 2s that appeared in A simultaneously, and since the matching problem had only one vertex per disk, they can all be balanced in the single parallel read/write operation. Thus, A is binary at the end of the loop and does not change before line (1) at the next iteration (or at the end of the algorithm, if this was the last iteration). □

We now have a theorem that tells how well each bucket is balanced.

Theorem 8 *Any bucket b will take no more than a factor of 3 above the optimal number of tracks to read.*

Proof: Let m_b denote the median element in bucket b. Then, by the Invariant 1, which is also maintained at termination of the algorithm, the number of tracks needed to read bucket b is no more than $m_b + 1$. But we also know that the bucket has at least $\lceil D/2 \rceil m_b$ elements in it, so it requires at least

$$\left\lceil \frac{\lceil D/2 \rceil m_b}{D} \right\rceil \geq \left\lceil \frac{m_b}{2} \right\rceil$$

tracks to read it. Thus, we are a factor of at most

$$\frac{m_b + 1}{\lceil m_b/2 \rceil} \leq 3$$

from the optimal. (The expression could be as large as 3 if D is even and $m_b = 2$.) □

It should be noticed that the Algorithm 3 assumes that it reads D blocks on every track, whereas the previous phase does not guarantee that every track up to the last is fully utilized. Adjusting the algorithm for partial tracks is simple: if no block is available to be read on some disk d, we pretend that we read from a dummy bucket 0. Bucket 0 will be have the characteristic that $x_{0d} = 0$ for all $1 \leq d \leq D$, so in particular, that row of the matching matrix will be all zeroes. The part of the algorithm that increments the values of X should ignore bucket 0.

Finally, we can bound the total number of parallel I/Os needed for balancing N records into S buckets.

Theorem 9 *A phase of Balance requires no more than $6N/DB$ parallel reads and a like number of parallel writes.*

Proof: We know from Theorem 8 that the N records we are processing occupy no more than $3N/DB$ tracks. Each of these tracks can give rise to at most 2 parallel reads and writes: one for the original assignment to disks according to the min-cost matching, and at most one parallel read and write in rebalancing. □

Algorithm 6 [Balance_Sort(N, T)]

```
    if N ≤ M
(1)     Read the whole bucket into memory
        Sort internally
(2)     Write the bucket out again
    else
        S := min(2√(M/B), 2N/M)
(3)     P := ComputePartitionElements(S)
        { The T array says what track to start with on each disk }
(4)     L := Balance(T)
(5)     Write L
        for b := 1 to S
(6)         T := Read bth row of L
            N_b := number of elements in bucket b
(7)         Balance_Sort(N_b, T)
(8)         Append sorted bucket to output area
```

The other modification that we need to make to the balancing algorithm before it can be used as part of a sorting routine is that it needs some provision for keeping track of where each bucket is located so that the next pass will know how to read it in. This is easily accomplished by keeping another $D \times S$ matrix L such that l_{bd} is the last track with a block from bucket b on disk d. With a constant amount of overhead in each block, we can store a pointer to the previous block on the same disk (or zero if this is the first block). Thus, starting from the final configuration of L, we can read the bucket backwards.

3.2 The Sorting Algorithm

This section describes how to fit the balancing algorithm into an optimal algorithm for sorting, which we call Balance Sort. It assumes that the Balance routine has been modified to return the starting (ending) blocks of each bucket on each disk, as described in the previous section. Algorithm 6 gives the actual algorithm, assuming for the time being that $B = 1$. Subsection 3.2.4 describes how to handle the case where $B > 1$.

The correctness of the algorithm is easy to establish, since the bottom level of recursion by definition produces a sorted list and each level thereafter concatenates sorted lists in the right order. In this algorithm, we will let $S = \min(2\sqrt{M/B}, 2N/M)$ in order to produce optimal performance, as shown in Subsection 3.2.2.

Theorem 10 *Algorithm 6 correctly sorts N elements using*

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

Algorithm 7 [Select(S_0, k)]

$t := \lceil |S_0|/M \rceil$
for $i := 1$ to t
    Read M elements in parallel ($\lceil M/DB \rceil$ tracks; the last may be partial)
    Sort internally
    $R_i :=$ the median element
if $t = 1$
    return(kth element)
$s :=$ the median of $R_1, \ldots, R_t$ using algorithm from [BFP]
Partition into two sets S_1 and S_2 such that $S_1 < s$ and $S_2 \geq s$
$k_1 := |S_1|$
$k_2 := |S_2|$
if $k \leq k_1$
    return(Select(S_1, k))
else
    return(Select$(S_2, k - k_1)$)

parallel I/O's, as long as $4DB \leq M - M^{1/2+\beta}$ *for some* $0 < \beta < 1/2$.

The next few subsections establish the proof of this theorem.

3.2.1 Finding the partition elements

The following procedure for finding the partition elements was presented in [ViSb], and is reproduced here for convenience in Algorithm 8. The algorithm is deterministic and guarantees to find $S - 1$ partition elements such that, for any bucket b, the number of elements in that bucket N_b obeys the constraint

$$\frac{N}{2S} \leq N_b \leq \frac{3N}{2S}.$$

The algorithm was analyzed in [ViSb] and shown to take $O(N/DB)$ parallel I/O operations which, as we will see in the analysis, is sufficient for achieving optimal performance.

The procedure for finding the partitioning elements uses as a subroutine Algorithm 7, which computes the kth smallest of n elements in $O(n/DB)$ I/Os. It does this by recursively subdividing about an element that is guaranteed to be close to the median.

3.2.2 Analysis of sorting algorithm

We have numbered each of the steps of Algorithm 6 that could cause an I/O operation. Let $T(N)$ denote the amount of time needed for sorting N records.

Algorithm 8 [ComputePartitionElts(S) returns P]

for each memoryload of records $\mathcal{M}_i (1 \leq i \leq N/M)$
  Read $\mathcal{M}_i$ into memory
  Sort internally
  Construct $\mathcal{M}'_i$ to consist of every $(S-1)$/4th record
  Write $\mathcal{M}'_i$
$\mathcal{M}' := \mathcal{M}'_1 + \cdots + \mathcal{M}'_{N/M}$
for $j := 1$ to $S-1$
  $b_j :=$ Select$(\mathcal{M}', 4Nj/(S-1)^2)$
return$(\{b_j\})$

Steps (1) and (2) occur only at the innermost level of recursion. From Theorem 8, the bucket will take no more than $3N/DB$ reads and writes.

Step (3), as mentioned, has been shown by [ViSb] to require only $O(N/DB)$ I/Os, regardless of whether $S = 2\sqrt{M/B}$ or $S = 2N/M$. Step (4), as shown in Theorem 9, requires no more than $6N/DB$ I/Os. Step (5) requires writing a set of cardinality DS, which can be done in S/B parallel writes. Step (6) can be done in a single parallel read, since the cardinality of the set is only D, for a total of S reads. Step (7) takes

$$\sum_{1 \leq b \leq S} T(N_b),$$

where N_b is the number of elements in bucket b. Let $N_b = \frac{N}{S} + \Delta_b$, where $|\Delta_b| \leq \frac{N}{2S}$ and $\sum_{1 \leq b \leq S} \Delta_b = 0$. Finally, step (8) can read the bucket in no more than $3N_b/DB$ I/Os and write it in no more than N_b/DB I/Os, for a total (over all buckets) of $4N/DB$ I/Os. Thus, we obtain the recurrence

$$T(N) = \begin{cases} \displaystyle\sum_{1 \leq b \leq S} T\left(\frac{N}{S} + \Delta_b\right) + O\left(\frac{N}{DB}\right) + O(S) & \text{if } N > M \\ \dfrac{3N}{DB} & \text{if } N \leq M \end{cases}$$

Furthermore, we can show that

$$S = \begin{cases} 2\sqrt{\dfrac{M}{B}} & \text{if } N \geq \dfrac{M^{3/2}}{\sqrt{B}} \\ \dfrac{2N}{M} & \text{if } N \leq \dfrac{M^{3/2}}{\sqrt{B}} \end{cases}$$

First, consider the case $N \leq M^{3/2}/\sqrt{B}$. This results in a recurrence of

$$T(N) = \sum_{1 \leq b \leq S} T\left(\frac{M}{2} + \Delta_b\right) + O\left(\frac{N}{DB}\right) + O\left(\frac{N}{M}\right),$$

where $|\Delta_b| \leq M/4$. In particular, each bucket has size no more than $3M/4$, and so we can sort it internally. Thus,

$$T(N) = \frac{1}{DB} \sum_{1 \leq b \leq S} \left(\frac{M}{2} + \Delta_b \right) + O\left(\frac{N}{DB} \right) + O\left(\frac{N}{M} \right) = O\left(\frac{N}{DB} \right),$$

since $M > DB$.

When $N \geq M^{3/2}/\sqrt{B}$, it is straightforward, though tedious, to demonstrate by substitution and approximating $\log(1+\epsilon)$ by the first few terms of its infinite expansion that a solution to the recurrence is

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

All of this analysis leads to the following theorem.

Theorem 11 *The algorithm given is optimal for sorting in the parallel disk model.*

Proof: The preceding analysis shows that the algorithm achieves the same time bound as was shown in [AgV] to be a lower bound for sorting with parallel block transfer. □

3.2.3 Memory usage of the sorting algorithm

One aspect of the algorithm that we still need to show is that we do not exceed the memory requirements of the computer. The algorithm maintains three $S \times D$ arrays: L, A, and X.

Theorem 12 *For $0 < \beta < 1/2$, the amount of primary memory space needed for the data structures of the sorting algorithm is $O(M^{1/2+\beta})$.*

Proof: In either range of S, it is easy to show that the space required is no more than $2\sqrt{M/B}D$. At first glance, it seems if $D = \Omega(M)$ and $B = 1$ that we will require $\Omega(M^{3/2})$ storage space in primary memory, which is clearly impossible. However, we can use a partial disk striping method throughout the course of the algorithm, as described earlier. Assume that D grows faster than M^β, where $0 < \beta < 1/2$. We can cluster our D disks into clusters of $D' = M^\beta$ clusters of $B' = BD/D'$ disks synchronized together. Each of the D' clusters acts like a logical disk with block size B'. Thus, the number of primary storage locations we need is at most

$$D'\sqrt{M/B'} \leq M^\beta \sqrt{M/B'} = O(M^{1/2+\beta}).$$

The expression for the number of I/Os remains the same, namely,

$$k \frac{N}{D'B'} \frac{\log \frac{N}{B'}}{\log \frac{M}{B'}} = O\left(\frac{N}{DB} \frac{\log \frac{N}{B}}{\log \frac{M}{B}} \right).$$

□

Algorithm 9 [Balance_With_Blocks(P)]

```
{ Initialize }
for b := 1 to S
    for d := 1 to D
        x_bd := 0
        a_bd := 0

{ Process the file }
for t := 1 to ⌈N/DB⌉
    read D blocks in parallel into input array
    Apportion records into blocks containing all the same bucket
    if there are at least D full blocks or t = ⌈N/D⌉
        if there are at least D full blocks
            Define array T to comprise D full blocks
        else
            Define array T to be any D blocks that need writing
        { Compute the matching matrix Z }
        for i := 1 to D
            τ[i] := bucket(T[i])
            for d := 1 to D
                z_{i,d} := a_{τ[i],d}
        find a permutation π of T that is a min-cost matching with matching matrix Z
        write out T in the permuted order
        for d := 1 to D
            increment x_{τ[π(d)],d}
        A := ComputeAux(X)
        if there is a 2 in A
            Rebalance(A, X)
            A := ComputeAux(X)
```

3.2.4 Dealing with larger blocks

Although we gave the analysis for general block sizes, there has until now been no description of how the algorithm collects the records into blocks. The modifications to do this take place in the Balance routine (Algorithm 3). Algorithm 9 shows the modified Balance routine. The number of I/O's performed is not affected; the changes only affect the amount of memory needed.

At the point in the loop where D records are read in parallel into array T, we read whole blocks, and repack into blocks containing only records that belong in the same bucket. Whenever we have D full blocks, we go into the normal process of min-cost matching and rebalancing if necessary. At the end, we also use the min-cost and

rebalancing method to output the partially full blocks.

There can be no more than S partially filled blocks, and consequently the amount of space needed before we find D completely filled blocks is never more than $(S+D)B$. We thus can show the following theorem.

Theorem 13 *The amount of space needed for buffering and collecting into buckets will never exceed M as long as $M \geq 4DB$.*

Proof: As shown above, the amount of space needed for collecting into blocks is no more than $(S+D)B$. In addition, we need space DB for the input buffer. Thus, the total amount of storage needed for buffering and collecting into blocks will never exceed $SB+2DB$. But $S \leq \sqrt{M/B}$ throughout the algorithm and by hypothesis, $M \geq 4DB$, so the amount of space needed is no more than

$$\begin{aligned} SB + 2DB &\leq \sqrt{MB} + 2DB \\ &\leq \sqrt{MDB} + 2DB \\ &\leq \sqrt{M\frac{M}{4}} + 2\frac{M}{4} \\ &= M \end{aligned}$$

□

3.3 Conclusions

In this chapter, we have described a new algorithm for sorting on parallel disk subsystems called Balance Sort. This algorithm has the following advantages over previously known deterministic algorithms:

1. The new algorithm is also applicable to parallel memory hierarchies. The difficulty with Greed Sort is that it was based on merge sort, and in many cases of hierarchical memories, there is no known way to make merge sort work optimally on even a single memory hierarchy. This problem would have had to have been solved before Greed Sort could have had any applicability to these parallel memory hierarchies.

2. The hidden constants in the big-oh notation are small. The problem with the Greed Sort algorithm is that it uses Columnsort [Lei] as a subroutine, which introduces at least an additional factor of 4 into the constant of proportionality.

3. The algorithm can operate using only striped write operations. Some parallel disk subsystems, such as Thinking Machine's Data Vault, impose this requirement so that they can maintain error checking and redundancy information to give acceptable levels of reliability when dealing with many disks, each of which has an independent probability of failure. It is unclear how to adapt Greed Sort to use only striped writes, again because of the embedded call to Columnsort.

While Balance Sort is applicable to parallel memory hierarchies, there is still a significant hurdle to be overcome when the hierarchies have an effectively logarithmic cost function. In the models that we consider, there is a requirement that the internal processing be able to be done in $\log P$ parallel time in order to achieve optimal performance, where P CPUs are connected together, each with its own memory hierarchy. It should be noted that the min-cost matching in the Balance routine is operating on a 0-1 matrix, and that therefore the full power of min-cost matching is not required; indeed, maximum matching on the complement of the matching matrix will do the job. Furthermore, the proof that the "block of 1s" is present does not even depend upon maximum matching, but only that the matching is a local maximum. Likewise, it is not hard to show that any maximal matching in the Rebalance routine is also a maximum matching. Thus, we can use maximal matching in both the Balance and Rebalance routines. Unfortunately, the best known deterministic parallel time for maximal matching is $O(\log^3 P)$ [IsS], which is not good enough to get optimal performance on the parallel memory hierarchies with effective logarithmic cost functions. The adjustments necessary to obtain optimal performance are the subject of the next chapter.

An interesting sorting algorithm in the context of fixed interconnection networks was proposed recently by Cypher and Plaxton [CyP]. The algorithm runs in $O(\log n(\log\log n)^2)$ time on an n-processor hypercube, shuffle-exchange, or cube-connected cycles network. We are currently exploring whether their techniques would be applicable to our I/O model, and *vice versa* whether our balancing techniques would yield good sorting algorithms for networks.

Chapter 4

Sorting in Parallel Memory Hierarchies

Section 4.1 describes the memory models considered in this chapter. Section 4.2 lists our main results. In Section 4.3, we give an algorithm that is optimal for all the parallel multi-level hierarchies. In Section 4.4, we show how to alter the algorithm to deal with parallelism of CPUs in the parallel disk model. Finally, we summarize our results in Section 4.5.

4.1 Memory Models

4.1.1 Parallel disk models

The first memory hierarchy we consider is the two-level memory hierarchy, known as the disk model. In this case, there is a P-processor CRCW (Concurrent Read, Concurrent Write) PRAM with some limited amount of global internal memory capable of storing M records, and a large external memory, in the form of D disks. The N records of the file reside on the disk and are grouped into blocks of size B.

$$\begin{aligned} N &= \text{\# records in the file} \\ M &= \text{\# records that can fit in internal memory} \\ P &= \text{\# internal processors} \\ D &= \text{\# disks} \\ B &= \text{\# records per block} \end{aligned}$$

We require that $1 \leq P \leq M$ and $2BD \leq M$.

In a single I/O, each of the D disks can simultaneously transfer a block of B records. Our main measure of performance is the number of I/Os, but we will also consider the internal processing time. The difficulty in designing optimal algorithms is dealing with the partitioning of secondary storage into separate disks.

Our disk model is a generalization of the Vitter-Shriver model in which we allow more than one processor. Figure 4.1a shows the uniprocessor ($P = 1$) multiple disk

Figure 4.1: (a) The parallel disk model. Each of the D disks can simultaneously transfer B records to and from internal memory in a single I/O. The internal memory can store $M \geq DB$ records. (b) Multiprocessor generalization of the I/O model in (a), in which each of the $P = D$ internal processors controls one disk and has an internal memory of size M/P. The P processors and their memories function collectively as a CRCW PRAM with global memory size M.

model, corresponding to the Vitter-Shriver model. The more general model, in which the internal processing is done on a CRCW PRAM with P processors, is shown in Figure 4.1b for the special case $P = D$, although we allow P to be as large as M.

4.1.2 Parallel multilevel hierarchies

The simplest multilevel hierarchy model is the Hierarchical Memory Model (HMM) proposed by Aggarwal *et al.* [AAC], depicted in Figure 4.2a. In the $\mathrm{HMM}_{f(x)}$ model, access to memory location x takes $f(x)$ time. Figure 4.2a suggests the $\mathrm{HMM}_{\lceil \log x \rceil}$ model, where each layer in the hierarchy has a size that is some constant multiple of the size of the previous layer. Accesses to records in the first layer take one time unit; in general each record in the nth layer takes n time units. Figure 4.2a can actually be taken as representative of every *well-behaved* cost function $f(x)$. The definition of a well-behaved cost function is that there exists a constant c such that $f(2x) \leq c\, f(x)$ for all x. Therefore, access to layer n will take no more than cn time units, so we have only a constant-factor slowdown. Our model is pessimistic in that $f(x)$ could grow much slower or not at all.

An elaboration of HMM is the Block Transfer (BT) model of Aggarwal *et al.* [ACSa], depicted schematically in Figure 4.2b. Like HMM, it has a cost function $f(x)$, but additionally it simulates the effect of block transfer by allowing the $\ell + 1$ locations $x - \ell, \ldots, x$ to be accessed for cost $f(x) + \ell$. So in the figure, the main cost is incurred

Figure 4.3: Parallel multilevel memory hierarchies. The H hierarchies (of any of the types listed in Figure 4.2) are connected at the base level by an H-processor CRCW PRAM with H global memory locations.

in injecting the "needle"; pushing additional items in (or pulling them out) once the needle has been placed takes only one time unit per item.

A possibly more realistic memory hierarchy is the Uniform Memory Hierarchy (UMH) of Alpern *et al.* [ACF], depicted in Figure 4.2c. This model is described in more detail in Chapter 5.

As with two-level hierarchies, multilevel hierarchies can be parallelized. The parallelization is done in the same way as for the two-level memory hierarchies, as shown in Figure 4.3. The base memory level of the H hierarchies forms an H-processor CRCW PRAM with H memory locations. We assume that the hierarchies are all of the same kind and all have the same parameters. The parallel hierarchical models for HMM, BT, etc., are denoted as P-HMM, P-BT, and so on.

4.2 Main Results

In this section, we present our main results. The modified Balance Sort algorithm we describe in the next section gives us optimal deterministic algorithms for all the models we consider. In particular we get deterministic (as well as more practical) versions of the optimal randomized algorithms based on [ViSa]and [ViN]. We also improve upon the deterministic Greed Sort algorithm in [NoVa], which was optimal only for the parallel disk models and could not be used optimally for hierarchical memories.

Theorem 14 *In the P-HMM model, the time for sorting is*

$$\Theta\left(\frac{N}{H}\log N \log\left(\frac{\log N}{\log H}\right)\right) \qquad \text{if } f(x) = \log x;$$

$$\Theta\left(\left(\frac{N}{H}\right)^{\alpha+1} + \frac{N}{H}\log N\right) \qquad \text{if } f(x) = x^{\alpha},\ \ \alpha > 0.$$

The upper bounds are given by a deterministic algorithm based on Balance Sort. In the lower bound for the $f(x) = x^{\alpha}$ case, the $(N/H)\log N$ term requires the comparison model of computation.

Theorem 15 *In the P-BT model, the time for sorting is*

$$\Theta\left(\frac{N}{H}\log N\right) \qquad \text{if } f(x) = \log x;$$

$$\Theta\left(\frac{N}{H}\log N\right) \qquad \text{if } f(x) = x^{\alpha},\ \ 0 < \alpha < 1;$$

$$\Theta\left(\frac{N}{H}\log N \log\frac{N}{H}\right) \qquad \text{if } f(x) = x^{\alpha},\ \ \alpha = 1;$$

$$\Theta\left(\left(\frac{N}{H}\right)^{\alpha} + \frac{N}{H}\log\frac{N}{H}\log H\right) \qquad \text{if } f(x) = x^{\alpha},\ \ \alpha > 1.$$

The upper bounds are given by a deterministic algorithm based on Balance Sort. The $(N/H)\log N$ terms in the lower bounds require the comparison model of computation.

Theorem 16 *The number of I/Os required for sorting N records in the parallel disk model is*

$$\Theta\left(\frac{N}{DB}\frac{\log(N/B)}{\log(M/B)}\right).$$

The upper bound is given by a deterministic algorithm based on Balance Sort, which also achieves simultaneously optimal $\Theta((N/P)\log N)$ internal processing time, unless $P > M\log\min\{M/B, \log M\}/\log M$ and $\log M/B = o(\log M)$. The lower bounds apply to both the average case and the worst case. The I/O lower bound does not require the use of the comparison model of computation, except for the case when M and B are extremely small with respect to N, namely, when $B\log(M/B) = o(\log(N/B))$. The internal processing lower bound uses the comparison model.

Algorithm 10 [Sort(N, T)]

```
    if N ≤ 12H
        n := ⌈N/H⌉
        for m := 1 to n
(1)         Read H locations to the base level (last read may be partial)
            Sort internally
(2)         Write back out again
(3)     Do binary merge sort of the up to 12 sorted lists
    else
(4)     E := ComputePartitionElements(S)
        { The T array says what location to start with on each hierarchy }
(5)     Initialize X, L
(6)     Balance(T)
        for b := 1 to S
(7)         T := Read bth row of L
            N_b := number of elements in bucket b
(8)         Sort(N_b, T)
(9)         Append sorted bucket to output area
```

4.3 Parallel Memory Hierarchies

This section describes the deterministic algorithm that serves as the basis for the upper bounds in Theorems 14 and 15. Subsection 4.3.1 gives the overall sorting algorithm. In Subsection 4.3.3, we analyze the algorithm for the P-HMM model. Subsection 4.3.4 analyzes the algorithm for the P-BT model.

4.3.1 The sorting algorithm

Algorithm 10 gives the algorithm used for sorting in parallel memory hierarchies. We assume that the records have been augmented with a pointer so that each record can point to the next record in the same bucket on the same hierarchy; this expansion of the records results in no more than a constant increase in the space needed (hence running time).

The difficult part of the algorithm is in making sure that the buckets are (approximately) evenly distributed among the hierarchies; the Balance routine and its descendants handles this complexity. In order for the balancing to occur in optimal deterministic time, it turns out that it is necessary to do partial striping of the hierarchies, so that we will have only H' *virtual hierarchies* with logical blocks of size $B = H/H'$. We will use $H' = \sqrt{H}/\log H$.

There are a number of parameters in the algorithm that merit explanation, some of which are "global" and accessed by the subroutine Balance. We use "global" in quotes, because there is actually a distinct copy of each variable for each level of recursion.

Algorithm 11 [ComputePartitionElements(S)]

partition into G groups of at most $\lceil N/G \rceil$ elements, $\mathcal{G}_1, \ldots, \mathcal{G}_G$
for $g := 1$ to G
(1) sort $\mathcal{G}_g$ recursively
(2) set aside every $\lfloor \log N \rfloor$th element into C
(3) sort C using binary merge sort with hierarchy striping
(4) $e_j :=$ the $\lfloor jN/((S-1)\log N) \rfloor$th smallest element of C

They can be thought of as allocated by the Sort routine in local storage and then having a pointer passed to the routines that need to access them.

N = # of elements to sort
T = array of H' elements pointing to the starting block on each virtual hierarchy
S = # of partition elements
X = (global) $S \times H'$ *histogram* matrix (described later)
E = (global) array of $S-1$ partition elements
L = (global) $S \times H'$ *location* matrix (described later)

We should make a remark about storage of these "global" arrays. We will assume throughout that each array will be stored at the lowest level in which it fits and that accesses to all elements of the array have the same cost. We are justified in doing this because we restrict ourselves to well-behaved cost functions, as described in Section 4.1, so that the levels give within a constant factor an upper bound of the actual cost. The alert reader will notice that we often will be storing more than one matrix at the same level. Since the level is chosen so that each individual matrix (barely) fits in the level, that level would not have enough space for all of the matrices combined. As long as we limit ourselves to a constant number of matrices inhabiting the same level (which we do), we will only be overlooking a constant factor in the cost. Taking these liberties simplifies the analysis considerably without doing damage to the integrity of our results.

The correctness of the algorithm is easy to establish, since the bottom level of recursion by definition produces a sorted list and each level thereafter concatenates sorted lists in the right order. We will determine the value of S differently depending upon which hierarchical model we are using in order to produce optimal performance, as shown in the subsections that analyze the performance for the various hierarchical memory models.

Algorithm 11 gives the routine for computing the partition elements. The specific value of G used in the algorithm is dependent upon which hierarchical memory model is being considered. Although the array $E = \{e_j\}$ is listed as being returned from the routine, it is too large to fit into the base memory level, and is therefore implemented as a "global" variable. This algorithm is essentially identical to that given in [ViSa]. We show the following lemma, which will lead to an analysis of the effectiveness of

Algorithm 12 [Balance(T)]

```
    for each track
(1)     read track
(2)     assign records to buckets (in parallel)
        collect buckets into blocks of size H/H' (all elements of block from same bucket)
(3)     update the placement matrix X
(4)     A := ComputeAux(X)
        if there is a 2 in A
(5)         Rebalance(A, X)
(6)     update the internal pointers of the blocks and the location matrix L
(7)     write track
```

the approximate partitioning elements in dividing the records into nearly equal-sized buckets.

Lemma 15 *Assuming that all keys are distinct, if we choose every pth element from the subsets in line (2) of Algorithm 11, the amount of slop in the size of the buckets produced is less than pG, that is, each bucket b will have N_b elements in it satisfying*

$$\frac{N}{S} - pG < N_b < \frac{N}{S} + pG.$$

Proof: We collect a set C of N/p elements by choosing every pth element from each of the G subsets, where $p = \log N$. We then sort C and choose the ith partition element to be the iqth element of C, where $q = N/((S-1)\log N)$. Let c_k denote the kth smallest element of C. The minimum rank in the original set of the c_{iq} is iqp, since each of the iq elements in C less than or equal to c_{iq} represents p elements in the original set. The maximum rank in the original set of c_{iq} occurs if each of the other $G-1$ subsets has $p-1$ elements greater than c_{iq}. Thus, the amount of slop does not exceed $(G-1)(p-1)$. □

Corollary 2 *If we choose every* $\log N$*th element and we choose* G *such that* $G \log N \leq N/S$ *then we get* $0 < N_b < 2N/S$ *for any bucket b.*

Notice that following the call to ComputePartitionElements, the original set of N records is left as G sorted subsets of approximately N/G records each. This fact is crucially important in allowing for partial hierarchy striping, as described in the next subsections.

Algorithm 12 gives the routine for balancing the buckets. A *track* is a set of H records, one from each hierarchy. We do not assume that the records are at the same location on each hierarchy; however we assume that the records all come from the same level. The locations of the first track on each hierarchy are gotten from the array T. Since T has only H' elements and we have H hierarchies, the first value in T refers to the first H/H' consecutive hierarchies, the second value in T to the second

H/H' hierarchies, and so on. The subsequent tracks are read by following pointers present within each hierarchy; these pointers were placed there by the previous level of recursion. The first level of recursion simply reads the tracks in order.

Collecting the buckets into virtual blocks is surprisingly easy. The value of G is chosen in such a way that the initial sorts of size N/G will put together an average of at least H/H' records from each bucket in each of the S buckets. Thus, we lose no more than a factor of two due to internal fragmentation of the blocks, as shown in the next lemma.

Lemma 16 *If a set containing k distinct values has $N \geq kB$ elements, then it can be broken into no more than $2\lceil N/B \rceil$ blocks such that each block contains all the same value.*

Proof: Write out as many complete blocks all containing the same value as possible. Then write out partially filled blocks containing all the same value. The number of partially filled blocks is at most k. By supposition, $k \leq \lfloor N/B \rfloor$. There are no more than $\lceil N/B \rceil$ completely filled blocks. So the total number of blocks will not exceed

$$\left\lceil \frac{N}{B} \right\rceil + \left\lfloor \frac{N}{B} \right\rfloor \leq 2 \left\lceil \frac{N}{B} \right\rceil .$$

□

We will apply this lemma using $k = S$ and $B = H/H'$. Processing an input track results in processing up to two output tracks; lines (3) to (7) will be executed once for each of the output tracks. We will not consider this issue further as it results in at most a factor of two in the running time.

The *histogram matrix* $X = \{x_{bh}\}$ specifies how the buckets are distributed among the hierarchies; x_{bh} = # of records of bucket b on hierarchy h. Updating the histogram matrix on line (3) simply means that if hierarchy h has read a record from bucket b, that it increments the value of x_{bh}. *Auxiliary matrix* $A = \{a_{bh}\}$ is designed to let us know if the placement becomes too badly skewed. The *location matrix* $L = \{l_{bh}\}$ tells what location was written last on each hierarchy for each bucket. We update the internal pointer in the record to hold the old value of L for the hierarchy and bucket and store the location to which the bucket is written (which is the same location from which the original record on that hierarchy was read) into L. Thus, by starting at l_{bh} and following the pointers, we can read all the records from bucket b on hierarchy h in the inverse of the order in which they were processed at the previous level of recursion.

Algorithm 13 gives the ComputeAux routine. It is listed as returning a matrix A, but since this matrix is larger than will fit into the base memory level, it is another "global" variable. ComputeAux is guaranteed to set at least $\lceil 2H'/3 \rceil$ elements of every row of A to 0. We wish to maintain the buckets so that no element of A is larger than 1. If any element of A becomes larger than 1, then we call the routine Rebalance to restore the balance. In fact, Rebalance will take care of as many buckets as have become unbalanced during the last track.

Algorithm 14 gives the rebalancing routine. At most H' 2s could be introduced by the track being processed into the auxiliary matrix A. The reason is that only H' values

Algorithm 13 [ComputeAux(X)]

for $b := 1$ to S
  $m :=$ $\lceil 2H'/3 \rceil$th smallest element of $x_{b1}, \ldots, x_{bH'}$
  for $h := 1$ to H' **in parallel**
    $a_{bh} := \max(0, x_{bh} - m)$

Algorithm 14 [Rebalance]

$\mathcal{H} :=$ {virtual hierarchies with the first $\lfloor H'/3 \rfloor$ 2s}
(1) Shuffle($\mathcal{H}$)
$\mathcal{H} :=$ {virtual hierarchies with the second $\lfloor H'/3 \rfloor$ 2s}
(2) Shuffle($\mathcal{H}$)
$\mathcal{H} :=$ {virtual hierarchies with the third $\lfloor H'/3 \rfloor$ 2s}
(3) Shuffle($\mathcal{H}$)
$\mathcal{H} :=$ {any remaining virtual hierarchies with 2s}
(4) Shuffle($\mathcal{H}$)

in the histogram matrix X are incremented by the track, one for each virtual hierarchy, and only values of the histogram matrix that are incremented can become 2. We call the routine Shuffle to remove any 2s that are introduced $\lfloor H'/3 \rfloor$ at a time; it follows that at most 4 calls to Shuffle will remove all the 2s that may have been introduced.

Algorithm 15 shows the routine Shuffle, which is able to remove up to $\lfloor H'/3 \rfloor$ 2s in a single parallel memory reference. Lines (4) and (5) of Algorithm 15 are done in parallel. The routine is based on the simple observation that if we read a block from a virtual hierarchy h corresponding to a bucket b that has $a_{bh} = 2$ and we write it to another virtual hierarchy h' for which $a_{bh'} = 0$, then we will have removed the 2. We set up the matching so that we can accommodate up to $\lfloor H'/3 \rfloor$ 2s simultaneously in a single parallel memory reference. We prove these two facts in the next two lemmas.

Lemma 17 *Reading a block from a virtual hierarchy h corresponding to a bucket b that has $a_{bh} = 2$ and writing it to a virtual hierarchy h' for which $a_{bh'} = 0$ and updating X to reflect the change will result in $a_{bh} \leq 1$ and $a_{bh'} = 0$ if ComputeAux is called to recompute A after the operation.*

Proof: There are two cases to consider: (1) incrementing $x_{bh'}$ does not change the $\lceil 2H'/3 \rceil$th smallest element of row b and (2) incrementing $x_{bh'}$ does change the $\lceil 2H'/3 \rceil$th smallest element. In either case, x_{bh} decreases by at least 1 with respect to the $\lceil 2H'/3 \rceil$th smallest element and so a_{bh} will no longer be 2. Also in either case, $x_{bh'} \leq$ the $\lceil 2H'/3 \rceil$th smallest element, so $a_{bh'}$ will be 0. □

Algorithm 15 [Shuffle($\mathcal{H}$)]

{ Create a bipartite matching problem }
$U := \mathcal{H}$
$V := \{1, \ldots, H'\}$
$E := \emptyset$
for $i := 1$ to $|U|$
  $h := U_i$
  $b :=$ (unique) bucket such that $a_{bh} = 2$
  for $j := 1$ to $|V|$ **in parallel**
    $h' := V_j$
(1) **if** $a_{bh'} = 0$
      $E := E \cup (U_i, V_j)$
{ We will swap a pair of blocks for every edge in the following match }
(2) Find a maximum bipartite matching on the graph $G = (U \cup V, E)$
{ The array R will tell us what buckets to read from each hierarchy }
{ The array W will tell us on what hierarchy to write the block }
for $h := 1$ to H'
  $r_h := 0$
  $w_h := 0$
for each match (U_i, V_j) **in parallel**
  $h := U_i$
  $r_h := b$
  $w_h := V_j$
(3) Update X to reflect the swap
(4) Read the last block of bucket r_h on virtual hierarchy h if $r_h \neq 0$
(5) Write the block read from virtual hierarchy h onto virtual hierarchy w_h if $r_h \neq 0$

Lemma 18 *The maximum matching at line (2) of routine Shuffle will always match all* $\lfloor H'/3 \rfloor$ *elements of* U.

Proof: We can give an elementary sequential algorithm to accomplish this task. First, notice that G is a rather special bipartite graph: it has no more than $\lfloor H'/3 \rfloor$ vertices on the left, H' vertices on the right, and every vertex on the left has edges to at least $\lceil 2H'/3 \rceil$ vertices on the right. So if we simply take the first vertex on the left and match it to any vertex on the right, it can remove at most one possible matching vertex on the right for each of the remaining vertices on the left. We can now match the second vertex on the left, and so on. We see that even when we get around to matching the $\lfloor H'/3 \rfloor$th vertex on the left, there are at least $\lceil 2H'/3 \rceil - \lfloor H'/3 \rfloor \geq \lfloor H'/3 \rfloor$ possibilities left for the match. Hence, the match will always complete. □

As can be seen by the proof to Lemma 18, it is not necessary for us to use a maximum matching routine in Algorithm 15; any maximal matching will be a maximum matching.

In order to achieve optimal time for memory hierarchies with a logarithmic access cost, we need to be able to do the matching in time $O(\log H)$. Unfortunately, even the fastest known deterministic parallel algorithm for maximal matching with n items is $O(\log^2 n)$ with a quartic number of processors for dense graphs (which we have) [Luba], or $O(\log^3 n)$ with a quadratic number of processors [IsS] or $n^2/\log n$ processors [GoS]. Since we have $n = H'$, this means that the fastest known algorithm is $\Theta(\log^2 H)$, and still doesn't work since we have only $H = \Theta((H')^2 \log^2 H')$ processors. Subsection 4.3.2 addresses the subject of how to do the matching efficiently.

For the current purposes it is enough to recognize that the Shuffle routine does successfully remove $\lfloor H'/3 \rfloor$ 2s from the auxiliary matrix A. By "removing a 2" from the auxiliary matrix, we mean that if the auxiliary matrix is immediately recomputed after the operation, an element that was a 2 will now be no more than 1, and no 2s will have been introduced by the operation. Hence it follows that if the auxiliary matrix were computed immediately after the call to Rebalance, there will be no elements in it larger than 1.

We now have a theorem that tells how well each bucket is balanced.

Theorem 17 *Any bucket b will take no more than a factor of around 3 above the optimal number of tracks to read.*

Proof: Let m_b denote the $\lceil 2H'/3 \rceil$th element in bucket b. When the Balance routine terminates, we have $\max_h a_{bh} \leq 1$ for any bucket b. Thus, the number of tracks needed to read bucket b is no more than $m_b + 1$. But we also know that the bucket has at least $\lceil H'/3 \rceil m_b$ blocks in it, since the $\lceil H'/3 \rceil$ virtual hierarchies with the most blocks from bucket b each have at least m_b such blocks, so it requires at least

$$\left\lceil \frac{\lceil H'/3 \rceil m_b}{H'} \right\rceil \geq \left\lceil \frac{m_b}{3} \right\rceil$$

tracks to read it. Thus, we are a factor of at most

$$\frac{m_b + 1}{\lceil m_b/3 \rceil} \sim 3$$

from the optimal. (The expression could be as large as 4 if H' is divisible by 3 and $m_b = 3$.) □

Algorithm 12 assumes that it reads H' blocks on every track, whereas the previous phase does not guarantee that every track up to the last is fully used. Adjusting the algorithm for partial tracks is simple: if no block is available to be read on some hierarchy h, we pretend that we read from a dummy bucket 0. Bucket 0 will be have the characteristic that $x_{0h} = 0$ for all $1 \leq h \leq H'$, so in particular, that row of the matching matrix will be all zeroes. The part of the algorithm that increments the values of X should ignore bucket 0.

Algorithm 16 [Fast_Match]

(1) **while** there are unmatched vertices on the left $\Lambda = \{\lambda_i\}$
(2) **while** λ_i has not picked adjacent previously unmatched right vert.
λ_i picks a random vertex on right $\{1, \ldots, H'\}$
(3) **if** vertex on right is picked by more than one on left, smallest wins
add left-right pair to matching

4.3.2 How to do deterministic log-time matching

In this subsection, we discuss how to do deterministic log-time matching. This is not the ground-breaking result it would seem to be, because we are able to take advantage of very special properties of our particular matching problem. Recall that we have $k \leq \lfloor H'/3 \rfloor$ vertices on the left, each of which has edges to at least $\lceil 2H'/3 \rceil$ of the H' vertices on the right. We start by giving a randomized algorithm called Fast_Match, shown in Algorithm 16, for doing our maximal bipartite matching in logarithmic time with a linear (in H') number of processors. Step (3) of this routine is the only place in the entire algorithm where we require the power of the CRCW-PRAM. To prove that this matching can be done in logarithmic time with H' processors, we will show in the following lemmas that Loop (1) of Algorithm 16 will be executed only expected $O(\log H')$ times, and Loop (2) only expected $O(1)$ times. Notice that we can implement the algorithm assigning one processor to each vertex on the right and left, using only $O(H')$ processors.

Lemma 19 *The expected number of iterations of Loop (2) of Algorithm 16 is* $O(1)$.

Proof: Since $k \leq \lfloor H'/3 \rfloor$, at most $\lfloor H'/3 \rfloor - 1$ vertices on the right could already have been matched when we get to Loop (2), or Loop (1) would already have terminated. Since each vertex has at least $\lceil 2H'/3 \rceil$ edges, it follows that at least

$$\left\lceil \frac{2H'}{3} \right\rceil - \left\lfloor \frac{H'}{3} \right\rfloor + 1 \geq \left\lfloor \frac{H'}{3} \right\rfloor + 1$$

of its edges are unmatched. Generating a number from 1 to H' will result in one of these edges being chosen in expected time

$$\frac{H'}{\lfloor H'/3 \rfloor + 1} \leq 3.$$

□

Loop (2) has each unmatched vertex on the left pick a previously unmatched vertex on the right to which it has an edge; this operation is equivalent to having every unmatched vertex on the left pick an edge at random in a graph where edges to any matched vertices on the right have been deleted. There is not enough time to do any actual deletions, but we have enough of a surfeit of edges that we can get the same effect in constant time by just ignoring picks to already-matched vertices on the right.

Lemma 20 *The expected number of iterations of Loop (1) of Algorithm 16 is* $O(\log H')$.

Proof: The question to settle is what fraction of the unmatched vertices on the left will be matched after each has randomly picked an edge to a previously unmatched vertex on the right. If we have k unmatched vertices on the left, then each vertex has at least $\lceil 2H'/3 \rceil - k$ edges to unmatched vertices on the right. If we view the selection process sequentially, the probability that vertex 1 hits a previously selected vertex is 0, the probability that vertex 2 will choose the same one that vertex 1 chose is no more than $1/(\lceil 2H'/3 \rceil - k)$, and in general,

$$\Pr\{\text{vertex } i \text{ hits a previously selected vertex}\} \leq \frac{i-1}{\lceil 2H'/3 \rceil - k}.$$

If we let X_i be a 0-1 random variable specifying whether vertex i hits a new vertex (with respect to vertices 1 to $i-1$), then

$$E(X_i) \geq \frac{\lceil 2H'/3 \rceil - k - i + 1}{\lceil 2H'/3 \rceil - k} = 1 - \frac{i-1}{\lceil 2H'/3 \rceil - k}.$$

If we let n be the expected number of vertices that do not collide, we get

$$\begin{aligned} n = \sum_{1 \leq i \leq k} E(X_i) &\geq \sum_{1 \leq i \leq k} \left(1 - \frac{i-1}{\lceil 2H'/3 \rceil - k}\right) \\ &= k\left(1 - \frac{k-1}{2\left(\lceil 2H'/3 \rceil - k\right)}\right). \end{aligned}$$

So a "constant" fraction of the vertices on the left is matched, where the "constant" is

$$1 - \frac{k-1}{2\left(\lceil 2H'/3 \rceil - k\right)}.$$

This "constant" is minimized as k gets large. But we know that $k \leq \lfloor H'/3 \rfloor$, so that the constant is at least $1/2$. Hence, in expected $O(\log H')$ time, we can match all the vertices on the left. □

This algorithm can be derandomized in an efficient way using the techniques of Luby [Luba, Lubb]. First, notice that we have H processors available instead of only H'. When $H' = \sqrt{H}/\log H$, that means that $H = \Theta((H')^2 \log^2 H')$. The above randomized matching algorithm uses only pairwise independence in the analysis of the running time, so we construct a special probability space to take advantage of the pairwise independence. We let q be a prime number in the range $H' \log H' \leq q \leq 2H' \log H'$ and generate a probability space with $H' \log H'$ random variables and q possibilities for each random variable. Those q possibilities will consist of the numbers $1, \ldots, H'$ repeated $\Theta(\log H')$ times. Since q is not divisible by H', we add enough 0s at the end to pad out to exactly q possibilities for each random variable. Selecting a 0 for the random variable will mean that the pick was missed; this modification has no effect

on the expected running time and leaves the probability distribution unchanged. We will use q^2 processors (which we have available to us) to search the probability space exhaustively in parallel. Since the algorithm terminates in expected $O(\log H')$ steps using expected $O(H')$ random variables on the average, there must be some point in the probability space that achieves at least the average, and hence that can complete the matching with the given number of random variables in $O(\log H')$ steps. We had to choose q at least as large as the number of random variables due to a technicality in the way that Luby's method achieves pairwise independence.

Notice that the matrix of random variables from which we are sampling occupies only $O((H')^2 \log^2(H')) = O(H)$ space, so that it can all be fit at the base memory level of the hierarchies, and we therefore need not consider the cost of accessing the random variables.

To summarize, we have shown the following theorem.

Theorem 18 *The maximal matching problem can be solved deterministically in $O(\log H)$ time using H processors.*

4.3.3 Analysis of P-HMM

We will do the analysis of the sorting algorithm using general $f(x)$ and then compute the specific bound for specific functions $f(x)$. In the P-HMM model, we choose

$$G = \begin{cases} \sqrt{N} & \text{if } N > H^2 \\ \sqrt{\dfrac{N}{2\sqrt{H}\log H}} & \text{if } N \leq H^2 \end{cases}$$

$$S = \begin{cases} \min\left\{ \dfrac{\sqrt{N}}{\sqrt{H}\log H}, \dfrac{\sqrt{N}}{\log N} \right\} & \text{if } N > H^2 \\ \sqrt{\dfrac{N}{2\sqrt{H}\log H}} & \text{if } N \leq H^2 \end{cases}$$

We show that the virtual blocking can be done and that the buckets are approximately the same size in the following lemmas.

Lemma 21 *With the above values of G and S, the average number of elements per bucket per sorted run created by ComputePartitionElements is at least $H/H' = \sqrt{H}\log H$.*

Proof: The average number of elements per bucket per run is N/SG. When $N > H^2$, we notice that $S \leq \sqrt{N}/(\sqrt{H}\log H)$, so $N/SG \geq \sqrt{H}\log H$ as promised. When $N \leq H^2$, $N/SG = 2\sqrt{H}\log H > \sqrt{H}\log H$. □

Lemma 22 *The buckets created using the above values of G and S are approximately the same size.*

Proof: The condition we derived in Corollary 2 is that $G \log N \leq N/S$. When $N > H^2$, $S \leq \sqrt{N}/\log N$, so that $N/S \geq \sqrt{N} \log N$. The condition follows directly. When $N \leq H^2$, we rewrite the condition to be $\log N \leq N/SG$. But when $N \leq H^2$, then

$$\log N \leq 2 \log H < 2\sqrt{H} \log H = \frac{N}{SG},$$

so the condition is satisfied. □

The recurrence for P-HMM is derived in Table 4.1. We assume throughout that access to a data structure of size s has a cost of $f(s/H)$. The extra $\log H$ factor on line (2) of Algorithm 15 is due to the matching time. In Algorithm 12, it is possible to assume that the number of iterations of the loop is $O(N/H)$ because of Theorem 17. There is a trick used to obtain the running time of assigning records to buckets at line (2) of Algorithm 12. Notice that the records have already been sorted into G sets of size N/G by ComputePartitionElements. It follows that in assigning the records to buckets, we will be using the partition elements G times in order using full parallelism; hence the overall time needed for reading the partition elements is what is given in the table. The evaluation of ComputeAux at line (4) of Algorithm 12 can be done incrementally in the given time.

A major simplification of the recurrence occurs by noticing that $H'/H = O((H')^2/H) = O(1)$, since $H' = \sqrt{H}/\log H$. It is also true that $S = O(N/H)$. To show the base of the recursion, consider what happens when $N \leq H^2$. The first recursive call will have arguments of N/G in the first term and $T(N_b)$ in the sum. But we have arranged it so that $N_b \leq 2N/S = 2N/G$, so the largest argument to a recursive call will be

$$\frac{2N}{G} = \sqrt{8N\sqrt{H} \log H} \leq \sqrt{8 \log H} H^{5/4},$$

since $N \leq H^2$. Now consider $N \leq \sqrt{8 \log H} H^{5/4}$, and look at the second level of recursion. Here, again, the maximum argument to any recursive call is $2N/G$, or

$$\begin{aligned} \sqrt{8N\sqrt{H} \log H} &\leq \sqrt{8\sqrt{8 \log H} H^{7/4} \log H} \\ &= 2^{9/4} H^{7/8} \log^{3/4} H \\ &< 12H \end{aligned}$$

for all H. Thus, when $N \leq H^2$, two further levels of recursion always suffice to bring the maximum size subfile to be sorted to no more than $12H$.

The following lemma helps establish the bound for the base case.

Lemma 23 *Two sorted lists whose length totals N can be merged in P-HMM in time $O((N/H)(f(N/H) + \log H))$.*

Proof: We will always keep $r_1 = \lfloor H/2 \rfloor$ records from list 1 and $r_2 = \lceil H/2 \rceil$ records from list 2 in the base memory level, until one of the lists runs out of records. We start by reading the first r_1 records of list 1 and r_2 records of list 2. This operation will

Alg.	Line	Repeats	Time/repeat	Total time
15	1	$\frac{(H')^2}{H}$	$f\left(\frac{SH'}{H}\right)$	$\frac{(H')^2}{H} f\left(\frac{SH'}{H}\right)$
	2	1	$(\log H) f\left(\frac{(H')^2}{H}\right)$	$(\log H) f\left(\frac{(H')^2}{H}\right)$
	3	1	$f\left(\frac{SH'}{H}\right)$	$f\left(\frac{SH'}{H}\right)$
	4,5	2	$f\left(\frac{N}{H}\right)$	$2f\left(\frac{N}{H}\right)$
	Total	$C_{15} = O\left(\left(\frac{(H')^2}{H}+1\right) f\left(\frac{SH'}{H}\right) + \log H\, f\left(\frac{(H')^2}{H}\right) + f\left(\frac{N}{H}\right)\right)$		
14	1–4	4	C_{15}	$4C_{15}$
	Total	$C_{14} = O(C_{15})$		
12	1,7	$O\left(\frac{N}{H}\right)$	$f\left(\frac{N}{H}\right)$	$\frac{N}{H} f\left(\frac{N}{H}\right)$
	2	1	see text	$\frac{GS}{H} f\left(\frac{S}{H}\right)$
	3,6	$O\left(\frac{N}{H}\right)$	$f\left(\frac{SH'}{H}\right)$	$\frac{N}{H} f\left(\frac{SH'}{H}\right)$
	4	$O\left(\frac{N}{H}\right)$	$\frac{(H')^2}{H} f\left(\frac{SH'}{H}\right)$	$\frac{N(H')^2}{H^2} f\left(\frac{SH'}{H}\right)$
	5	$O\left(\frac{N}{H}\right)$	C_{14}	$\frac{N}{H} C_{14}$
	Total	$C_{12} = O\left(\frac{N}{H}\left(f\left(\frac{N}{H}\right) + f\left(\frac{SH'}{H}\right) + C_{14}\right) + \frac{GS}{H} f\left(\frac{S}{H}\right) + \frac{N(H')^2}{H^2} f\left(\frac{SH'}{H}\right)\right)$		
11	1	G	$T\left(\frac{N}{G}\right)$	$G\, T\left(\frac{N}{G}\right)$
	2	G	$\frac{N}{GH\log N} f\left(\frac{\log N}{H}\right)$	$\frac{N}{H\log N} f\left(\frac{\log N}{H}\right)$
	3	1	$\frac{\log N \log\log N}{H} f\left(\frac{\log N}{H}\right)$	$\frac{\log N \log\log N}{H} f\left(\frac{\log N}{H}\right)$
	4	1	$\frac{S}{H}\left(f\left(\frac{S}{H}\right) + f\left(\frac{\log N}{H}\right)\right)$	$\frac{S}{H}\left(f\left(\frac{S}{H}\right) + f\left(\frac{\log N}{H}\right)\right)$
	Total	$C_{11} = G\, T\left(\frac{N}{G}\right) + \left(\frac{N}{H\log N} + \frac{\log N \log\log N}{H} + \frac{S}{H}\right) f\left(\frac{\log N}{H}\right) + \frac{S}{H} f\left(\frac{S}{H}\right)$		
10	4	1	C_{11}	C_{11}
	5	2	$\frac{SH'}{H} f\left(\frac{SH'}{H}\right)$	$2\frac{SH'}{H} f\left(\frac{SH'}{H}\right)$
	6	1	C_{12}	C_{12}
	7	S	$f\left(\frac{SH'}{H}\right)$	$Sf\left(\frac{SH'}{H}\right)$
	8	S	$T(N_b)$	$\sum_b T(N_b)$
	9	S	$\frac{N_b}{H} f\left(\frac{N}{H}\right)$	$\frac{N}{H} f\left(\frac{N}{H}\right)$
	Total	$T(N) = \sum_b T(N_b) + C_{11} + O\left(C_{12} + \left(\frac{SH'}{H} + S\right) f\left(\frac{SH'}{H}\right) + \frac{N}{H} f\left(\frac{N}{H}\right)\right)$		

Table 4.1: Derivation of the recurrence for P-HMM$_{f(x)}$.

require two parallel reads, since the elements will reside on hierarchies $1, \ldots, r_1$ and $1, \ldots, r_2$, respectively. Between the two reads, it is necessary to shift the items from list 1 by r_2 processors to make room in the base memory level for the items from list 2; this shift operation takes $O(\log H)$, since one could implement it by sorting.

We need to add a field to each record keeping track of which list it originated from. We also keep track of which is the next hierarchy to read for each of the lists and which hierarchy is the first to write with our output elements. At each step, we sort the H records in the base memory locations, shift so that the record with the smallest key is on the next hierarchy to write, and write out the $\lfloor H/2 \rfloor$ records with the smallest keys. We update which hierarchy will be the next to write and count how many records from each list were written, say t_1 records from list 1 and t_2 records from list 2. We then read in t_1 records from list 1 and t_2 records from list 2, with appropriate shifts before each read, and update the pointers to the next hierarchy to read for each list. Since at each step, we have at least the next $\lfloor H/2 \rfloor$ records from each list, we can always output the next $\lfloor H/2 \rfloor$ records for the merged list. We require $3N/\lfloor H/2 \rfloor$ I/Os to process the N records, each of which takes time $f(N/H)$. We also need $O(\log H)$ time for sorting and shifting between I/Os, yielding a total time that is $O((N/H)(f(N/H) + \log H))$. □

Corollary 3 *When $N \leq 12H$, the amount of time needed to merge two lists is $O(\log H)$.*

Thus, the overall recurrence reduces to

$$T(N) = \begin{cases} G\,T\left(\frac{N}{G}\right) + \sum_b T(N_b) + O\left(\frac{N}{H}\left(f\left(\frac{N}{H}\right) + \log H\right)\right) & \text{if } N > 12H \\ O(\log H) & \text{if } N \leq 12H \end{cases} \tag{4.1}$$

Lemma 24 *When $f(x) = \log x$, the algorithm given sorts in deterministic time*

$$\frac{N}{H} \log N \log\left(\frac{\log N}{\log H}\right).$$

Proof: Simply plug the correct $f(x)$ into Recurrence (4.1). When $N > H^2$, the given value of S satisfies the recurrence. When $N \leq H^2$, there are only two further levels of recursion giving time $O((N/H)(f(N/H) + \log H) = O((N/H)\log N)$. □

Lemma 25 *When $f(x) = x^\alpha$, the algorithm given sorts in deterministic time*

$$\left(\frac{N}{H}\right)^{\alpha+1} + \frac{N}{H} \log N.$$

Proof: Plug the correct $f(x)$ into Recurrence (4.1). When $N > H^2$, the given value of S satisfies the recurrence. When $N \leq H^2$, there are only two further levels of recursion giving time $O((N/H)(f(N/H) + \log N))$, which is exactly the result. □

Lemmas 24 and 25 complete the proof of Theorem 14. In fact, we can go even farther for the P-HMM model and show that the algorithm is uniformly optimal for any well-behaved cost functions $f(x)$ as defined in Section 4.1.

Theorem 19 *The algorithm given above is optimal for all well-behaved cost functions $f(x)$.*

Proof: This proof is essentially that of the corresponding theorem in [ViSa] for their randomized algorithm. Let $T_{M,H}(N)$ denote the average number of I/O steps the sorting algorithm does with an internal memory of size M, where we allow each hierarchy to move simultaneously, in a single I/O step, a record between internal memory and external memory. Note that we are only viewing the algorithm from the standpoint of I/O, and therefore we can ignore any internal computation time whenever $M > H$. In particular, the matching time does not get counted. When $N > M^2$, the algorithm gives the recurrence

$$T_{M,H}(N) = \sqrt{N} T_{M,H}(\sqrt{N}) + \sum_{1 \le b \le S} T_{M,H}(N_b) + \frac{N}{H},$$

where $\sum_{1 \le b \le S} N_b = N$ and $N_b < 2N/S$ for all $1 \le b \le S$. We have used the cost function

$$f(x) = \begin{cases} 1 & \text{if } x > M/H \\ 0 & \text{otherwise} \end{cases}.$$

When $N < M^2$, we get $T_{M,H} = O(N/H)$. It can be shown that this recurrence is solved by

$$T_{M,H} = O\left(\frac{N}{H}\frac{\log N}{\log M}\right) \tag{4.2}$$

Aggarwal and Vitter showed a lower bound for the problem of sorting with internal memory size M and no blocking or parallelism of

$$T_M = \Omega\left(\frac{N \log N}{\log M} - M\right),$$

where the "$-M$" term allows up to M items to be present initially in the internal memory. At best we could achieve a speed-up of a factor of H by using H disks in parallel, so dividing T_M by H gives a lower bound on the I/O complexity with H disks. But Equation (4.2) is within an additive term of $O(M/H)$ of this optimal I/O bound. At the base memory level, we consider $M = H$, and find that we load in $O(N \log N/(H \log H))$ memory loads, each of which takes $O(\log H)$ communication time for matching or sorting. Thus, the amount of time used in the base memory level by the algorithm is $O((N/H) \log N)$, all of which is above the optimal algorithm. Define

$$\delta f\left(\frac{M}{H}\right) = f\left(\frac{M+1}{H}\right) - f\left(\frac{M}{H}\right).$$

Then the total time above the optimal can be found by adding the amount of time used in the base memory level of the algorithm to the amount used incrementally for

every memory size between H and N. Thus, we get that the amount of time spent above the optimal is

$$\begin{aligned} & O\left(\frac{N}{H}\log N + \sum_{H \le M < N} \frac{M}{H}\delta f\left(\frac{M}{H}\right)\right) \\ = \;& O\left(\frac{N}{H}\log N + \frac{N}{H}f\left(\frac{N}{H}\right) - f(1) - \frac{1}{H}\sum_{H \le M < N} f\left(\frac{M+1}{H}\right)\right) \\ = \;& O\left(\frac{N}{H}\log N + \frac{N}{H}f\left(\frac{N}{H}\right)\right) \end{aligned} \tag{4.3}$$

where the sum at the end of the first line has been expanded and rearranged to produce the second line and the sum at the end of the second line contains fewer than N items each of which has value no greater than $f(N/H)$. The first term in (4.3) is the lower bound resulting from using a comparison-based sorting algorithm. The second term in (4.3) is the minimum amount of time needed to read all of the elements into the base memory level at least once, assuming that the cost function $f(x)$ is well-behaved and that we are therefore justified in counting all the records in the set to be sorted as taking the same amount of time to access as the last record. It follows that the sorting algorithm is optimal. □

4.3.4 Analysis of P-BT

Essentially the same algorithm will work for the P-BT model as we used in the P-HMM model. The only difference is that in Algorithm 10, we need to add another line right after step (6) to reposition all the buckets into consecutive locations on each virtual memory hierarchy. This repositioning is done on a virtual-hierarchy-by-virtual-hierarchy basis, using the generalized matrix transposition algorithm given in [ACSa].

We concentrate in this section on the cost function $f(x) = x^\alpha$, where $0 < \alpha < 1$. Deterministic algorithms for $f(x) = x^\alpha$ with $\alpha \ge 1$ have been reported previously [ViSa]. The upper bound for $f(x) = \log x$ can be no larger than that for $f(x) = \sqrt{x}$ since $\log x < \sqrt{x}$ for all $x > 0$; since the lower bounds are identical it follows that the algorithm for $f(x) = \sqrt{x}$ applied to hierarchies with cost function $f(x) = \log x$ is optimal.

In the algorithm given, we will choose

$$G = \begin{cases} \dfrac{1}{4\log H}N^{1-\alpha} & \text{if } N > H^2 \\ \sqrt{\dfrac{N}{2\sqrt{H}\log H}} & \text{if } N \le H^2 \end{cases}$$

$$S = \begin{cases} \dfrac{N^\alpha}{\sqrt{H}\log N} & \text{if } N > H^2 \\ \sqrt{\dfrac{N}{2\sqrt{H}\log H}} & \text{if } N \le H^2 \end{cases}$$

We show that the virtual blocking can be done and that the buckets are approximately the same size in the following lemmas.

Lemma 26 *With the above values of G and S, the average number of elements per bucket per sorted run created by ComputePartitionElements is at least $H/H' = \sqrt{H}\log H$.*

Proof: The average number of elements per bucket per run is N/SG. When $N > H^2$, $N/SG = \sqrt{H}\log N \geq \sqrt{H}\log H$. When $N < H^2$, the values of S and G are the same as in the P-HMM case, so the same proof holds as before. □

Lemma 27 *The buckets created using the above values of G and S are approximately the same size.*

Proof: The condition we derived in Corollary 2 is that $G\log N \leq N/S$. When $N > H^2$, we require $N^{1-\alpha}\log N \leq \sqrt{H}N^{1-\alpha}\log N$. When $N < H^2$, the values of S and G are the same as in the P-HMM case, so the same proof holds as before. □

As with the P-HMM algorithm, there are at most two additional levels of recursion after $N \leq H^2$. [ViSc] showed that two lists can be merged in P-BT using the same amount of time as that needed to touch all the elements.

We need to make one more change to the algorithm for BT hierarchies, but one that is hard to write explicitly. Aggarwal *et al.* gave an algorithm called the "touch" algorithm [ACSa]. This algorithm takes an array of n consecutive records stored at the lowest possible level and passes them through the base memory level in order, using time $O(n\log\log n)$ for $0 < \alpha < 1$. As it turns out, all the data structures are processed in order throughout the algorithm due to the sorted runs of size N/G created by finding the partitioning elements. The effect of this change is that we get the same recurrence as for the P-HMM model, using an effective cost function $f(x) = \log\log x$. It turns out that the limiting step in the algorithm for P-BT is the need for repositioning the buckets, which can be done using the cited algorithm in time $O\left((N/H)(\log\log(N/H))^4\right)$. So the overall recurrence is

$$T(N) = G\,T\left(\frac{N}{G}\right) + \sum_b T(N_b) + O\left(\frac{N}{H}\left(\left(\log\log\frac{N}{H}\right)^4 + \frac{GS}{H}\log\log\frac{S}{H} + \log H\right)\right). \tag{4.4}$$

for the case $N > 12H$. The case $N < 12H$ is the same as in (4.1).

Lemma 28 *The given algorithm sorts in the P-BT model using time*

$$T(N) = \frac{N}{H}\log N.$$

Proof: Substituting the solution above into Equation (4.4) demonstrates that it is a solution. □

We have now completed the proof for Theorem 15.

Algorithm 17 [Balance_Disks]

for each memoryload
(1) read memoryload
(2) assign all M records to buckets
(3) group buckets together (sort using bucket number as key)
(4) **for** each consecutive D blocks of values **in parallel**
update X
$A :=$ ComputeAux(X)
Rebalance(A, X)
(5) write memoryload

4.4 Parallel Disks with Parallel Processors

In this section, we present a version of Balance Sort for the parallel disk model. The algorithm is optimal in terms of parallel disk I/Os and will also be optimal in the internal processing time when more than one processor is present, up to $P = M \log(M/B)/\log M$.

The algorithm for the parallel disk model is the same as that used for the P-HMM model, except for a few small changes. In Algorithm 10, we use $N \leq M$ rather than $N \leq H$ as the termination condition for the recursion. We use the Balance_Disks algorithm given in Algorithm 17 instead of the Balance routine given in Algorithm 12. Note that the parameter we call D (the number of disks) is equivalent to the parameter we called H for parallel memory hierarchies. We will likewise use partial striping to group D/D' disks together to make D' virtual disks, with $D' = \sqrt{D}/\log D$. We also use a different method for computing the partitioning elements, described in [ViSa]. Finally, we will select

$$S = \left(\frac{M}{B}\right)^{\gamma}.$$

The procedure for finding the partitioning elements uses as a subroutine Algorithm 18, which computes the kth smallest of n elements in $O(n/DB)$ I/Os. It does this by recursively subdividing about an element that is guaranteed to be close to the median. Algorithm 19 gives the algorithm for finding the partitioning elements.

We start by showing the correctness of the overall sorting algorithm. The general proof of correctness for distribution sort applies, except that we must show that we can group into virtual blocks on our virtual hierarchies. If this grouping is not possible, the matching will not correctly distribute records into buckets, and the sorting algorithm will fail. We need to achieve a virtual blocking with $D/D' = \sqrt{D}\log D$ blocks per virtual block so that all the records in the virtual block fall in the same bucket. When we read in a memoryload, we have M records in memory, and an average of M/S records in each bucket. Thus, each bucket fills an average of $M/(SB)$ blocks. We need

Algorithm 18 [Select(S_0, k)]

$t := \lceil |S_0|/M \rceil$
for $i := 1$ to t
    Read M elements in parallel ($\lceil M/DB \rceil$ tracks; the last may be partial)
    Sort internally
    $R_i :=$ the median element
if $t = 1$
    return(kth element)
$s :=$ the median of $R_1, \ldots, R_t$ using algorithm from [BFP]
Partition into two sets S_1 and S_2 such that $S_1 < s$ and $S_2 \geq s$
$k_1 := |S_1|$
$k_2 := |S_2|$
if $k \leq k_1$
    Select(S_1, k)
else
    Select$(S_2, k - k_1)$

Algorithm 19 [ComputePartitionElements(S)]

for each memoryload of records $\mathcal{M}_i (1 \leq i \leq N/M)$
    Read $\mathcal{M}_i$ into memory
    Sort internally
    Construct $\mathcal{M}'_i$ to consist of every $(S-1)$/2th record
    Write $\mathcal{M}'_i$
$\mathcal{M}' := \mathcal{M}'_1 \cup \cdots \cup \mathcal{M}'_{N/M}$
for $j := 1$ to $S - 1$
    $e_j :=$ Select$(\mathcal{M}', 2Nj/(S-1)^2)$

to ensure that $M/(SB) \geq \sqrt{D} \log D$. But

$$\frac{M}{SB} = \left(\frac{M}{B}\right)^{1-\gamma} \geq D^{1-\gamma} > \sqrt{D} \log D.$$

Here we have used the fact that $M \geq DB$. If we choose $\gamma = 1/4$, then we find that for any value of D, $D^{1-\gamma} \geq k\sqrt{D} \log D$, where $k = (e \log 2)/4$, so we have an additional factor of no more than $4/(e \log 2) \approx 2.123$ due to internal fragmentation from having blocks that are not quite full on average.

From the standpoint of showing I/O and internal processing bounds, we need to show that the routine for computing the partition elements produces buckets that have

nearly the same size. We re-apply Lemma 15, except this time we have $p = (S-1)/2$. Substituting in G and S, it results in a slop that is less than $(M/B)^\gamma N/2M$. The ratio between the slop and the average number of elements per bucket is

$$\frac{1}{2M}\left(\frac{M}{B}\right)^{2\gamma} < \frac{1}{2}$$

for any $\gamma > 0$. Thus, we have for any bucket b,

$$\frac{N}{2S} < N_b < \frac{3N}{2S}.$$

The I/O bound is easy to show for the new algorithm. A, X, L, and E all reside in the internal memory, so there is no cost to access them. We have shown that these data structures will fit in the internal memory in a previous chapter.

A "memoryload" is the collection of $O(M)$ records that fit into memory. The only place that I/Os are done in the disk version of the algorithm is in lines (1) and (5) of Algorithm 17 and in computing the partition elements. We can read and write a memoryload using full parallelism using $O(M/DB)$ I/Os. Since there are overall $\lceil N/M \rceil$ memoryloads, we find that the number of I/Os done per call to Balance_Disks is $O(N/DB)$. Vitter and Shriver showed that the partition elements can also be computed using $O(N/DB)$ I/Os [ViSa]. Thus, we get the recurrence

$$T(N) = \begin{cases} ST\left(\frac{N}{S}\right) + O(N/DB) & \text{if } N > M \\ 0 & \text{if } N \le M \end{cases},$$

which has solution

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

This is the same bound as was shown to be optimal for the parallel disk model [AgV].

We devote the remainder of this section to showing that the algorithm operates with optimal internal processing time as long as the number of processors $P \le M \log \min\{M/B, \log M\}/\log M$. The optimal internal processing time (assuming comparison-based sorting) is $\Omega((N/P)\log N)$. The crux of the matter is in showing that we can use the given number of processors efficiently in all aspects of the sorting algorithm.

The algorithm for finding the partition elements does a total of $O(N/DB)$ I/Os. Since the data are always processed in terms of memoryloads, this corresponds to $O(N/M)$ memoryloads. Each of the memoryloads can be processed in time $O((M/P)\log M)$ for any number of processors $P \le M$, giving a total time $O((N/P)\log M) = O((N/P)\log N)$.

The remainder of the internal computation time occurs in the routine Balance_Disks. We assume that the CPUs are inactive during I/O, so there is no CPU time charged during steps (1) and (5) of Algorithm 17.

Steps (2) and (3) of Algorithm 17 are easy to analyze in terms of processing time needed per memoryload. The next two lemmas state these bounds.

Lemma 29 *Step (2) of Algorithm 17 can be done in time $O((M/P)\log(M/B))$ for each memoryload for any $P \leq M$.*

Proof: Assign M/P records to each processor. Each processor can then do a binary search on the $(M/B)^\gamma$ partition elements for each of its records, giving a total time of $O((M/P)\log(M/B))$. □

Lemma 30 *Step (3) of Algorithm 17 can be done in time $O((M/P)\log(M/B))$ for each memoryload if $P \leq M \log\min\{M/B, \log M\}/\log M$ or $\log M = O(\log M/B)$.*

Proof: To group the records within each bucket together, we sort the records in internal memory. If $\log M = O(\log M/B)$ then we can sort the M records using Cole's parallel merge sort algorithm [Col] in time $(M/P)\log M = O((M/P)\log(M/B))$ for any $P \leq M$.

Otherwise, instead of sorting using the keys contained in the records, we sort using the bucket number as the key. We will start by considering only $P \geq M/\log M$. We will make use of a deterministic algorithm by Rajasekaran and Reif that appears as Lemma 3.1 in [RaR]. Their algorithm sorts n elements in the range $1, \ldots, \log n$ in time $O(\log n)$ using $n/\log n$ processors. We need to consider two cases:

> *Case 1:* $M/B \leq \log M$. In this case, we let $n = M$. The key values go from 1 to $(M/B)^\gamma < M/B \leq \log M$ so we can apply the Rajasekaran-Reif algorithm directly to sort in time $\log M$ with $M/\log M$ processors, which we have. But $\log M \leq (M/P)\log(M/B)$ as long as $P \leq M\log(M/B)/\log M \leq M\log\log M/\log M$.
>
> *Case 2:* $M/B > \log M$. We will still let $n = M$, but this time $\sqrt{M/B}$ may be larger than $\log M$. What we do is a radix sort using the Rajasekaran-Reif algorithm as a subroutine, sorting $\log M$ bits at a time. Thus, we need $O(\log(M/B)/\log\log M)$ applications of the algorithm, for time
>
> $$O\left(\log M \frac{\log(M/B)}{\log\log M}\right) = O\left(\frac{M}{P}\log\frac{M}{B}\right)$$
>
> whenever
>
> $$P < \frac{M\log\log M}{\log M} < M\frac{\log(M/B)}{\log M}$$
>
> since $\log M < M/B$.

If $P < M/\log M$, then we can attain the same bound by multitasking $P' = M/\log M$ virtual processors onto the P real processors. □

In order to show that loop (4) of Algorithm 17 can be parallelized efficiently, we need to understand more precisely what we mean by "**in parallel**" on line (4). We are dividing the memoryload into tracks, where each track consists of D consecutive blocks of values. We make the observation that if any track consists entirely of elements from the same bucket, it is not necessary for that track to update X or recompute the

portion of A that will need to be updated by changing X for that bucket. We assume that A is updated incrementally, only modifying those rows that were actually affected by the given track. We thus arrive at the following crucial lemma.

Lemma 31 *At most two tracks need to access values of any specific row of A or X during a memoryload.*

Proof: We use here the fact that the records have been sorted according to their bucket numbers. So any pair of consecutive tracks can have only one bucket in common. Thus, if we ignore any track that consists entirely of elements from the same bucket, the lemma is established. □

So we can parallelize this operation by casting out any tracks that consist of all the same bucket, renumbering the tracks, and then doing all the odd-numbered tracks followed by all the even-numbered tracks. The odd/even tracks are guaranteed not to interfere with one another since they access disjoint rows of A and X. This crucial fact leads to the following lemma.

Lemma 32 *Loop (4) of Algorithm 17 can be done in time $O((M/P)\log(M/B))$ for any $P \leq M$.*

Proof: As shown in Lemma 31, at most two tracks need to access any row of X or A. We will have two repetitions of at most $M/2DB$ tracks. We need to update an element of X for each of the M/BD' virtual blocks, which can be done completely in parallel for any number of processors up to M/BD'. The total amount of time updating X is

$$T_X = \begin{cases} \dfrac{M \log D}{PB\sqrt{D}} & \text{if } P \leq M \log DB\sqrt{D} \\ 1 & \text{if } P > M \log DB\sqrt{D} \end{cases}$$

In both cases, the time is less than $(M/P)\log(M/B)$ for any $P \leq M$.

We can do the ComputeAux routine by these two steps.

1. Sort S sets of D' numbers in parallel to pick the $\lceil 2D'/3 \rceil$th smallest element.
2. Update all the SD' elements of A.

The sorts take time

$$T_{A1} = \begin{cases} \dfrac{1}{P}\left(\dfrac{M}{B}\right)^{\gamma} \dfrac{\sqrt{D}}{\log D} \log\left(\dfrac{\sqrt{D}}{\log D}\right) & \text{if } P \leq \left(\dfrac{M}{B}\right)^{\gamma} \dfrac{\sqrt{D}}{\log D} \\ \log(\sqrt{D}/\log D) & \text{if } P > \left(\dfrac{M}{B}\right)^{\gamma} \dfrac{\sqrt{D}}{\log D} \end{cases}$$

In both cases, the time is no more than $(M/P)\log(M/B)$ for any $P \leq M$. The update time is

$$T_{A2} = \begin{cases} \dfrac{1}{P}\left(\dfrac{M}{B}\right)^{\gamma} \dfrac{\sqrt{D}}{\log D} & \text{if } P \leq \left(\dfrac{M}{B}\right)^{\gamma} \dfrac{\sqrt{D}}{\log D} \\ 1 & \text{if } P > \left(\dfrac{M}{B}\right)^{\gamma} \dfrac{\sqrt{D}}{\log D} \end{cases}$$

Again, the time is no more than $(M/P)\log(M/B)$ for any $P \leq M$ if $M \geq D^{2/3}/(B^{1/3}\log^{4/3} D)$, which is true since $D^{2/3}/\log^{4/3} D < D$ for all $D \geq 2$.

Finally, we can accomplish the rebalancing by the following steps:

1. Set up M/BD matching problems, each of which is $D' \times D'$.
2. Solve the M/BD matching problems.
3. Do M/BD sets of swaps.

The set-up time can be done in parallel time

$$T_{R1} = \begin{cases} \dfrac{M}{PB\log^2 D} & \text{if } P \leq \dfrac{M}{B\log^2 D} \\ 1 & \text{if } P > \dfrac{M}{B\log^2 D} \end{cases}$$

Each matching problem can use up to D processors to achieve time $\log D$. The matching takes time

$$T_{R2} = \begin{cases} \dfrac{M\log D}{PBD} & \text{if } P \leq \dfrac{M}{B} \\ \log D & \text{if } P > \dfrac{M}{B} \end{cases}$$

Both of these steps take time no more than $(M/P)\log(M/B)$ for any $P \leq M$. Finally, the swaps involve moving up to M items, but no internal processing time other than touching them, for a total time of $O(M/P) \leq (M/P)\log(M/B)$ for any $P \leq M$.

We have shown that each individual step meets the time bound proposed by the lemma, establishing its validity. □

Finally, we are ready to prove our main result for disk sorting, Theorem 16.

Proof: We established the I/O bound above. To show that the CPU time is optimal, we need to show that it takes a total of $O((N/P)\log N)$ time. We have shown that each memoryload can be processed using time $O((M/P)\log(M/B))$ whenever either $P \leq M\log\min\{M/B, \log M\}/\log M$ or $\log M = O(\log M/B)$. The number of memoryloads we need to process the file is $O((N/M)\log(N/B)/\log(M/B))$. Therefore, the total time is

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

as desired. □

4.5 Conclusions

In this chapter, we have described the first known deterministic algorithm for sorting optimally using parallel hierarchical memories. This algorithm improves upon the randomized algorithms of Vitter and Shriver [ViSa]. The algorithm applies to P-HMM, P-BT, the parallel UMH hierarchies in Chapter 5, as well as the parallel disk model. The parallel disk version of the algorithm also improves upon the algorithms of the previous chapters in the following ways:

1. The hidden constants in the big-O notation are small. The problem with the Greed Sort algorithm is that it uses Columnsort [Lei] as a subroutine, which introduces at least an additional factor of 4 into the constant of proportionality.

2. The algorithm can operate using only striped write operations. Some parallel disk subsystems, such as Thinking Machine's Data Vault, impose this requirement so that error checking and redundancy information can be maintained to give acceptable levels of reliability when dealing with many disks, each of which has an independent probability of failure. It is unclear how to adapt Greed Sort to use only striped writes, again because of the embedded call to Columnsort.

3. The new algorithm can be made to work optimally in terms of internal processing time even in the presence of large degrees of processor parallelism.

The primary disadvantage of the algorithm presented in this chapter is that it requires that the CPUs be connected to the memories by a CRCW-PRAM. The CRCW-PRAM is required only in the log-time matching algorithm, so any improvement there will automatically improve the overall algorithm. Ideally, one would like to be able to do the matching in $O(\log H)$ time on a hypercube or other actual parallel processor topology.

Although we have presented a deterministic algorithm, in practice the randomized algorithm resulting from doing randomized matching and not requiring any partial hierarchy striping would be simpler to implement.

Chapter 5

Sorting in Uniform Memory Hierarchies

5.1 Introduction

In most large-scale computer systems, memory progresses from very small but very fast registers to successively larger but slower components, such as several layers of cache, primary memory, disks, and archival storage. In order to achieve optimal performance on such a computer system, it is often necessary for the algorithm designer to take into account the physical characteristics of the entire memory hierarchy. Unfortunately, there are too many possible variables to consider (e.g., the block size of each level, the number of blocks at each level, the bandwidth between one level and the next) to allow the design of general algorithms; hence some degree of abstraction of the memory hierarchy is required.

Several interesting and elegant hierarchical memory models have been proposed recently to model the many levels of memory typically found in large-scale computer systems. The HMM model of Aggarwal, Alpern, Chandra, and Snir [AAC] views the entire memory hierarchy as a linear address space and allows access to individual location x in time $f(x)$. We normally think of $f(x)$ as a non-decreasing function, such as $\log x$ or x^α, for some $\alpha > 0$,[1] depending on the characteristics of the memory hierarchy.

One important effect that is missing from the HMM model but that is generally present in memory hierarchies is that it is often almost as fast to transfer a whole block of data as it is to retrieve a single data element. This effect happens in disks, for example, because the amount of time needed to seek to a given sector is typically much larger than the amount of time needed to transfer the sector once the read/write head is in place. Accordingly, data are usually transferred in large units of *blocks*. The BT model of Aggarwal, Chandra, and Snir [ACSa] represents a notion of block transfer applied to HMM; in the BT model, access to the $t + 1$ records at locations $x - t$, $x - t + 1$, ..., x takes time $f(x) + t$. A model similar to the BT model that

[1] We use the notation $\log x$, where $x \geq 1$, to denote the quantity $\max\{1, \log_2 x\}$.

allows pipelined access to memory in $O(\log n)$ time was developed independently by Luccio and Pagli [LuP]. Optimal sorting algorithms for each of these models have been developed [AAC, ACSa, LuP], and optimal algorithms for the parallelizations of the HMM and BT models was given in Chapter 4.

In this chapter, we concentrate on a newer hierarchical memory model introduced by Alpern, Carter, and Feig [ACF, ACSb], called the Uniform Memory Hierarchy (UMH), which offers an alternative model of blocked multilevel memories. Let us consider a hierarchy comprising several levels of memory models, where the ℓth level has the following parameters associated with it:

$$\begin{aligned} s_\ell &= \#\text{ of elements that fit in a block} \\ n_\ell &= \#\text{ of blocks} \\ b_\ell &= \#\text{ of elements that can move from level } \ell \text{ to level } \ell+1 \text{ per unit time} \end{aligned}$$

This model has enough parameters that it could be adjusted to suit any memory hierarchy. Unfortunately, the very flexibility in the parameters makes the model too detailed to be of theoretical or practical interest. A programmer would like to be able to design an algorithm that could work without modification in a wide range of memory hierarchies. In the $\mathrm{UMH}_{b(\ell)}$ model (for integer constants $\alpha, \rho \geq 2$), the ℓth memory level (as illustrated in Figure 5.1) consists of $n(\ell) = \alpha\rho^\ell$ blocks, each of size $s(\ell) = \rho^\ell$; it is connected by buses to levels $\ell - 1$ and $\ell + 1$. Each individual block on level ℓ can be randomly accessed as a unit and transferred to or from level $\ell + 1$ at a *bandwidth* of $b(\ell)$; that is, each block transfer takes $\rho^\ell / b(\ell)$ time units. We have switched from subscripts on s, n, and b to emphasize that there is a functional relationship. Thus, in the UMH model, the number of blocks and the size of each block on a given level is a multiplicative factor of ρ larger than those on the previous level, and the bandwidth is a well-behaved function of the level. A block on one level corresponds to a sub-block in the next level.

We can assign a linear ordering to the addresses in a UMH. The CPU resides at level 0. Level 0 will contain locations $1, \ldots, \alpha$, level 1 will contain locations $\alpha + 1, \ldots, \alpha + \alpha\rho^2$, and so on.

Table 5.1 gives a comparison between a real system, the IBM RISC System/6000 memory hierarchy, and a UMH model whose parameters are chosen to approximate the architecture. The UMH model captures the flavor of the real memory hierarchy, although there are some differences. In this chapter, we will consider three bandwidth functions: $b(\ell) = 1$, $b(\ell) = 1/(\ell + 1)$, and $b(\ell) = \rho^{-c\ell}$. The constant bandwidth case is the fastest one that makes any sense theoretically, although it is unrealistic in practice. The exponentially decaying function approximates the real bandwidth better than the linearly decaying one. Level 2 (and its connection to level 3) seems to be somewhat out of kilter from the rest of the hierarchy; discounting that level results in a linearly decaying bandwidth that fits as well as the exponential one.

A model for parallel hierarchies was introduced by Vitter and Shriver, in which H hierarchies are connected at their base level via an interconnection network as shown in Figure 5.2. Communication between the H hierarchies takes place at the *base memory*

Figure 5.1: The uniform memory hierarchy (UMH), pictured here for the case $\alpha = 3$, $\rho = 2$. The ℓth memory level contains $\alpha\rho^\ell$ blocks, each of size ρ^ℓ. It is connected by buses to levels $\ell - 1$ and $\ell + 1$. Each level ℓ block can be randomly accessed and transferred to level $\ell + 1$ at a *bandwidth* of $b(\ell)$ (that is, in $\rho^\ell/b(\ell)$ time). This figure shows only the blocks (and sub-blocks for level $\ell + 1$); the individual records are not shown.

level (call it level 0), which consists of location 1 from each of the H hierarchies. In this chapter we will assume that the "network" connecting the processors is a CRCW-PRAM, as in Chapter 4. Vitter and Shriver introduced optimal randomized sorting algorithms for P-HMM and P-BT [ViSa]. The algorithms were based on their randomized two-level partitioning technique applied to the optimal single-hierarchy algorithms for HMM and BT developed in [AAC, ACSa]. In Chapter 4, we gave optimal deterministic sorting algorithms for P-HMM and P-BT.

We can consider parallel UMH hierarchies (analogous to P-HMM and P-BT), and we call the resulting model P-UMH. (This is fundamentally different from the parallel type of UMH called UPHM mentioned in [ACF].) Each level ℓ in each of the H hierarchies of the P-UMH holds $\alpha\rho^{2\ell}$ records, for a total of $\alpha H\rho^{2\ell}$ records on level ℓ. Accordingly, we assume that the initial input of N elements resides at level $s = \lceil \frac{1}{2} \log_\rho \frac{N}{\alpha H} \rceil$, which is the smallest level that holds all the elements.

Figure 5.2: A parallel hierarchical memory. The H individual memory hierarchies are all of the same type, such as HMM, BT, or UMH. The H CPUs can communicate among one another via the interconnection network (which can be a hypercube or cube-connected cycles, for example).

A UMH or P-UMH "program" consists of a schedule of choreographed block transfers and computations. If a RAM program that runs in $T(N)$ steps can be scheduled in UMH in $\sim T(N)$ time, the program is said to be *parsimonious*; note that the constant

factor must be 1. If the UMH program runs in time $O(T(N))$, it is said to be *efficient*. A UMH program whose running time is within a constant factor of the best possible for that problem in the UMH model is said to be *optimal*. We can define "parsimonious", "efficient", and "optimal" analogously for P-UMH by comparing to a PRAM program.

When we consider sorting in UMH, we assume that each memory location can hold one of the records to be sorted. In Section 5.2 we give optimal and near-optimal sorting algorithms for UMH and P-UMH for a wide range of bandwidth rates $b(\ell)$, and we present a parsimonious schedule for merge sort for the case $b(\ell) = 1$. In Section 5.3 we also introduce a natural and easy-to-program restriction of UMH, called random-access UMH (or RUMH), for which we have optimal upper and lower bounds for all bandwidths and amounts of parallelism, and a sequential model of UMH called SUMH, for which we do the same.

5.2 Sorting in UMH and its Parallelization

Optimal sorting in $O(N \log N)$ time in UMH is possible only when the bandwidth $b(\ell)$ at level ℓ is $\Omega(1/\ell)$, or else the time required just to access the N records will be greater than $O(N \log N)$. Many buses may be active simultaneously in the UMH model, so conceivably it is possible to sort in $O(N \log N)$ time even with small bandwidth $b(\ell) = 1/(\ell+1)$. The FFT computation, which is similar to sorting, can be done in $O(N \log N)$ time with bandwidth $b(\ell) = 1/(\ell+1)$. Whether or not an $O(N \log N)$-time $\mathrm{UMH}_{1/(\ell+1)}$ algorithm exists is a big open problem.

In this section we give a near-optimal sorting algorithm for the small bandwidth case $b(\ell) = 1/(\ell+1)$, and optimal sorting algorithms for several other bandwidths. For the special case of constant bandwidth, we present a parsimonious algorithm. Since optimal sorting seems to require nonoblivious UMH programs, the oblivious UMH model of [ACF] must be modified in a reasonable way. In Theorem 20, we assume that the ℓth level of the hierarchy can initiate a transfer from the $(\ell+1)$st level without involving the CPU when one of its blocks becomes empty. In the remaining theorems, we assume that the CPU can originate the transfer of a block at level ℓ given the address of the block, with suitable delay.

The fastest oblivious algorithm we have found for sorting in $\mathrm{UMH}_{1/(\ell+1)}$ is based on a simple schedule of Batcher's bitonic sort [Akl] where each of the $\log^2 N$ parallel time steps is implemented in $O(N \log N)$ time for an overall running time of $O(N \log^3 N)$. It is also possible to schedule a recursive version of Columnsort [Lei] on $\mathrm{UMH}_{1/(\ell+1)}$ in a manner that is efficient with respective to the RAM algorithm, but this observation is not very useful since both algorithms have running time that is $O(N \log^c N)$, where $c \approx 3.4$.

5.2.1 Parsimonious sorting in UMH_1

Theorem 20 *A variant of merge sort can be scheduled in UMH_1 parsimoniously, assuming $\alpha \geq 2$ and $\alpha\rho \geq 6$.*

Proof: The basic idea is to schedule a systolic binary merge sort in such a way that the CPU is always kept busy (except for a small initial delay and with a small final delay for propagating the results back). The amount of time needed to move the first element from each of the two lists down to the CPU is $2\sum_{0\le k<s}\rho^k = O(\sqrt{N})$, where $s = \frac{1}{2}\log_\rho(N/\alpha)$ is the starting level. After the initial delay, the CPU (level 0) reads one element from each of the two lists. At every time step thereafter, the CPU compares the two elements, writes the smaller element to the output list and then reads the next element from the list that had the smaller element at the previous step. We use a double-buffering scheme so that level ℓ, for $\ell \ge 1$, contains room for two blocks from each of the two lists being merged. It also has two blocks for the output list. When level $\ell - 1$ requests a sub-block from one of the lists, and this request causes level ℓ's buffer to be emptied, then level ℓ requests the next block from level $\ell + 1$. In this way, level ℓ always has at least one sub-block for level $\ell - 1$ available on demand. The output blocks fill up at a known rate, so they can be scheduled in advance (again using double-buffering to keep an empty sub-block available for writing from level $\ell - 1$). At the end of each list, we immediately begin to send a new list down for the next merge. The CPU can keep track of how many elements have been read from each list, so that when one list is finished it knows to copy the rest of the other list to the output. There will be a final delay of $O(\sqrt{N})$ as the last results propagate back to the starting level. The number of wasted CPU cycles is only $O(\sqrt{N}) = o(N \log N)$, so the schedule is parsimonious. □

5.2.2 Sorting in P-UMH

In the previous subsection, we were rather explicit in describing the choreography of data movement within the UMH. The remainder of the chapter uses simulations of P-HMM and P-BT algorithms. Accordingly, we have to state how to derive a choreography from simulating one of these algorithms. The difficulty in choreographing the data movement is in making effective use of the blocking on the UMH. We therefore restrict ourselves to simulating algorithms that access blocks of data (even though the P-HMM model does not take advantage of those access patterns). When a block of data is moved from one level ℓ of a UMH to a lower level ℓ', we choreograph this movement by sending the first block from level ℓ to level $\ell - 1$. We start sending the second block from level ℓ to level $\ell - 1$ while simultaneously sending the first block to level $\ell - 2$ sub-block by sub-block. Since we consider only non-increasing bandwidth functions, the space occupied by the first block on level $\ell - 1$ will always have been completely freed by the time we are ready to send the third block from level ℓ to level $\ell - 1$. The net effect is that we can keep the bus between levels ℓ and $\ell - 1$ always busy. In transferring to a level $\ell' > \ell$, we adopt the same tactics in reverse: we time the transfers from the lower levels so as always to keep the bus between levels $\ell' - 1$ and ℓ' busy.

Theorem 21 *Distribution sort algorithms can be scheduled on P-UMH with the following running times. The algorithms for nonconstant H for the first two bandwidth*

cases are based on Balance Sort.

$$\Theta\left(\frac{N}{H}\log N\right) \qquad \text{if } b(\ell)=1;$$

$$O\left(\frac{N}{H}\log N\log\left(\frac{\log N}{\log H}\right)\right) \qquad \text{if } b(\ell)=\frac{1}{\ell+1};$$

$$\Theta\left(\left(\frac{N}{H}\right)^{1+c/2}+\frac{N}{H}\log N\right) \qquad \text{if } b(\ell)=\rho^{-c\ell},\ c>0.$$

Proof: The lower bound for the first case $b(\ell)=1$ follows from the conventional $\Omega(N\log N)$ serial bound for sorting on a RAM. With H processors, the P-UMH sorting time can be at most H times faster, giving an $\Omega((N/H)\log N)$ lower bound. The best known lower bound for the case $b(\ell)=1/(\ell+1)$ is the same as when $b(\ell)=1$. To prove the lower bound when $b(\ell)=\rho^{-c\ell}$, we assume that the N records reside initially in level $s=\lceil\frac{1}{2}\log_\rho\frac{N}{\alpha H}\rceil$. To move the N items from level s to level $s-1$, we need to move N/ρ^s blocks, each of which takes time $\rho^s/b(s)=\rho^{(c+1)s}$, with at most H blocks transferred simultaneously. It follows that it takes time

$$\frac{N}{\rho^s}\frac{\rho^{(c+1)s}}{H}=\frac{N\rho^{cs}}{H}=\frac{N}{H}\rho^{(c/2)\log_\rho(N/\alpha H)}=\frac{N}{H}\left(\frac{N}{\alpha H}\right)^{c/2}=\Omega\left(\left(\frac{N}{H}\right)^{1+c/2}\right)$$

to get the N records from level s to level $s-1$, thus completing the lower bound for the third case.

The algorithm that achieves the upper bound for the first case $b(\ell)=1$ is based on a simulation of the P-BT algorithm for access cost function $f(x)=\sqrt{x}$ given in Chapter 4. This observation is based on the fact that any algorithm for P-BT with cost function $f(x)=\sqrt{x}$ can be simulated on a P-UMH with bandwidth function $b(\ell)=1$ with a slowdown of at most the constant $2/\sqrt{\alpha}$. Consider a transfer of the $b+1$ elements from locations $x-b,\ldots,x$ to locations $y-b,\ldots,y$ in the BT model with access cost function $f(x)=\sqrt{x}$. This transfer will take time $T_{\rm BT}=\sqrt{x}+\sqrt{y}+b$ when the access cost function $f(x)=\sqrt{x}$. We now compute the amount of time taken by a UMH to do the same transfer with multiple levels of the hierarchy active concurrently. We define the level function

$$\mathrm{lev}(x)=\left\lceil\frac{1}{2}\log_\rho\frac{x}{\alpha}\right\rceil,$$

which is the level on which location x appears in a single UMH. Let us assume for the time being that all of the elements in the block $x-b,\ldots,x$ occur on the same level and that $x>y$. At most two partly-filled blocks will be transferred from any level in the range $\mathrm{lev}(y)+1,\ldots,\mathrm{lev}(x)$. Level $\mathrm{lev}(x)-1$ will receive its first elements after time $\rho^{\mathrm{lev}(x)-1}$ (we assume that the whole block has to arrive before any of the elements are accessible). It can then start sending elements down to level $\mathrm{lev}(x)-2$. Level $\mathrm{lev}(y)$ will start getting its first elements after a delay of

$$\sum_{\mathrm{lev}(y)\le k<\mathrm{lev}(x)}\rho^k=\frac{\rho^{\mathrm{lev}(x)}-\rho^{\mathrm{lev}(y)}}{\rho-1}.$$

Since

$$\rho^{\mathrm{lev}(x)} = \rho^{\log_\rho(x/\alpha)/2} = \sqrt{\frac{x}{\alpha}},$$

the delay before the first elements arrives is

$$\frac{\sqrt{x/\alpha} - \sqrt{y/\alpha}}{\rho - 1} = \frac{\sqrt{x} - \sqrt{y}}{\sqrt{\alpha}(\rho - 1)}.$$

Bad blocking imposes an additional delay of $\rho^{\mathrm{lev}(x)-1} = \sqrt{x/\alpha}/\rho$, and each additional element after the first block imposes a delay of 1. Thus, the total amount of time that is needed is no more than

$$\frac{\sqrt{x} - \sqrt{y}}{\sqrt{\alpha}(\rho - 1)} + \frac{\sqrt{x}}{\sqrt{\alpha}\rho} + b,$$

which is bounded by cT_{BT} for all $c \geq 2/\sqrt{\alpha}$. The case where $x < y$ is handled analagously and is likewise bounded by cT_{BT}. Thus, the P-UMH time is within a constant factor of the P-BT time with cost access function $f(x) = \sqrt{x}$, which is $O((N/H)\log N)$.

The other simulations presented in this chapter are not quite so straightforward, since they are not based on a wholesale simulation of one parallel memory hierarchy by another, but rather by simulating an algorithm intended for one hierarchy on another. As a result, we need to be certain that the algorithm meets certain constraints. A sufficient constraint is that operations in the algorithm process all the elements at any level in the hierarchy consecutively. It may be convenient for the algorithm to describe the elements as comprising groups that may be unrelated to the block size for that level, but as long as all the elements are accessed consecutively, the intermediate levels of the hierarchy can act as buffers to allow reblocking to occur as needed without losing efficiency, as long as $\alpha \geq 3$. The algorithms given in Chapter 4 all meet this constraint.

For the second case $b(\ell) = 1/(\ell + 1)$, the upper bound is related to the P-HMM approach for $f(x) = \log x$ given in Chapter 4. We need to modify the P-HMM algorithm to reblock the buckets prior to sorting them recursively, in the same way as the algorithm for P-BT was, by reblocking the buckets into consecutive locations after the call to Balance. The cost of accessing an element at location x in the HMM model is $\log x$; the amortized cost of accessing the same element in the UMH model when an entire block is brought to the base memory level (like the "touch" algorithm in the BT model) is $\log(x/\alpha)/\log\rho$, which is within a constant factor of the HMM cost. Thus, the overall cost is

$$O\left(\frac{N}{H}\log N \log\left(\frac{\log N}{\log H}\right)\right),$$

just as for P-HMM with cost function $f(x) = \log x$.

The upper bound third case $b(\ell) = \rho^{-c\ell}$ is possible using deterministic, two-way merge sort. We can use the same algorithm as we used in the proof for Theorem 20. The exact choreography is unimportant, since the overall transfer time of any transfer will be dominated by the amount of time needed to move to and from the highest level

involved. This algorithm gives rise to the recurrence relation

$$S(N) = \begin{cases} 2S\left(\frac{N}{2}\right) + O\left(\left(\frac{N}{H}\right)^{c/2+1}\right) & \text{if } N > H; \\ O(\log N) & \text{if } N \leq H, \end{cases}$$

which gives the stated bound. □

The algorithms are optimal, except for the middle $b(\ell) = 1/(\ell+1)$ case, which is off from the best known lower bound of $\Theta((N/H)\log N)$ by a $\log((\log N)/\log H)$ factor.

5.3 Sorting in SUMH and RUMH and their Parallelizations

The UMH model can be difficult to program because many buses can be active simultaneously. An earlier version of [ACF] introduced a *sequential* UMH model, appropriately called SUMH, that allowed at most one bus to be active at a time. However, the SUMH restriction can be regarded as too severe, since it forfeits much power of the UMH model.

We introduce the following more natural and less severe restriction that fits in nicely with feasible and easy-to-use programming languages: We require that the UMH program correspond exactly to a RAM program in which we augment the RAM instruction set with a block move command that can move t contiguous memory elements in time t, for arbitrary t. Each such block transfer can be implemented in UMH by a coordinated series of transfers in which several buses are simultaneously active but cooperating on that single transfer. We call this natural variant of UMH the *random-access* UMH model, or simply RUMH. For example, a block of $\sqrt{N}$ elements can be moved from the top memory level all the way down to the CPU (or anywhere in between) in $\sim \sqrt{N}$ time in UMH_1 and RUMH_1, but it requires $\Theta(\sqrt{N}\log N)$ time in SUMH_1.

The parallel versions of RUMH and SUMH are called P-RUMH and P-SUMH, respectively. Theorems 22 and 23 give matching upper and lower bounds for sorting in the RUMH and SUMH models and their parallelizations. The structures of the formulas in Theorems 22 and 23 suggest several different relationships between the RUMH and SUMH models on the one hand and the HMM, BT, and two-level models on the other hand (cf. Theorems 14 and 15); accordingly the upper and lower bounds combine in an interesting way several techniques from [AAC, ACSa, AgV, ViSa].

Theorem 22 *The running times mentioned in Theorem 21 are matching upper and lower bounds for sorting in P-RUMH. The algorithms for nonconstant H for the first two bandwidth cases are based on Balance Sort.*

Proof: The upper bounds all follow directly from the proof of Theorem 21, since all the algorithms given there are P-RUMH algorithms.

The lower bounds for $b(\ell) = 1$ and $b(\ell) = \rho^{-c\ell}$ are the same as and follow directly from those for P-UMH. When $b(\ell) = 1/(\ell+1)$, we can prove a tight lower bound by

simulating $\text{RUMH}_{1/(\ell+1)}$ by HMM with access cost function $f(x) = \log x$. We compute the amount of time that it takes to transfer a block of $\rho^{\ell-1}$ elements from level ℓ to level ℓ' where $\ell > \ell'$. The amount of time before any elements start arriving at level ℓ' is

$$\sum_{\ell' \le k < \ell} \frac{\rho^k}{b(k)} = \sum_{\ell' \le k < \ell} (k+1)\rho^k.$$

Since

$$\sum_{0 \le k \le n} k a^k = \frac{a^{n+1}((a-1)(n+1)-a)+a}{(a-1)^2},$$

the amount of time is

$$\frac{\rho^\ell((\rho-1)\ell-\rho) - \rho^{\ell'}(\rho-1)\ell'-\rho)}{(\rho-1)^2} + \frac{\rho^\ell - \rho^{\ell'}}{\rho-1} = \Theta(\ell\rho^{\ell-1}).$$

The extra time needed after the first elements have arrived at level ℓ' is no more than $\ell\rho^{\ell-1}$, so the overall time needed is $\Theta(\ell\rho^{\ell-1})$. Doing the same block move in the HMM model requires time at most

$$\rho^{\ell-1}(\log(\alpha\rho^{2\ell}) + \log(\alpha\rho^{2(\ell'+1)})) = O(\ell\rho^{\ell-1}),$$

so the simulation by HMM is bounded by a constant times the $\text{RUMH}_{1/(\ell+1)}$ running time. Hence, the lower bound for P-HMM for $f(x) = \log x$ given in [ViSa] also holds for $\text{P-RUMH}_{1/(\ell+1)}$. □

Theorem 23 *The following bounds are matching upper and lower bounds for sorting in P-SUMH. The algorithms for nonconstant H for the first two bandwidth cases are based on Balance Sort.*

$$\begin{aligned}
&\Theta\left(\frac{N}{H}\log N \log\left(\frac{\log N}{\log H}\right)\right) && \text{if } b(\ell) = 1;\\
&\Theta\left(\frac{N}{H}\log N \log\frac{N}{H}\right) && \text{if } b(\ell) = \frac{1}{\ell+1};\\
&\Theta\left(\left(\frac{N}{H}\right)^{1+c/2} + \frac{N}{H}\log N\right) && \text{if } b(\ell) = \rho^{-c\ell},\ c > 0.
\end{aligned}$$

Proof: We prove the lower bounds using an approach similar to that of [ViSa]. Let us define the "sequential time" of a P-SUMH algorithm to be the sum of its time costs for each of the H hierarchies. The sequential time can be at most H times the P-SUMH running time. We superimpose on the P-SUMH model a sequence of one-disk, two-level memories of the type studied in [AgV, ViSa], in the following way: For $1 \le \ell \le \frac{1}{2}\log_\rho(N/\alpha H)$, the ℓth two-level memory has one disk, internal memory size $M_\ell = H\alpha(\rho^{2(\ell+1)}-1)/(\rho^2-1)$, and block size $B_\ell = \rho^\ell$. An I/O in the ℓth two-level memory corresponds to a single block transfer between levels ℓ and $\ell+1$ in one of

the hierarchies in the P-SUMH model, which requires sequential time $C_\ell = B_\ell/b(l)$. Substituting various bandwidth functions gives us

$$C_\ell = \begin{cases} B_\ell & \text{if } b(\ell) = 1; \\ \Theta(B_\ell \log B_\ell) & \text{if } b(\ell) = \dfrac{1}{\ell+1}; \\ B_\ell^{1+c} & \text{if } b(\ell) = \rho^{-c\ell},\ c > 0. \end{cases}$$

The minimum number of I/Os required for sorting in the ℓth two-level memory is

$$\Omega\left(\frac{N}{B_\ell}\frac{\log(N/B_\ell)}{\log(M_\ell/B_\ell)} - \frac{M_\ell}{B_\ell}\right),$$

as shown in [AgV]. Each such I/O contributes C_i to the sequential time in the P-SUMH model, since in the P-SUMH model only one level can be active at a time in each hierarchy. This gives a lower bound on the P-SUMH sequential time:

$$T(N) = \Omega\left(\sum_{1 \le \ell \le \frac{1}{2}\log_\rho(N/\alpha H)} C_\ell \left(\frac{N}{B_\ell}\frac{\log(N/B_\ell)}{\log(M_\ell/B_\ell)} - \frac{M_\ell}{B_\ell}\right)\right).$$

We get the desired lower bound on the P-SUMH time by substituting the values of M_ℓ, B_ℓ, and C_ℓ for the three cases into the above summation, and then dividing by H. The $b(\ell) = \rho^{-c\ell}$ case additionally requires the use of the conventional $N \log N$ serial bound for sorting.

The upper bounds for the first two cases $b(\ell) = 1$ and $b(\ell) = 1/(\ell+1)$ are achieved by simulating the optimal P-HMM algorithm of [ViSc], for access cost functions $f(x) = \log x$ and $f(x) = \log^2 x$, respectively. In each case, we alter the P-HMM algorithm in the same way as that for P-BT by reblocking the buckets into consecutive locations after the call to Balance. Since a UMH in each case can simulate an HMM with the appropriate cost function in a running time that is at most a constant times the HMM time, the P-HMM bound holds for the P-UMH simulation. The stated bounds can be plugged into Recurrence (4.1) along with the appropriate cost function to show that the recurrence is satisfied. The same deterministic merge sort as the previous theorems achieves the upper bound for the $b(\ell) = \rho^{-c\ell}$ case. □

5.4 Conclusions

We have given optimal or near-optimal sorting algorithms for UMH and its parallelization that we have introduced called P-UMH. We have derived tight matching upper and lower bounds for sorting in the restricted models RUMH and SUMH and their parallelizations. All of the algorithms are deterministic. The RUMH model is particularly useful because it is easy to visualize and it matches well with current programming languages and compilers.

An interesting open problem is whether it is possible to sort in $O(N \log N)$ time with the $\mathrm{UMH}_{1/(\ell+1)}$ model. The related FFT computation can be done in $\mathrm{UMH}_{1/(\ell+1)}$ in $O(N \log N)$ time. Another open problem is whether a parsimonious oblivious algorithm exists to replace our non-oblivious one in UMH_1.

Chapter 6

Lattice Computations

A two-dimensional cellular automaton, in its simplest form, is a discrete, infinite rectangular grid of cells, each of which assumes one of two possible states ("on" or "off") at any given instant. The evolution of a cellular automaton takes place in discrete time steps called *generations*. Each cell determines in parallel what its state will be at the next time step, based on its current state and the states of the cells around it. Cellular automata can be generalized to lattice computations, in which each cell retains more than a single bit of information. In this chapter, we restrict ourselves for brevity to the class of lattice computations where the computation at each cell requires information from its eight neighboring cells, as well as from itself. We call these *nine-cell lattice computations*. The algorithms and lower bounds we develop can be extended easily to other interconnection patterns.

A famous example of a cellular automaton is the game of "Life," which John Conway introduced in 1969. Despite the apparent simplicity in having purely local rules govern the time evolution of the system, Life exhibits complex behavior and in fact possesses the same computational power as a Turing machine [Dew]. It also admits of a universal constructor to allow self-replicating structures [Pou].

Lattice computations have important applications in physical simulations. Examples of simulations that use such automata are two-dimensional lattice gas computations [KSS], diffusion-limited aggregation [MaT], two-dimensional diffusion, fluid dynamics, spin glasses, and ballistics [ToM]. A VLSI circuit to solve the Poisson equation has been implemented using lattice computation techniques [Man].

Highly local data movement coupled with a tremendous potential for parallelism would seem to make these problems an ideal match for VLSI. Kugelmass *et al.* showed, however, that VLSI-based machines performing such computations are severely constrained by input/output (I/O) requirements [KSS]. Using a simpler new argument, we improve by a constant factor the theoretical lower bound on I/O that can be derived from their work. We then present and analyze four VLSI architectures within this framework. One of these, the multigeneration sweep architecture, is optimal in that it meets the lower bound asymptotically within a small constant factor. In all cases, we derive and compare the constants of proportionality.

The quantity of interest in the following analysis is the *I/O overhead* ψ, which we define as

$$\psi = \frac{I + O}{Cg}, \tag{6.1}$$

where

$$\begin{aligned} I &= \#\text{ of input operations,} \\ O &= \#\text{ of output operations,} \\ C &= \#\text{ of cells computed,} \\ g &= \#\text{ of generations computed.} \end{aligned}$$

The I/O overhead ψ reflects an amount of I/O needed per unit of computation that is independent of the problem size. This quantity does not assume that the results must be viewed after every generation; the number of I/O operations is amortized against the amount of progress made.

In Section 6.1, we give a lower bound argument based on pebbling to show that the I/O overhead for nine-cell lattice computations is at least $1/\sqrt{2S}$, where S is the amount of on-chip storage. In Section 6.2, we give several architectures for lattice computations, one of which, the multigeneration sweep architecture, is within a constant factor of the lower bound. Appendix A proves a closed form for the optimal number of generations to compute in the multigeneration sweep architecture before viewing the results. Our conclusions are given in Section 6.3.

6.1 Lower Bounds on I/O Overhead

Graph pebbling is a powerful technique for proving computational lower bounds [Coo, HoK, KSS, SaV]. For nine-cell lattice computations, a general result in Kugelmass *et al.* can be specialized to

$$\psi \geq \frac{1}{8\sqrt{\alpha}} n^{-1},$$

assuming that each of n^2 processors (e.g., an $n \times n$ array) has α bits of local storage. Their result can be improved by factor of $\sqrt{2}$ by taking advantage of the connectivity of nine-cell lattice computations instead of the five-cell computations they consider.

In this section, we improve upon this lower bound by a simpler argument. We make no assumptions about when results of the computation are viewed. If the states of all cells must be examined after every generation, we show that $\psi \geq 2$. This last result implies that the most effective use of VLSI implementations is for analyzing the long-term behavior of lattice computations.

Our arguments, like those of Kugelmass *et al.*, are based on the *red-blue pebble game*. The red-blue pebble game is played by placing pebbles on the vertices of a *computation graph* $G = (V, E)$, which is a directed acyclic graph modeling a computation. Each vertex in V corresponds to a result that is computed at some stage of the computation. An edge $e = (v_1, v_2)$ in the graph indicates that the result corresponding to vertex v_1

is used to compute the result for vertex v_2. Pebbles represent memory locations. A red pebble represents a result that is in memory on the chip, and a blue pebble represents a result that is off-chip. There is an implicit assumption that there is a limited number of red pebbles, while the number of blue pebbles is as large as needed. The actual pebbling takes place according to the following rules:

1. A pebble of either color may be removed from a vertex at any time.
2. A red pebble may be placed on any vertex that has a blue pebble.
3. A blue pebble may be placed on any vertex that has a red pebble.
4. If all the immediate predecessors of a vertex v have a red pebble, then a red pebble may be placed on v.

Rule 1 represents the forgetting of information (usually to reuse the memory). Rules 2 and 3 model input and output operations, respectively. Rule 4 models a computation taking place within the chip. In this chapter, we ignore internal computations and consider only I/O. Our measure of performance is the number of applications of rules 2 and 3.

Those vertices of the graph that have no predecessors are the *lattice inputs*, and those that have no successors are the *lattice outputs*. The initial configuration has blue pebbles on all the lattice inputs.

Definition 3 A *pebbling* of a computation graph is a sequence of ordered pairs (n, v), where $n \in \{1, \ldots, 4\}$ is a rule number and $v \in V$, such that starting from the initial configuration and applying the rules to the nodes in sequence results in all of the lattice outputs having blue pebbles. An *S-pebbling* is a pebbling with the restriction that at most S red pebbles are on vertices of the lattice at any one time.

We define the computation graph $G_g = (V_g, E_g)$ for a lattice computation of g generations as follows: Choose m such that the size of the problem during all g generations can be bounded by an $m \times m$ array of cells. The vertex set is

$$V_g = \{(i, j, t) \mid 0 \leq i, j < m \text{ and } 0 \leq t \leq g\},$$

and the edge set is

$$\begin{aligned} E_g \quad = \quad & \{((i,j,t),(i',j',t')) \mid (i,j,t),(i',j',t') \in V_g \\ & \text{with } t' = t+1,\ |i'-i| \leq 1, \text{ and } |j'-j| \leq 1\}. \end{aligned}$$

The cells on the border have either three or five predecessors, rather than eight. The computation is then a sequence starting at time $t = 0$ and progressing towards larger t. When $t = g$, the outputs have been computed, and the computation halts.

We first present a few lemmas leading up to our main lower bound result. The basic strategy is to bound the number of computations that can be done with no inputs starting from any configuration, and then compute the minimum number of

inputs that must be done in order for every node to have been red-pebbled. We start by showing that we can consider only pebblings that red-pebble the generations in order.

Lemma 33 *Let us consider a pebbling game in which an initial configuration of $S < m$ red pebbles is given, and only rules* 1 *and* 4 *are allowed for pebbling moves (that is, no I/O operations are allowed). Then there is a pebbling for maximizing the number of computations in the lattice such that all red pebbles are placed on nodes in earlier generations before any are placed on nodes in later generations.*

Proof: Let R be the set of vertices pebbled by some pebbling starting from an initial configuration K of $S-1$ red pebbles. We do not need to consider configurations of S red pebbles, since the first operation in such a configuration will always be an application of rule 1, giving a configuration of $S-1$ red pebbles. We show that there is a schedule that pebbles earlier generations before later ones that results in all the vertices in R being pebbled. Let $g+1$ be the first generation containing some node in R. For $S < m$, there is a pebble on some vertex v on generation g that contributes to the placing of exactly one pebble on generation $g+1$, say on vertex w. Apply rule 4 to place a red pebble on w and apply rule 1 to remove the pebble from vertex v. Since the pebble on vertex v was used only in computing w, removing the pebble from there does not decrease the number of possible vertices that can be pebbled. Moreover, we can consider this as a new starting configuration K' with a new set $R' = R - \{w\}$, since again at most $S-1$ pebbles are on the graph. Continuing in this way guarantees that all the vertices in R eventually have a red pebble on them. Since this can be done for any pebbling, then in particular any pebbling that maximizes the number of nodes pebbled can be redone in an order that pebbles earlier generations before later ones. □

Next, we show that we can group the inputs into phases, suffering at most a penalty of requiring twice as many red pebbles.

Lemma 34 *Any red-blue S-pebbling P with T inputs can be simulated by some $2S$-pebbling P' of the following type:*

1. *P' can be divided into* phases *such that in each phase, all inputs are done consecutively at the beginning of the phase.*
2. *P' has $\lceil T/S \rceil$ phases, each containing at most S input operations.*
3. *The individual input operations in P' are a subsequence of those in P.*

Proof: We can prove this lemma by an easy simulation of the original pebbling. Let $I_1, \ldots, I_T$ be the inputs in pebbling P. We transform P into P' by moving $I_{kS+2}, \ldots, I_{kS+S}$ to follow immediately after I_{kS+1}, for $k = 1 \ldots \lceil T/S \rceil$, eliminating those inputs that would be to a location that already has a red pebble on it. We then maintain the invariant that at the beginning of each phase k in P', the same nodes in the computation graph have red pebbles on them as in P just prior to doing I_{kS}. This invariant can be maintained inductively. It is initially true since no nodes have

red pebbles on them in either pebbling. Assume that P and P' are in the same configuration just prior to doing I_{kS}. Then P' must have $r < S$ pebbles on the graph, so it has at least S available to do all the inputs for the phase. After doing the inputs, P and P' both have $S - r$ pebbles left, so P' can exactly simulate P with the exception that it need not put a red pebble on any vertex already containing one. At the end of the phase, P' removes red pebbles from all vertices not covered with red pebbles by P, maintaining the induction hypothesis. □

Finally, we show how many nodes in the lattice can be red-pebbled using only internal computations.

Lemma 35 *The maximum number of nodes in the lattice that can be pebbled with S red pebbles using rule 4 alone is $(S^{3/2} - S)/2$, assuming $S < m$.*

Proof: Let us consider any schedule that maximizes the number of nodes pebbled using rule 4 only. By Lemma 33 we can assume that the schedule proceeds generation by generation, for generations $0, 1, \ldots, g-1$. Each generation will be pebbled in parallel. Let $A_i \geq 0$ be the number of red pebbles moved while pebbling generation $i+1$ from generation i, and let $B_i \geq 0$ be the number of red pebbles "left behind" on generation i. We have

$$\sum_{0 \leq i \leq g-1} B_i = S, \tag{6.2}$$

since each red pebble must eventually be left behind, or more moves would be possible. The term A_i is maximized if all S pebbles are on generation i at some point in the schedule, arranged in a square in a corner of the lattice. The fact that $S \leq m$ means that the square can touch at most one corner. If this is the case, then A_i will be exactly $(\sqrt{S} - 1)^2$. Thus, for any value of i we have

$$A_i \leq \left(\sqrt{S} - 1\right)^2. \tag{6.3}$$

Likewise, the border of each cluster of pebbles must be left behind unless the border is at the edge of the lattice. The perimeter of a cluster of S pebbles is at least $4\sqrt{S}$, at most half of which can be along the edge of the lattice, so we have

$$B_i > 2\sqrt{A_i}. \tag{6.4}$$

We can get an upper bound on $\sum_i A_i$, the number of nodes pebbled, by maximizing $\sum_i A_i$ subject to constraints (6.2), (6.3), and (6.4), where A_i and B_i are allowed to be real nonnegative numbers. Thus, $\sum_i A_i$ is strictly bounded by a hypothetical case with $A_i = (\sqrt{S} - 1)^2$ for $S/B_i = S/(2(\sqrt{S} - 1))$ values of i. This gives us the bound $\sum_i A_i \leq (S^{3/2} - S)/2$, which proves the lemma. □

We are now ready to prove our main lower bound result.

Theorem 24 *For lattice computations with $g \geq 2$, in which there are $S < m$ bits of on-chip storage, we have*

$$\psi \geq \frac{1}{\sqrt{2S}}.$$

Proof: Lemma 34 showed that every S-pebbling strategy with T I/Os can be simulated efficiently by a $2S$-pebbling of roughly T/S phases in which at most S inputs occur at the beginning of each phase. We use Lemmas 33 and 35 to bound the number of internal computations that can be done on the lattice during one phase. Since we know how many nodes there are in the lattice that must be red-pebbled, we get a lower bound on the number of inputs required.

From Lemma 35, the maximum number of internal computations that can be done without I/O using $2S$ red pebbles is

$$\frac{(2S)^{3/2} - 2S}{2}.$$

Since a total of gm^2 lattice elements must be red-pebbled, the number of phases is at least

$$G = \left\lceil \frac{gm^2}{\frac{1}{2}\left((2S)^{3/2} - 2S\right)} \right\rceil \geq \frac{gm^2}{\frac{1}{2}(2S)^{3/2}} + \frac{gm^2}{2S^2} \geq \frac{gm^2}{\sqrt{2}S^{3/2}} + 1.$$

Here, we have used the facts that $m > S$ and $g \geq 2$. Since each S-pebbling of the lattice with T I/Os has a corresponding $2S$-pebbling with $\lceil T/S \rceil$ phases, every S-pebbling must have at least $S(G-1)$ input operations. Thus, from formula (6.1) defining the I/O overhead, the number of input operations is at least

$$\frac{gm^2}{\sqrt{2S}},$$

leading to the result. □

In particular, in an array of n^2 processors, each of which has α bits of storage, the I/O overhead is at least

$$\psi \geq \frac{1}{\sqrt{2\alpha}} n^{-1}. \tag{6.5}$$

What happens if we insist that we want to see the results of the calculation after every generation? This corresponding lower bound is easy to determine.

Lemma 36 *The I/O overhead to compute one generation of a lattice computation and look at the results is at least 2.*

Proof: In this case, the computation graph consists of only two layers, each with m^2 cells. Clearly it will take a minimum of m^2 inputs to red-pebble the inputs and a minimum of m^2 outputs to blue-pebble the outputs, once they have been red-pebbled. Thus, we have

$$\psi_{\text{every gen.}} \geq \frac{2m^2}{m^2} = 2, \tag{6.6}$$

as claimed. □

Figure 6.1: Interprocessor communication.

So any lattice computation in which we wish to examine all the cells at each generation will have an I/O overhead $\psi \geq 2$, regardless of the number of generations computed, assuming no values are retained on the chip between generations. The bound in (6.6) is clearly worse than that of (6.5), where the cells do not have to be examined at each generation. VLSI chips to do lattice computations seem to be most suitable for studying the asymptotic behavior of initial configurations.

6.2 Architectural Approaches

In this section we present and analyze four parallel VLSI architectures for lattice computations. Each one uses a two-dimensional processor grid. The limited-size architecture attempts to keep the entire problem in the processor array. The various sweep architectures permit the processor array to sweep over portions of a much larger problem. In the case of the sweep architectures, the I/O overhead is derived for the "steady-state" case, that is, assuming that the problem size is large with respect to the number of processors.

6.2.1 Limited-size architecture

The most straightforward approach to simulating a lattice computation is to assume there exists a one-to-one correspondence between processors and cells. For this we use n^2 processors arranged in an $n \times n$ square mesh, each communicating directly with eight neighbors, as shown in Figure 6.1. We can only consider computations of size $m \times m$, with $m \leq n$, as there are no provisions for saving intermediate results. Each processor requires one storage location to hold its current state. The algorithm used is:

Figure 6.2: Tessellating the problem space.

1. Input $m \times m$ problem.
2. Compute for G generations.
3. Output results.

Notice that, although the problem initially fits entirely within the array of processors, there is no guarantee that it will continue to do so as $G \to \infty$. Conservatively, we must take $G = \lfloor (n-m)/2 \rfloor$. After this point, we must stop and output the results, since we can no longer be sure that we will get the same result as would be obtained in an infinite grid. A short derivation using (6.1) shows that the I/O overhead for this architecture is

$$\psi_{ls} = 2n^{-1} + 2mn^{-2} + O\left(m^2 n^{-3}\right).$$

With $\alpha = 1$, the lower bound from (6.5) tells us that

$$\psi > \frac{1}{\sqrt{2}} n^{-1}.$$

Thus, the limited-size architecture has an I/O overhead at most 2.83 times the optimal. In practice, of course, this scheme is not useful because it ignores problems larger than $n \times n$.

6.2.2 Array sweep architecture

In this architecture, described by Toffoli [ToM], the complete lattice computation is stored in an $m \times m$ grid of memory cells. The processor array repetitively loads $n \times n$

subproblems, updates states by a single generation, and writes results back to their original off-chip locations. In this manner, the processors wind their way through the problem space, tessellating the computation as demonstrated in Figure 6.2. The algorithm for this architecture is:

1. For each subproblem in tessellation, do steps 2–4.
   2. Input $n \times n$ subproblem.
   3. Compute for one generation.
   4. Output results.

Here the I/O overhead is found to be

$$\psi_{as} = 2 + 4n^{-1} + 4n^{-2}.$$

Because the array sweep architecture outputs each state at every generation, (6.6) correctly predicts that $\psi \geq 2$. A slight improvement over this is possible by observing that the rightmost two columns of a given computation become the leftmost two columns in the next computation; these values need not be written to memory as they will always be re-read immediately. With this change the I/O overhead becomes

$$\psi_{mas} = 2 + 2n^{-1}.$$

6.2.3 $n/2$-Generation sweep architecture

The array sweep architecture computes only a single generation for each subproblem before moving on to the next. This policy seems wasteful when I/O overhead is a primary concern. Why not compute multiple generations once a given subproblem has been loaded? The potential pitfall here is that states at the memory-processor border are not updated as they should be. As the computation progresses, these incorrect values propagate their effect inward. Eventually, after $(n-1)/2$ generations, only the centermost cell is correct (assuming n is odd). Nevertheless, we can proceed by saving this one valid value and loading a new subproblem. Fortunately, this last step is greatly simplified if each processor stores an additional bit, its original state. These values are left-shifted one position before beginning the next computation, so that only $n+2$ new values need to be input along the array's right border. In this architecture, we no longer tessellate the problem space, but truly sweep it. The $n/2$-generation sweep architecture employs the following algorithm:

1. For each row in problem space, do steps 2–5.
   2. For each column in problem space, do steps 3–5.
      3. Left-shift original states one position, input $n+2$ new values.
      4. Compute for $(n-1)/2$ generations.
      5. Output result from centermost processor.

The I/O overhead can be computed as

$$\psi_{n/2} = 2 + 4n^{-1} - 4n^{-2} + O(n^{-3}).$$

Thus, this architecture is slightly better than the original array sweep architecture, but not as good as the modified version.

Figure 6.3: Multigeneration sweep architecture.

6.2.4 Multigeneration sweep architecture

The previous two schemes are merely the extreme cases in a spectrum of architectures. It is possible to work anywhere between the array sweep architecture and the $n/2$-generation sweep architecture, as shown in Figure 6.3. Processor requirements in this case are the same as those in the $n/2$-generation sweep architecture, except for the need to output a $k \times k$ matrix of results. We modify the algorithm slightly so that the processor array computes only $(n-k)/2+1$ generations in step 4.

Toffoli and Margolus mention this technique under the name "scooping", but do not analyze it from the standpoint of I/O efficiency [MaT, ToM].

I/O overhead for this architecture is easiest to derive in terms of k, but can be related to g since

$$k = n - 2(g-1).$$

The equation is

$$\psi_{mgs} = \frac{2(n-g+2)}{g(n-2g+2)}. \tag{6.7}$$

If we do the asymptotics directly from this equation for I/O overhead, we get the unrealistic idea that the I/O overhead can become arbitrarily good if the number of generations computed goes to infinity. This is not only unreasonable, since at most $\lfloor n/2 \rfloor$ generations can be validly computed, but it also seems to contradict the finding for the $n/2$-generation sweep architecture, which has a constant I/O overhead. The

discrepancy is that the approximation used in generating the asymptotic expression becomes invalid when g approaches $n/2$.

We can adopt a different approach and compute the value of g that minimizes the expression for I/O overhead. By setting the partial derivative of ψ with respect to g equal to zero, we get

$$n + 2 \pm \frac{\sqrt{2}}{2}\sqrt{n^2 + 4n + 4} = \left(1 \pm \frac{\sqrt{2}}{2}\right)(n + 2),$$

of which the negative root gives the minimum. Thus,

$$\tilde{g}_{\min} = \left(1 - \frac{\sqrt{2}}{2}\right)(n + 2) \tag{6.8}$$

The tilde superscript is to indicate that this quantity is always an integer multiple of an irrational number and is therefore always non-integral. What we want is the discrete minimum for (6.7). We show in Appendix A that taking the nearest integer of (6.8) gives the discrete minimum:

$$g_{\min} = \left\lfloor \tilde{g}_{\min} + \frac{1}{2} \right\rfloor.$$

Thus, we can calculate the asymptotic I/O overhead by noting that

$$g_{\min} = \left(1 - \frac{\sqrt{2}}{2}\right) n + O(1),$$

so

$$\psi_{mgs} = \left(6 + 4\sqrt{2}\right) n^{-1} + O(n^{-2}).$$

The pebbling bound in this case (assuming $\alpha = 2$) is

$$\psi > \frac{1}{2} n^{-1}.$$

Thus, this architecture is asymptotically optimal and falls within a factor of about 23 of the best possible I/O overhead.

6.3 Conclusions

In this chapter, we discussed I/O overhead ψ as a measure of merit for parallel VLSI architectures for lattice computations. We derived theoretic lower bounds on ψ based on the red-blue pebbling game. We presented and analyzed four potential architectures showing that one, the multigeneration sweep architecture, was optimal in terms of ψ within a small constant factor. Finally, we proved the discrete minimization formula that results in the optimal performance of the multigeneration sweep architecture.

Do the asymptotic differences exhibited in this chapter have any practical significance? Table 6.1 indicates values of ψ for three of our schemes. The multigeneration

n	array sweep ψ	$n/2$-gen. ψ	multigeneration $g_{\min}$	multigeneration ψ
1	4	4	1	4
2	3	—	1	3
3	2.67	3	1	2.67
4	2.5	—	2	2
5	2.4	2.67	2	1.67
6	2.33	—	2	1.5
7	2.28	2.5	3	1.33
8	2.25	—	3	1.17
9	2.22	2.4	3	1.07
10	2.2	—	4	1
11	2.18	2.33	4	0.9
21	2.10	2.18	7	0.51
31	2.06	2.12	10	0.35
101	2.02	2.04	30	0.11
1001	2.00	2.00	294	0.01

Table 6.1: I/O overhead for various architectures.

.

sweep architecture is noticeably superior even when n is relatively small. A value of 31 is achievable with current technology, resulting in an almost sixfold improvement in I/O performance; using wafer technology, a chip with 100^2 processors is conceivable, in which case the factor is more than 18.

It is noteworthy that when n is larger than about 6, the I/O performance of the array sweep architecture proposed by other researchers is almost independent of n. In a problem like this, for which the limiting factor is I/O, we have the undesirable result that packing more processors onto a chip does not help much.

We conclude that a VLSI implementation of a two-dimensional lattice computation does not need to be severely restricted by its I/O performance so long as viewing the results after each generation is unnecessary. In this case, the increased complexity of designing a chip to use the multigeneration sweep architecture may be more than offset by its I/O efficiency.

An interesting extension to this work currently being investigated is the I/O overhead associated with simulating neural net computations.

Chapter 7

Blocking for External Graph Searching

7.1 Introduction

Searching is a fundamental topic in computer science [Knu]. In external searching, the records to be searched cannot fit simultaneously in the internal memory. In this chapter, we consider searching in graphs. There is a process that generates a path through a graph, and it is necessary for each vertex in the path to be present in memory in order to decide which vertex to visit next. We assume that internal memory is capable of holding M vertices, and that data are read from the secondary storage in blocks of B vertices, where of course $B \leq M$. The goal is to minimize the total number of block transfers.

We show that optimal blocking speed-ups can be achieved for many graphs as long has some degree of replication of data is present. We study this problem under a wide variety of assumptions concerning replication of data and structure of the underlying graph. For the case of grid graphs blocked using isothetic hypercubes, we show that with no duplication of data, there is a provably worse bound on the optimal blocking speed-up.

We are expanding on the research area of efficient data storage, where in our case we have some knowledge (in the form of a graph) of the possible data accesses. We do not, however, know the sequence of accesses in advance. There are many applications for using blocks efficiently in external graph searching. These include A.I. searching in constraint networks, robot motion planning, the simulation of large Deterministic Finite Automata (DFA), browsing in hypertext applications, accesses in object-oriented databases, and some matrix algorithms such as searching in monotone arrays [AgP]. Sometimes the graphs are well-structured, as in the case of matrix searches and robot motion planning in a space discretized in a grid (which gives rise to grid graphs). At other times the graphs are unstructured, as in A.I. searching and DFA simulation. For the purposes of this paper, we assume that all graphs are undirected, an assumption that may not hold for certain applications such as hypertext and object-oriented

databases.

One important assumption of our model is that data may be multiply represented in blocks. This is a stronger assumption than that used, for example, by external B-trees, where a record of data may be pointed to from several places, although the data (exclusive of the key being indexed) exist in only one block [Ull]. We allow actual replication of the data and define the storage blow-up s to be the ratio between the actual number of blocks used to represent the data and the minimum number that could be used.

We know of no previous work for this problem, but there is considerable related work. Borodin *et al.* considered a similar problem where they were traversing an access graph, but they considered only block size $B = 1$ and worked on developing an efficient paging algorithm [BIR]. Alternatively, all the work done in the database community on B-trees could be viewed as a solution to our problem for complete trees with $s = 1$.

There is considerable previous work on embedding of one type of data structure into another [AlR, CRS, DEL, Rosa, Rosb, Rosc, RoS]. This work is relevant because if we view the data structures as graphs, the goal of the embedding is to maintain locality in the original graph when accesses are done according to the target graph. If the data structures can be stored so that local references in the structure correspond to local references in storage, then it is hoped that such a locality-of-reference would lead to fewer page faults. Of course, this is only a heuristic, and local references in the target graph do not always get translated into local references in the host. For example, Rosenberg considered the problem of preserving proximity in arrays [Rosa] when mapping onto a linear access structure. A grid graph describes the proximity properties of arrays, so the embedding problem here is to map a grid graph into a semi-infinite number line. An interesting hypothesis is that blocking the data elements that are mapped into consecutive sets of B numbers on the number line would give good performance for graph searching in a grid graph. Rosenberg showed that there is no linear mapping scheme that preserves proximity globally in arrays that are extendible in more than one dimension. DeMillo *et al.* considered the same problem and showed that proximity could be maintained if the storage mechanism was a binary tree [DEL]. Both concluded that the standard row-major (or column-major) way of storing arrays was asymptotically optimal for preserving locality in arrays. We find, however, that our earlier hypothesis concerning blocking does not hold even for finite arrays, as long as the array structure is much larger than the memory size. The embedding literature also fails to take into account the fact that data items may be multiply represented. An alternative approach is taken by Lipton *et al.*, who consider hierarchies of embeddings of graphs, where they do allow a given vertex to be replicated up to S times in going from one level of the hierarchy to the next, as well as a factor of T expansion in the distance between vertices [LED]. Their paper, however, focuses on lower bounds for general graphs, and does not deal with the issue of how to find optimal embeddings. It is also not clear what the relevance of any optimal embeddings is to blocking.

In Section 7.2, we state the specifics of the model we are using for blocking in external graph searching. Section 7.3 gives some general results on paging strategies. Section 7.4 gives upper and lower bounds for searching in general graphs. Sections 7.5

and 7.6 consider upper and lower bounds for the blocking efficiency of searching in trees and grid graphs, respectively. Finally, Table 7.1 in Section 7.7 gives a summary of the results.

7.2 Model

We summarize the key assumptions of the model here, for the graph $G = (V, E)$.

1. The data are associated only with the vertices of the graph. Each vertex can be represented with a fixed amount of space.

2. Data for the vertices are stored in blocks, each of which can hold data for up to B vertices.

3. Data for a single vertex may be redundantly present in more than one block. Only one such block needs to be present in memory to satisfy the request.

4. The blocking algorithm must decide which data to store in which block before the search occurs, having no pre-knowledge of the search path.

5. The internal memory can hold data for at most M vertices, after which B elements will have to be flushed from memory to make room for a new block. In the weak model, those B elements must comprise a single block; in the strong model any B elements may be flushed from the memory.

6. Initially there are no vertices in internal memory.

7. The total number of vertices in the graph is $n = \rho M$, where $\rho \gg 1$.

8. We will traverse a path of length L, where L could exceed n.

Assumption 3 is reasonable for searching (a read-only traversal in the graph), where the data for the vertices are not changed, but would not be a reasonable assumption if we needed to perform updates on the graph, since then all copies of a vertex would have to be present in memory to service the request. If the total amount of storage used to represent the graph is S, we define the storage blow-up $s = S/(n/B)$. Intuitively, s is the average number of blocks containing each vertex in the graph. One of the major efforts of this research is to see what effect storage blow-up has on the efficiency of blocking algorithms, which in turn determines how much gain can be achieved by blocking for a read-only application versus an application that is required to update the data structures.

If a vertex is present in internal memory at least once, we will call the vertex *covered* (by analogy with graph pebbling); conversely those vertices present nowhere in memory are *uncovered*. We will use the term *page fault* to refer to an event where the path we are tracing through a graph extends to an uncovered vertex, and for which the searching algorithm therefore needs to read at least one block from the secondary memory. If a path generates a page fault only every σ steps (on average), we say that it achieves a

blocking speed-up of σ. We usually wish for σ to be an increasing function $\sigma(B)$. We will refer to the vertex currently being traversed at any time as the *pathfront*.

When a page fault causes a block to be brought into memory, it may be the case that it is necessary for it to overwrite data that are already in memory so as not to exceed the memory size. If the data are always flushed out by blocks, we say that were are using the *weak* paging strategy. A *strong* paging strategy would allow any B copies of vertices to be flushed, independent of whether they were originally in the same block. We count "copies of vertices" rather than the vertices themselves, since any vertex may be present multiple times in memory.

Notice that an upper bound on the blocking speed-up σ derives from theoretical considerations that apply no matter what the algorithm, and a lower bound on σ is an algorithm to achieve that speed-up. This is the reverse of the usual upper bound/lower bound situation encountered in previous chapters. An algorithm consists of two parts:

1. An assignment of vertices of the graph to blocks, known as the *blocking*. It is here that the storage blow-up is a consideration.

2. A *paging algorithm* to specify what vertices/blocks are in memory. It is the paging algorithm that determines whether we are in the strong or the weak model.

All algorithms presented in this paper are valid for the strong model. In this paper, all logarithms are base 2 unless otherwise specified.

7.3 Paging Strategies

It is not obvious given this model that blocking can ever be used efficiently. In fact, there is a graph for which an adversary can limit $\sigma(B) \leq 1$, independently of B. For example, consider K_{M+1}, the complete graph on $M+1$ vertices, with the path starting on any vertex. At any subsequent point, there will be at least one vertex that is not in memory. Extending the path to any such vertex will force a page fault.

It would seem that restricting the graphs to planar ones would prevent such a terrible case from occurring (at least for $M > 4$). However, as the next section shows, even this restriction does not help very much in the worst case.

We can distinguish between paging algorithms based on whether they are on-line or off-line. An *off-line* paging strategy may consider the entire path before deciding what blocks need to be brought in. An *on-line* paging strategy must base its decision on what block to bring in at any point only upon the previous history of the path (indeed it suffices to know only the pathfront). Note that in both cases, however, the blocks are fixed *a priori* without knowledge of the path. There is no distinction between an on-line and an off-line lazy strategy if $s = 1$. Permitting a paging strategy to be off-line and an unlimited storage blow-up allows for very efficient blocking.

Lemma 37 *There is a blocking and an off-line paging strategy that always achieves a speed-up of at least* B*, even if* $B = M$.

Proof: Let the blocking be $\mathcal{B} = \{\text{vertices of all paths of length } B-1\}$. Clearly the storage blow-up is large for this blocking. The off-line algorithm is simple: at any page fault, pick a block that contains all the vertices belonging to the next $B-1$ steps of the path. Thus, a page fault can occur at most every B steps. □

The interesting cases come, of course, with using an on-line paging strategy and with blockings with constant-degree storage blow-ups. We can term a paging algorithm as *lazy* if it only brings a block into memory that services an immediately preceding page fault. It is possible to show that we can restrict ourselves to lazy algorithms from the standpoint of proving upper bounds, at least if we are dealing in the weak model, as the following theorem shows.

Theorem 25 *For any blocking $\mathcal{B} = \{B_i\}$, there is an optimal on-line weak paging strategy that is lazy.*

Proof: We want to convert any on-line weak paging algorithm A to a corresponding strategy A' that is lazy, without increasing the total number of page faults. Let the path $P = (v_1, \ldots, v_L)$ be fixed. We define the set of blocks

$$\mathcal{M}_i = \{B_k \mid \text{algorithm } A \text{ has } B_k \text{ in memory when stepping to vertex } v_{i+1}\}.$$

$\mathcal{M}_i$ is a set of blocks; we will represent the set of vertices that is the union of all the blocks as $\cup\mathcal{M}_i$. We define $\mathcal{M}'_i$ similarly for algorithm A'. We will let our derived algorithm proceed in a lazy fashion, maintaining the invariant $\mathcal{M}'_i \subseteq \mathcal{M}_i$ for all $0 \leq i < L$. We consider three transactions that may take place by algorithm A in stepping from vertex v_i to v_{i+1}:

1. Some set of blocks $\mathcal{A}_i$ is flushed from memory.

2. Some block B_j may brought into memory as a result of a page fault. Let

$$\mathcal{F}_i = \begin{cases} \{B_j\} & \text{if } v_{i+1} \notin \cup\mathcal{M}_i \text{ and } v_{i+1} \in B_j \\ \emptyset & \text{if } v_{i+1} \notin \cup\mathcal{M}_i \end{cases}$$

   We maintain $|\mathcal{F}_i| \leq 1$; if more than one block contains a faulted vertex, we include only one of them.

3. Some other set of blocks $\mathcal{C}_i$ is read into memory. Some of these blocks may contain v_{i+q}.

By definition, $\mathcal{M}_{i+1} = (\mathcal{M}_i - \mathcal{A}_i) \cup \mathcal{F}_i \cup \mathcal{C}_i$. We are now ready to define how to form $\mathcal{M}'_{i+1}$.

1. If $v_{i+1} \in \cup(\mathcal{M}'_i - \mathcal{A}_i)$ then $\mathcal{M}'_{i+1} = \mathcal{M}'_i - \mathcal{A}_i$.

2. If $v_{i+1} \notin \cup(\mathcal{M}'_i - \mathcal{A}_i)$ then

   (a) If $\mathcal{F}_i \neq \emptyset$ then let $\mathcal{M}'_{i+1} = (\mathcal{M}'_i - \mathcal{A}_i) \cup \mathcal{F}_i$.

(b) If $\mathcal{F}_i = \emptyset$. In this case, algorithm A' will cause a page fault where algorithm A did not. A' has in memory (at least) all those blocks which A brought into memory for servicing page faults and has not subsequently flushed. It follows that $v_{i+1} \in B_k$ for some k for which $B_k \in \mathcal{C}_\ell$ for some $\ell \leq i$ and $B_k \notin \mathcal{A}_m$ for any $\ell < m \leq i+1$. Then we let $\mathcal{M}'_{i+1} = (\mathcal{M}'_i - \mathcal{A}_i) \cup \{B_k\}$. Set $\mathcal{C}_\ell = \mathcal{C}_\ell - \{B_k\}$.

It is easy to see that each of the possibilities preserves the invariant $\mathcal{M}'_{i+1} \subseteq \mathcal{M}_{i+1}$, thus guaranteeing that algorithm A' will never exceed available memory. We can also compare the number of pages read by A' at step $i+1$, $P_{i+1}(A')$ to those read by A at the same step, $P_{i+1}(A)$:

1. $P_{i+1}(A') = 0 \leq P_{i+1}(A)$.
2. For the two subcases:
   (a) $P_{i+1}(A') = 1 \leq P_{i+1}(A)$.
   (b) $P_{i+1}(A') = 1$. In this case, $P_{i+1}(A')$ could be higher than $P_{i+1}(A)$. However, then we must have had the case that $P_\ell(A') \leq P_\ell(A) - 1$, since $|\mathcal{C}_\ell| \geq 1$, so that $P_{i+1}(A') + P_\ell(A') \leq P_{i+1}(A) + P_\ell(A)$. We reduced the cardinality of $\mathcal{C}_\ell$ by one to compensate, so that $P_{i+1}(A') + P_\ell(A') \leq P_{i+1}(A) + P_\ell(A)$.

Thus, algorithm A' needs no more reads than algorithm A, but has operated throughout in a lazy fashion. □

It is possible that this theorem could be extended to the strong model by considering the multi-set of vertices present in memory instead of the set of blocks and proceeding similarly. The extension is not straightforward, however, since it is not so easy to decrement the cardinality of $\mathcal{C}_\ell$, since the block that read in the faulted vertex back in step ℓ may have had many of its vertices subsequently flushed, so that it is difficult to argue that $P_{i+1}(A') + P_\ell(A') \leq P_{i+1}(A) + P_\ell(A)$.

7.4 Searching in general graphs

We here consider the problem of searching in general graphs. First we must start with some definitions. Consider we have a connected graph $G = (V, E)$ such that $|V| = n = \rho M$, where $\rho > 1$.

Definition 4 Let $v \in N \subset V$. Then we say that N is a *neighborhood* of v. Note that N is not necessarily connected. Let $\mathcal{N}_v = \{\text{all neighborhoods of } v\}$. If $N \in \mathcal{N}_v$ and $|N| = k$, then we say N is a *k-neighborhood* of v. Let $\mathcal{N}_v(k) = \{\text{all } k\text{-neighborhoods of } v\}$.

Definition 5 Let $N \in \mathcal{N}_v$. Then $b(v, N)$, the *break-out distance* of v in N, is defined to be the minimum distance from v to any point not in N, i.e.,

$$b(v, N) = \min_{v' \notin N} d(v, v'),$$

where $d(v, v')$ is the minimum path length from v to v'.

Definition 6 We call any $N \in \mathcal{N}_v(k)$ for which $b(v, N)$ is a maximum over all k-neighborhoods of v a *compact k-neighborhood*. If N is a compact k-neighborhood of v, then we define $r_v(k) = b(v, N)$ to be the *k-radius of v*. We let $\mathcal{N}_v^*(k)$ denote the set of all compact k-neighborhoods of v.

We can show a simple lemma concerning compact neighborhoods.

Lemma 38 *Assume we have a graph $G = (V, E)$. At least one compact k-neighborhood of vertex v is connected for any vertex $v \in V$.*

Proof: Consider the sets $W = \{v' \in V \mid d(v, v') \leq r_v(k)\}$ and $W' = \{v' \in V \mid d(v, v') \leq r_v(k)\}$ and consider any $V' \in \mathcal{N}_v^*(k)$. By definition of $r_v(k)$, $|W| \leq S < |W'|$. Since V is a compact k-neighborhood, it must be the case that $W \subseteq V'$. It can be shown by simple induction on k that W and W' are connected. So any set V' such that $W \subseteq V' \subset W'$ will be connected, since $V' - W$ will comprise only vertices connected to some vertex in W and W was connected. There is some such V' for which $|V'| = k$, which will be a connected, compact k-neighborhood of v. □

Definition 7 We can define two k-radii for graphs, then *minimum k-radius* and the *maximum k-radius* respectively to be

$$r^-(k) = \min_{v \in V} r_v(k)$$

and

$$r^+(k) = \max_{v \in V} r_v(k).$$

Definition 8 Any class of graphs for which the ratio $r^+(k)/r^-(k)$ is bounded by a constant for some k is called a *k-uniform class of graphs*.

Clearly, for any particular graph, the ratio between the upper and lower k-radius is constant, so the uniformity definition applies only to an infinite class of graphs.

We start with a few examples of k-radii, followed by a few simple lemmas.

Example 1 We can compute the k-radii of complete d-ary trees. It is simple to see that the root has $(d^{r+1} - 1)/(d - 1)$ within a distance of r (assuming that the height of the tree is at least r). This fact leads to

$$r_{\text{root}}(k) = \frac{\log\left(k(d-1)+1\right)}{\log d} - 1.$$

Similarly, a vertex that is at least a distance r from both the root and the leaves has radius

$$r_{\text{int}}(k) = \frac{\log\left(k(d-1)+2\right) - \log(d+1)}{\log d}.$$

The leaves are a little harder to analyze, but each leaf of a tree with height at least r has

$$\frac{d^{\lfloor r/2 \rfloor + 1} + d^{\lceil r/2 \rceil} - 2}{d - 1}$$

vertices within a distance r of it. This leads to a radius

$$\begin{aligned} r_{\text{leaf}}(k) &= \min \left\{ 2 \left\lceil \frac{\log\left(k(d-1)+2\right) - 1}{\log d} - \frac{1}{2} \right\rceil, \right. \\ &\qquad \left. 2 \left\lceil \frac{\log\left(\left(k(d+1)+2\right)/d - 1\right) - 1}{\log d} \right\rceil + 1 \right\}. \end{aligned}$$

Comparing these values, it follows that for complete d-ary trees, $r^-(k) = r_{\text{int}}(k)$ and $r^+(k) = r_{\text{leaf}}(k)$. Those internal vertices that are within a distance r of the root have a number of vertices within a distance r of them that is intermediate between those of the internal vertices and the root; the radii are correspondingly intermediate when k becomes large enough. A similar fact holds for those internal vertices that are close to the leaves. □

Lemma 39 *The set of all complete d-ary trees is a uniform class of graphs.*

Proof: The minimum and maximum radii are always within about a factor of 2. □

Example 2 We can also compute the leading term of the radii for grid graphs. See Section 7.6 for a definition of grid graphs. We first note that all the vertices in a d-dimensional grid graph have the same radius $r_d(k)$ for any value k; consequently grid graphs comprise a uniform class of graphs. We start by computing the number of vertices $k_d(r)$ within a distance r of any vertex in a d-dimensional grid graph; the radii are found by inverting this function to get r as a function of k. The problem of computing how large the neighborhoods of radius r in d-dimensional grid graphs was considered by Rosenberg [Rosa], where he derived the answer $O(r^d)$. In the following derivation, we find the exact coefficient of the r^d term.

It is immediately apparent that in one dimension, there are $2r + 1$ vertices within a distance r of any vertex; in other words, $k_1(r) = 2r + 1$. We will only be able to compute the leading term, so we will consider the functions $\tilde{k}_d(r)$ that are the same as $k_d(r)$ with all but the leading terms removed. Thus, $\tilde{k}_1(r) = 2r$. We observe that

$$k_d(r) = k_{d-1}(r) + 2 \sum_{0 \le r' < r} k_{d-1}(r'),$$

or equivalently

$$\tilde{k}_d(r) = 2 \sum_{0 \le r' < r} \tilde{k}_{d-1}(r').$$

It is also clear that $\tilde{k}_d(r) = c_d r^d$ for some c_d. We can compute the recurrence for c_d as follows:

$$\begin{aligned} c_d &= \frac{2}{d} c_{d-1} \\ c_1 &= 2 \end{aligned}$$

using the fact that

$$\sum_{0 \le i < n} i^d = \frac{1}{d+1} n^{d+1} + O(n^d).$$

We can solve the recurrence for c_d to show that

$$\tilde{k}_d(r) = \frac{2^d}{d!} r^d.$$

Inverting this formula, we arrive at our radius:

$$r_d(k) = \frac{1}{2} (d!\, k)^{1/d} + o(k^{1/d}).$$

We can use Stirling's formula to approximate the $d!$ term, giving the result

$$r_d(k) \sim \frac{1}{2e} (2\pi d)^{1/2d}\, d k^{1/d}.$$

Finally, notice that $(2\pi d)^{1/2d}$ converges to 1 as $d \to \infty$ (and is never larger than about 2.5), so that

$$r_d(k) \sim \frac{1}{2e}\, d k^{1/d}.$$

Since all the vertices have the same radius, we get

$$r_d^+(k) = r_d^-(k) \sim \frac{1}{2e}\, d k^{1/d}. \tag{7.1}$$

□

Lemma 40 *For any graph $G = (V, E)$, if $k \le k'$, then*

1. *$r_v(k) \le r_v(k')$ for any $v \in V$,*
2. *$r^+(k) \le r^+(k')$, and*
3. *$r^-(k) \le r^-(k')$.*

Proof: Assume that $k \le k'$.

(1) Let N be a compact k-neighborhood of v. Let N' be N plus any $(k' - k)$ other vertices. Then N' is a k'-neighborhood with radius at least $r_v(k)$.

(2) Let v be a vertex such that $r_v(k') = r^-(k')$. Then we have

$$r^-(k) \le r_v(k) \le r_v(k') = r^-(k').$$

(3) Let v be a vertex such that $r_v(k) = r^+(k)$. Then

$$r^+(k) = r_v(k) \le r_v(k') \le r^+(k').$$

□

Figure 7.1: Vertex w will have a larger k-radius than $r^+(k)$.

The next lemma states that if we increase a compact j-neighborhood of any vertex by adding k additional vertices, we cannot increase the radius by more than $2r^+(k)$.

Lemma 41 *For any vertex v, $r_v(j+k) \leq r_v(j) + 2r^+(k)$.*

Proof: Figure 7.1 illustrates this proof. Assume by way of contradiction that for some j, k, and v, we have $r_v(j+k) > r_v(j) + 2r^+(k)$. We can find two compact neighborhoods N_j and N_{j+k} consisting of j and $j+k$ vertices, respectively, with $N_j \subset N_{j+k}$. By definition of the radius, those vertices on the border of N_j (i.e., those vertices v' such that $b(v', N_j) = 1$) have $b(v', N_{j+k}) > 2r^+(k) + 1$. Choose that vertex v' such that $b(v', N_{j+k})$ is minimized and consider a path that realizes the break-out distance. This path will have length more than $2r^+(k) + 1$. Consider the vertex w that is the midpoint of the path. Define $N = N_{j+k} - N_j$; N is a k-neighborhood of w. From vertex w, it takes more than $r^+(k)$ steps to reach any vertex in N_j; likewise it takes more than $r^+(k)$ steps to reach any vertex in $G - N_{j+k}$. It follows that $b(w, N) > r^+(k)$, contradicting the definition of $r^+(k)$. Hence there can be no such j, k, and v. □

The previous lemma leads to the following important result.

Lemma 42 *Assume that $k \leq k'$. Then*

$$r^+(k') \leq \left(2\frac{k'}{k} + 2\right) r^+(k).$$

Proof: Let v be some vertex with radius $r_v(k') = r^+(k')$ and let $d = \lceil k'/k \rceil$. By Lemma 41, if we start from some compact k-neighborhood of v, we can increase the

radius by at most $2r^+(k)$ for every additional k vertices we add to it. It follows that

$$r^+(k') = r_v(k') \leq r_v(k) + 2dr^+(k) \leq r^+(k) + 2dr^+(k) \leq \left(2\frac{k'}{k} + 2\right) r^+(k).$$

□

7.4.1 Upper bounds

Armed with the above definitions and lemmas, we are ready to prove some upper bounds. The first few upper bounds are trivial.

Lemma 43 *The fastest worst-case speed-up that could be achieved in any graph is* $\sigma \leq r^+(M)$.

Proof: Let the vertices in memory be some k-neighborhood of the pathfront just after a page fault has been serviced, for some $k \leq M$. By definition of $r^+(M)$, there will always be some vertex within that distance of the pathfront that is not in memory, so it can take at most that many steps to cause the next page fault. Causing the next page fault leaves the same configuration as we assumed, so this process can be continued indefinitely. □

A second lemma is like this first one.

Lemma 44 *The fastest worst-case speed-up that could be achieved in any graph is* $\sigma \leq 2r^-(M)$.

Proof: Let v be a vertex such that $r_v(M) = r^-(M)$ (there must be at least one such vertex by the definition of minimum M-radius), and let $W = \{v' \in V \mid d(v, v') \leq r^-(M)\}$. By definition, $|W| > M$ and the distance between any two points $w, w' \in W$ is

$$d(w, w') \leq d(w, v) + d(v, w') \leq 2r^-(M).$$

Then as long as the pathfront is somewhere in W, it is always possible to extend the path to a vertex in W that is not in memory in fewer than $2r^-(M)$ steps, no matter which M vertices are in memory. We can begin at vertex v to establish the initial condition. □

These lemmas correctly handle the example where a vertex is attached to M other vertices; $r^+(M) = 1$ in this case and as we found before, no speed-up was possible regardless of the block size.

The difficulty with these two lemmas is that they are independent of the block size. Intuitively, it would seem that by using a block size of B, the fastest possible speed-up would be B or at least $O(B)$. The next lemma proves such a bound.

Lemma 45 *The fastest worst-case speed-up that could be achieved in a graph containing* ρM *vertices,* $\rho > 1$, *is*

$$\sigma \leq 2\frac{\rho}{\rho - 1}B.$$

Proof: Assume that we are anywhere in the graph and that there are at most M vertices in memory. We can walk through the entire graph using exactly $2\rho M$ steps by doing an Eulerian tour of the graph resulting from doubling all the edges of some spanning tree; this tour will leave us back at the same node as we started. During this walk, we are guaranteed to cause at least $(\rho - 1)M/B$ page faults. □

This upper bound reduces to $2B$ as we let $\rho \to \infty$. In the special case of graphs containing a Hamiltonian cycle, we can clearly get an upper bound of B by having the adversary continually follow the Hamiltonian cycle.

We would like to prove an even stronger upper bound, specifically something that is proportional to $r^+(B) \leq B$. The next lemma gives a result along this line.

Lemma 46 *The fastest worst-case speed-up that could be achieved for any graph is* $\sigma \leq \left(2\frac{M}{B} + 2\right) r^+(B)$.

Proof: From Lemma 43, we know that $r^+(M)$ is an upper bound. But from Lemma 42, $r^+(M) \leq \left(2\frac{M}{B} + 2\right) r^+(B)$. □

In particular, the previous lemma shows that we have a bound that is proportional to $r^+(B)$ as long as we use blocks that grow as a constant fraction of the memory size—in other words, as long as we have large blocks. As the block sizes become smaller, it becomes much more difficult to get an upper bound proportional to $r^+(B)$. The following lemma gives such a bound with some restrictions on the blocking. We later generalize the lemma to remove any such restrictions.

Before presenting the lemma, however, we want to make some observations about hypergraphs [Ber]. First, a hypergraph is a pair $G = (V, \mathcal{H})$, where V is a set of vertices and $\mathcal{H}$ is a set of subsets of V called *edges*. If each set $H \in \mathcal{H}$ has cardinality 2, then G is an ordinary graph. If $|H| = r$ for all $H \in \mathcal{H}$, then we say that the hypergraph is *r-regular*. Let $S \subset V$. Then if for every $H \in \mathcal{H}$ we have $|H \cap S| \leq 1$, then we say that S is a *strongly stable set of vertices*. Notice that blocking the vertices of the graph in which we wish to search induces a B-regular hypergraph, where each of the blocks is an edge of the hypergraph. The next lemma says that we can get an upper bound proportional to $r^+(B)$ as long as we can find a strongly stable set of vertices of cardinality at least $\lceil N/B \rceil$.

Lemma 47 *Assume that there are $\lceil N/B \rceil$ vertices such that there is no block containing any two of them. Then the blocking speed-up satisfies*

$$\sigma(B) \leq 8r^+(B)$$

as $N/M \to \infty$.

Proof: Define the *ball of radius r around vertex v* to be $\{v' \in V \mid d(v, v') \leq r\}$, denoted by $K_v(r)$. Define a *close packing* of balls of radius r in a graph to be a packing of balls of radius r such that each ball is touching at least one other ball (assuming there is

Figure 7.2: Creating a spanning tree visiting all the "must-visit" vertices.

more than one ball in the packing). Two balls are touching if the distance between their centers is $2r$. Choose any maximal close packing of balls of radius $r^+(B)$ in the graph. Connect the centers of any touching balls together via paths of length $2r^+(B)$ and create a spanning tree of the resulting subgraph, as shown in Figure 7.2. This spanning tree is called the *skeletal Steiner tree*. Since each ball of radius $r^+(B)$ by definition contains at least B vertices of the graph, there are at most $\lfloor N/B \rfloor$ balls in the maximal close packing. Hence, the total length of this skeletal Steiner tree is no more than $2r^+(B)\,(\lfloor N/B \rfloor - 1)$.

We claim that every vertex in the graph is within a distance $2r^+(B)$ of the skeletal Steiner tree. Assume that some point exists whose distance to the skeletal Steiner tree is greater than $2r^+(B)$. Then it must likewise have a distance at least $2r^+(B)$ from the centers of each of the balls, and consequently there exists some vertex with distance exactly $2r^+(B)$ from some ball whose ball of radius $r^+(B)$ does not overlap any of the other balls, contradicting the assumption that the close packing was maximal.

Now choose $\lceil N/B \rceil$ strongly stable vertices. We call these vertices the "must-visit" vertices, since by visiting all of them, we guarantee that we will cause at least $\lceil N/B \rceil - M$ page faults. This set of strongly stable vertices was assumed to exist in the lemma. Attach each of these vertices to the skeletal tree using a path whose length is no more than $2r^+(B)$ to form an augmented Steiner tree of total length no more than

$$2r^+(B) \left\lceil \frac{N}{B} \right\rceil + 2r^+(B) \left(\left\lfloor \frac{N}{B} \right\rfloor - 1 \right) < 4r^+(B) \left\lceil \frac{N}{B} \right\rceil .$$

Traversing this augmented Steiner tree using an Eulerian tour is guaranteed to touch on $\lceil N/B \rceil$ distinct blocks, with a total path length of no more than $8r^+(B)\lceil N/B \rceil$.

Thus, we can have a blocking speed-up no more than

$$\frac{8r^+(B)\lceil N/B\rceil}{\lceil N/B\rceil - M} \rightarrow 8r^+(B)$$

as $N/M \rightarrow \infty$, since $M \geq B$ and therefore $N/B \rightarrow \infty$ as $N/M \rightarrow \infty$. □

This lemma has a simple corollary in the case that $s = 1$.

Corollary 4 *If the storage blow-up s is 1, then the blocking speed-up σ is less than $8r^+(B)$.*

Proof: There are at least $\lceil N/B\rceil$ blocks required to cover the graph. Pick a representative of each of these blocks as the "must-visit" vertices. These vertices are guaranteed to satisfy the strongly-stable condition of Lemma 47. □

In the case $s = 1$, choosing and connecting the $\lceil N/B\rceil$ "must-visit" vertices is an instance of what is described in the literature at the *group Steiner tree* problem. In the group Steiner tree problem, the input consists of several sets of vertices, and the goal is to find a Steiner tree that touches at least one representative of each set. Reich and Widmayer gave an approximation algorithm for the minimum group Steiner tree problem, but did not give any guarantees on how much worse than optimal their results could be [ReW]. Ihler has shown the problem to be NP-hard, even if the graph is a tree, and gave a trivial algorithm to approximate the optimal tree within a factor of $g - 1$, where g is the number of groups [Ihl].

The problem with applying Lemma 47 to blockings where there are not $\lceil N/B\rceil$ strongly stable vertices is that we are not guaranteed that each of the "must-visit" vertices will cause a page fault. Choosing a static set of "must-visit" vertices allows the paging algorithm to possibly satisfy several of the requests with a single block. In the extreme case where every set of B vertices comprises a block, the paging algorithm might always bring in the next B "must-visit" vertices, achieving a blocking speed-up of $Br^+(B)$. The following lemma is a generalization to Lemma 47 that gets around this problem by choosing the "must-visit" vertices dynamically.

Lemma 48 *The fastest worst-case speed-up that could be attained in any graph is*

$$\sigma \leq 8r^+(B).$$

Proof: We construct a skeletal Steiner tree in the graph identically to how we did in the proof for Lemma 47. However, instead of picking a fixed set of $\lceil N/B\rceil$ "must-visit" vertices, we instead consider all the vertices in the graph as potential "must-visit" vertices. We show that every time we extend our path by $8\lceil N/B\rceil r^+(B)$, we will cause at least $\lceil (N - M)/B\rceil$ page faults, giving us the desired upper bound.

We are going to order all the vertices in the graph. We start by associating with each vertex of the skeletal Steiner tree a group consisting of all the vertices in the graph that are closer to it than to any other vertices of the skeletal Steiner tree. Vertices that are equidistant from several vertices of the skeletal Steiner tree are arbitrarily assigned

to the group of one of them. We call the vertex on the skeletal Steiner tree the parent of its group. Each group has at least one vertex in it: its parent. We now pick some vertex of the skeletal Steiner tree as our distinguished start vertex, and assign it the number 1. We number any other vertices in its group in arbitrary order starting at 2. We complete the numbering by doing an Eulerian tour of the skeletal Steiner tree, numbering all the vertices of any previously unnumbered groups as we come across the group's parent node in the skeletal Steiner tree.

Assume for the moment that none of the vertices is in memory and that we start our path at the start vertex of the graph, vertex 1. In response to the request at vertex 1, the paging algorithm will generate a page fault, bringing in some block of B vertices. We then consider the next "must-visit" vertex to be the first vertex according to the ordering that is not in memory (call it v). As in the proof to Lemma 47, we visit v by going along the skeletal Steiner tree in Eulerian order until we get to its parent node and then taking a shortest path to v. After causing the page fault at v, we find the next "must-visit" vertex (call it w) in the same way, extend the path to w by going back to the skeletal Steiner tree reversing the path that brought us to v, taking the Eulerian path in the skeletal Steiner tree to the parent of w, and taking a shortest path to w. We continue this process until we have caused $\lceil N/B \rceil$ page faults. At this point we will not have completed the tour, since we have N potential "must-visit" vertices, but the paging algorithm can only bring in B at a time, so that we can always cause $\lceil N/B \rceil$ page faults. Complete the tour by considering vertex 1 as a dummy "must-visit" node and extending the path appropriately. This process creates an augmented Steiner tree among $\lceil N/B \rceil$ dynamically chosen "must-visit" vertices, with total path length no more than $8r^+(B)$, just as in the proof of Lemma 47. If there are vertices in memory, we can create the augmented Steiner tree in the same fashion, except that we can only guarantee $\lceil (N-M)/B \rceil$ "must-visit" vertices in the augmented Steiner tree. We can obviously continue cycling through the vertices indefinitely, choosing a different set of "must-visit" vertices at each pass. Thus, the overall blocking speed-up can be no more than

$$\sigma(B) \leq \frac{8r^+(B)\lceil N/B \rceil}{\lceil (N-M)/B \rceil},$$

which gives the desired result as $N/M \to \infty$. □

The previous lemmas lead up to our upper bound theorem.

Theorem 26 *The fastest worst-case speed-up that could be attained in any graph containing ρM vertices, $\rho > B$, is*

$$\sigma \leq \min\left\{ r^+(M), 2r^-(M), 2\frac{\rho}{\rho-1}B, \left(2\frac{M}{B}+2\right) r^+(B), 8r^+(B) \right\}.$$

Proof: This theorem follows directly from Lemmas 43 to 48. □

7.4.2 Lower bounds

If we do not care about the storage blow-up, we can easily obtain a lower bound for general graphs.

Lemma 49 *There is a blocking and an on-line paging algorithm that achieves a worst-case speed-up of $r^-(B)$ with average storage blow-up $s = B$ as long as $M \geq B$.*

Proof: Let the blocking $\mathcal{B}$ consist of one representative of $\mathcal{N}_v^*(B)$ for each vertex v in the graph. When a page fault occurs on some vertex w, we bring in the representative of $N_w^*(B)$, replacing whatever else is in memory. By the definition of the radius, it will take at least $r_w(B) \geq r^-(B)$ steps to cause another page fault.

We can easily compute the storage blow-up of the blocking. The blocking has one block of size B for every vertex in the graph; thus, on average, each vertex is represented B times. However, there is no bound on the maximum number of times a vertex may be represented. □

It is natural to ask to what extent we can reduce the storage blow-up in Lemma 49. It seems excessive to center a block on every vertex. It is indeed possible to reduce the storage blow-up somewhat. First, let us define a problem called BALL COVER(r).

- Problem: BALL COVER(r)
- Input: Connected graph $G = (V, E)$ and distance r
- Output: A subset $V' \subseteq V$ of minimal cardinality such that for all $v \in V$ there exists a $v' \in V'$ such that $d(v, v') \leq r$

The problem of BALL COVER(r) is to pack a minimum number of balls of radius r into the graph in such a way that the entire graph is covered.

We can show a simple lemma relating BALL COVER(1) to VERTEX COVER. Recall that VERTEX COVER is the problem of finding a minimum cardinality subset $V' \subseteq V$ such that for every edge $e = (u, v) \in E$, either $u \in V'$ or $v \in V'$.

Lemma 50 *For a connected graph, any vertex cover is a ball cover(1).*

Proof: Let V' be a solution to VERTEX COVER. Assume by way of contradiction that there is some vertex v such that $d(v, v') \geq 2$ for all $v' \in V'$. Since G is connected, v is connected to some vertex u. $u \notin V'$ by assumption since $d(v, u) = 1$. But then the edge (v, u) has no representative in V', so V' was not a vertex cover. □

VERTEX COVER is an NP-complete problem [GaJ], but there is a well-known polynomial algorithm that approximates the optimal vertex cover within a factor of 2. Take any maximal matching of G. A maximal matching is a set of vertex-disjoint edges in a graph such that no additional edges can be added to it that will still be vertex-disjoint. Let $V' = \{$all endpoints in the maximal matching$\}$. V' is a vertex cover, since if any edge had neither endpoint in V, we could add it to the matching, implying that the matching was not maximal. It approximates the optimal vertex cover within a factor of two, since at least one endpoint of every edge in the maximal matching must be present in the vertex cover. Unfortunately, the upper bound on the cardinality of V' is n.

We can use the idea to give upper bounds on the maximum cardinality subsets of BALL COVER(r).

Lemma 51 *There is a solution V' to BALL COVER(2) such that $|V'| \leq \lfloor \frac{n}{2} \rfloor$ whenever $n \geq 2$.*

Proof: Find a maximal matching of G and let V' consist of one endpoint of every edge in the matching. If the matching is empty, the graph must consist of a single point. If the matching is non-empty, any vertex that is a member of some matching is within a distance 1 of some $v' \in V'$. Likewise, any vertex that is adjacent to a vertex in the matching is within a distance 2 of some $v' \in V'$. Therefore, any vertex v that is not within a distance 2 of some $v' \in V'$ must have a distance at least 2 from any vertex in the matching. But then there is a vertex u adjacent to v that has a distance 1 from any matching, so (v, u) could be added to the matching, contradicting our assumption that the matching was maximal. Hence, V' is a ball cover(2). The matching can use at most n vertices, of which only one half are in V', so $|V'| \leq \frac{n}{2}$. □

In finding a ball cover(1), we packed as many graphs of the form ●—● as we could, where a filled circle denotes a vertex that we include in V'. When searching for a ball cover(2), we packed subgraphs of the form ●—○. What happens of we pack subgraphs of the form ○—●—○?

Lemma 52 *There is a solution V' to BALL COVER(3) such that $|V'| \leq \lfloor \frac{n}{3} \rfloor$ as long as $n \geq 3$.*

Proof: Find a maximal packing of graphs of the form ○—●—○, where we include the middle vertex in V'. If the packing is empty, the graph has only two vertices in it (either of which is a ball cover(3)). Assume that V' is non-empty. To show that V' is a ball cover(3), consider the fact that any vertex within one of the packed subgraphs must have a distance at most 1 from a vertex in V'. Thus, if there were a vertex v that had a distance of 4 from any vertex in V', then it must have a distance at least 3 from any vertex in the packing. But if that were the case, then it would be adjacent to a vertex u of distance 2 which is adjacent to a vertex w of distance 1 from any vertex in the packing. But then the path (v, u, w) could have been added to the packing (and u added to V'), contradiction the assumption that the packing was maximal. □

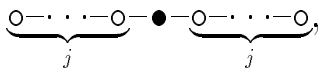
We can summarize the results above in the following table:

r	|BALL COVER(r)|
1	n
2	$\lfloor \frac{n}{2} \rfloor$
3	$\lfloor \frac{n}{3} \rfloor$

It is tempting to extrapolate the table to say that $|\text{BALL COVER}(r)| \leq \lfloor \frac{n}{r} \rfloor$. Unfortunately, the above construction does not generalize to $n = 4$. However, we can extend the method to show the following theorem.

Theorem 27 *Let $j \in \mathbf{Z}^+$. Then there is a solution V' to BALL COVER($3j$) such that $|V'| \leq \left\lfloor \frac{n}{2j+1} \right\rfloor$ whenever $n \geq 2j + 1$.*

Proof: Consider a maximal packing of subgraphs of the form



choosing only the centermost point to be in V'. If the packing is empty, then there are no two points in the graph that are separated by a distance of more than $2j+1 \leq 3j$, so any single vertex will comprise a ball cover($3j$); the cardinality, 1, will be no more than $n/(2j+1)$ whenever there are at least $2j+1$ vertices in the graph. Now consider the case where the packing is not empty. (By definition, there are at least $2j+1$ vertices in the graph.) We claim that V' is a ball cover($3j$). Assume that there is some vertex v that has a distance (without loss of generality) of $3j+1$ from any vertex in V'. Every vertex in one of the packed subgraphs is within a distance j of some point in V', so vertex v must have a distance that is at least $2j+1$ from any vertex in the packing. But then there is a path of length $2j+1$ that does not include vertices of any matching, contradicting the assumption that the packing was maximal. Thus, V' is a ball cover($3j$). It is evident by inspection that $|V'| \leq \lfloor n/(2j+1) \rfloor$. □

What about BALL COVER($2j+1$) and BALL COVER($2j+2$)? We can extend Theorem 27 by the simple observation that any ball cover(r) is also a ball cover($r+1$).

Corollary 5 *There is a solution V' to BALL COVER(r) such that $|V'| \leq \frac{n}{2\lfloor r/3 \rfloor + 1}$.*

Proof: Choose $j = \lfloor r/3 \rfloor$ and apply Theorem 27. Every vertex is within a distance $3\lfloor r/3 \rfloor \leq r$ of some vertex of V' and the cardinality $|V'|$ is exactly the given expression. □

We are now ready to reduce the storage blow-up needed to attain a blocking speed-up of $cr^-(B)$.

Theorem 28 *There is a blocking and an on-line paging algorithm that achieves a speed-up $\sigma \geq \lceil r^-(B)/2 \rceil$ with asymptotically $s = 3B/r^-(B)$ as long as $M \geq B$.*

Proof: For compactness of notation, let $r = r^-(B)$. Construct a set V' to be a ball cover($\lfloor r/2 \rfloor$), and choose as our blocking a representative of $\mathcal{N}_v^*(B)$ for every $v \in V'$. Recall that a compact B-neighborhood will have radius at least r. When a page fault occurs at some vertex v, we know that there is some $v' \in V'$ such that $d(v,v') \leq r/2$. Bring in the block associated with v', $\mathrm{B}_{v'}$, replacing whatever else is in memory. The block contains every vertex within a distance $r_{v'}(B) - 1 \geq r - 1$ of v'. Consider any vertex w not within the block $\mathrm{B}_{v'}$, as illustrated in Figure 7.3. By the triangle inequality, we know that

$$d(v',w) \leq d(v,w) + d(v,v'),$$

or

$$d(v,w) \geq d(v',w) - d(v,v').$$

Figure 7.3: The distance between vertex v and vertex w must be at least $\lceil \frac{r}{2} \rceil$.

Furthermore, $d(v', w) \geq r$ and $d(v, v') \leq \lfloor r/2 \rfloor$. Consequently it will take at least $\lceil r^-(B)/2 \rceil$ steps to cause an additional page fault from v. The memory needs clearly never exceed B.

To show the storage blow-up, notice that as long as the graph has at least $\lfloor r/2 \rfloor$ vertices in it, the number of vertices in V' is

$$|V'| \leq \frac{n}{2\lfloor r/6 \rfloor + 1}.$$

Since each block contains B vertices, each vertex is represented an average of

$$s \leq \frac{B}{n} \frac{n}{2\lfloor r/6 \rfloor + 1}$$

times, giving the result. □

Notice that the lower bounds give us tight upper and lower bounds on the blocking speed-up in uniform classes of graphs, since the upper bound we have achieved is proportional to $r^+(B)$ and the lower bound is proportional to $r^-(B)$, which are within a constant factor of each other for uniform classes of graphs. We can give a different solution to the BALL COVER problem that works out better for grid graphs in terms of storage blow-up, but not quite as well for complete trees. First, we define something that is as close as we can come to the inverse functions for $r^+(k)$ and $r^-(k)$.

Definition 9 Let $k_v(r) = |K_v(r)|$, the cardinality of the ball of radius r around vertex v. We call $k_v(r)$ the *volume* of $K_v(r)$. Then we define

$$\begin{aligned} k^+(r) &= \max_{v \in V} k_v(r) \\ k^-(r) &= \min_{v \in V} k_v(r) \end{aligned}$$

Now we can show our other bound for BALL COVER.

Theorem 29 *Let $G = (V, E)$ and radius r be given. Then there exists a solution V' to BALL COVER(r) with cardinality*

$$|V'| \leq \frac{n}{k^-(\lfloor r/2 \rfloor)}.$$

Proof: Consider a maximal packing of G with balls of radius $\lfloor r/2 \rfloor$ and include in V' the center point of all these balls. We claim that every vertex v is within a distance r of some $v' \in V'$, i.e., that V' is a ball cover(r). Assume by way of contradiction that there is a point v that is more than a distance r from all the vertices, so that $d(v, v') \geq r + 1$ for all $v' \in V'$. Then each of the neighbors $w_{1,i}$ of v has $d(w_{1,i}, v') \geq r$ for all $v' \in V'$. Similarly, those vertices of distance 2 from v have distance $d(w_{2,i}, v') \geq r - 1$, and so on down to $d(w_{\lfloor r/w \rfloor,i}, v') \geq \lceil r/2 \rceil + 1 > \lfloor r/2 \rfloor$. But then, we could include $K_v(\lfloor r/2 \rfloor)$ in the packing, contradicting the assumption that the packing was maximal. Hence, V' is a ball cover(r). To show the cardinality, simply notice that each of the balls of radius $\lfloor r/2 \rfloor$ removes at least $k^-(\lfloor r/2 \rfloor)$ vertices from further consideration in the packing; at most

$$\frac{n}{k^-(\lfloor r/2 \rfloor)}$$

such balls could be removed. □

Now we can do a new version of Theorem 28.

Theorem 30 *There is a blocking and an on-line paging algorithm that achieves a speed-up of $\lceil r^-(B)/2 \rceil$ with $s \leq B/k^-(\lfloor r^-(B)/4 \rfloor)$.*

Proof: As in the proof to Theorem 28, construct V' to be a ball cover($\lfloor r^-(B)/2 \rfloor$) and pick a compact B-neighborhood around each $v \in V'$. The same argument shows that the speed-up is at least $\lceil r^-(B)/2 \rceil$. By Theorem 29, the cardinality of V' need be no more than

$$|V'| \leq \frac{n}{k^-\left(\left\lfloor \frac{r^-(B)}{4} \right\rfloor\right)}.$$

□

In the case of d-dimensional grid graphs, this theorem results in a storage blow-up of 4^d. It might be possible to reduce the base of the exponent by using a different argument, but the fact that we need to consider neighborhoods of radius $r/2$ to get a ball cover of radius r means that the result will be exponential.

We do not have any lower bound results for general graphs that have constant storage blow-up. We hope to get something with s = constant that will be proportional to $r^-(B)$. It is likely that when we disallow storage blow-up that we will not be able to obtain such good results; a general proof technique for showing upper bounds in the absence of storage blow-up is not known.

In the next two sections, we consider special-case arguments for complete trees and grid graphs.

7.5 Searching in Complete Trees

First we will consider upper bounds on the blocking speed-up in complete d-ary trees. After showing the upper bounds, we will give algorithms to achieve them within a constant factor with a small constant storage blow-up.

7.5.1 Upper bounds

If we allow the arity of the tree to be arbitrarily large, we immediately run into problems where we cannot use blocking efficiently. Consider, for example, a graph in which some vertex v is connected to M other vertices. It is possible to cause (at least) one page fault by extending the path to v and then going to any of the other vertices that are not in memory. There must be at least one such vertex. Thus, we can generate a page fault with every other vertex for a worst-case blocking speed-up $\sigma(B) \leq 2$ independently of B.

We can generalize this argument to complete d-ary trees to give a result that is about a factor of 8 better than the general $8r^+(B)$ result of Lemma 48.

Theorem 31 *In a complete d-ary tree of height h, the maximum worst-case speed-up that could be guaranteed is*

$$\sigma(B) \leq \frac{2h}{\frac{h}{\log_d B} - \log_d M}. \tag{7.2}$$

Proof: Consider an adversary who is trying to cause as many page faults as possible. Assume that the pathfront is at the root, with some M vertices in memory. Thereafter, the adversary will walk down the tree, taking at each level the branch that goes to the closest vertex that is not in memory. Since at most $\frac{d^{r+1}-1}{d-1}$ vertices can be within a distance r of the pathfront, it is guaranteed that a page fault will be generated at least every $\log_d B$ steps in regions that were previously not in memory. The fact that there were M items in memory means that the path could traverse up to $\log_d M$ of these, for a total of at most $\log_d M$ page faults not caused. Once the adversary has reached a leaf, he will extend the path by going directly back to the root. If we assume that the tree has a total height of h, and that no page faults are caused on the way back to the root, then the adversary can cause a minimum of $\frac{h}{\log_d B} - \log_d M$ page faults in $2h$ steps. Since this process leaves the same state as was assumed initially, the process can continue indefinitely. Taking the ratio of steps to page faults gives the result. □

Corollary 6 *As the height of the tree grows without bound, the maximum worst-case speed-up is $\sigma(B) \leq 2 \log B / \log d$.*

Proof: This result follows by taking the limit of equation (7.2) as $h \to \infty$. □

For upper bounds on other kinds of trees, see Section 7.4, which gives upper bounds for general graphs.

Figure 7.4: A blocking of a d-ary tree using $s = 2$.

7.5.2 Lower bounds

So far, there is no known optimal algorithm that does not have storage blow-up greater than one. However, we can give a simple blocking with $s = 2$ that achieves optimal performance.

Lemma 53 *There is a blocking in a d-ary tree with $s = 2$ and an on-line paging algorithm such that the worst-case speed-up σ is at least $\frac{\log B}{2 \log d}$, with no assumptions on memory size other than the obvious $M \geq B$.*

Proof: Consider a blocking where we put complete trees of height $k = \log B / \log d + o(\log B)$ into each block. Each of the children of the leaves of a block will be the roots of new blocks. Assume that the root of the tree is the root of the top block. We will call this blocking $\mathcal{B}_1$. We will overlap a second such blocking $\mathcal{B}_2$ offset by a height of $k/2$, where $k = \log B / \log d + o(\log B)$, as shown in Figure 7.4. Clearly this can be done with a storage blow-up of 2, since vertices are represented at most once in each of $\mathcal{B}_1$ and $\mathcal{B}_2$. Now consider a step where the pathfront causes a page fault. There are two cases to consider: (1) the pathfront stepped off the top (i.e., towards the root) of a block that was in memory or (2) the pathfront stepped off the bottom (i.e., towards the leaves) of a block that was in memory. Without loss of generality assume that the block being stepped out of belongs to $\mathcal{B}_1$. In either case, we bring in the block in $\mathcal{B}_2$ containing the pathfront into memory (getting rid of anything else in memory to make room for the block). By construction, it will require at least $k/2$ additional steps to cause another page fault, proving the result. □

The lower bound when we allow a constant degree of storage blow-up is within a factor of 4 of the upper bound. This result contrasts with the general case, where potentially large (non-constant) storage blow-up is required to come within a constant

Figure 7.5: A diagonal grid graph in two dimensions.

factor of the optimal blocking speed-up. It is an open question what is possible in the absence of storage blow-up.

7.6 Searching in Grid Graphs and Diagonal Grid Graphs

By a grid graph in d dimensions, we mean a graph, where the vertex set consists of $\mathbf{Z}^d$ and the edge set is

$$E = \{((a_0, \ldots, na_d), (b_0, \ldots, b_d)) \mid \sum_i |a_i - b_i| = 1\}.$$

For example, in one dimension, the grid graph consists of the integer points on a number line. In this section, we will consider mainly infinite grid graphs. We will also consider diagonal grid graphs, which have the same vertex set as grid graphs, but have edge set

$$E = \{((a_0, \ldots, na_d), (b_0, \ldots, b_d)) \mid |a_i - b_i| \leq 1 \text{ for all } 1 \leq i \leq d\}.$$

Figure 7.5 shows an example of a diagonal grid graph in two dimensions.

7.6.1 1-dimensional grid graphs

We start with the results for 1-D grid graphs, where we have tight upper and lower bounds for the speed-up. The bounds, in particular the upper bound, may seem trivial,

but understanding the argument in one dimension is key to understanding the higher-dimensional analogues. (The upper bound is also better in some ways than any we have been able to prove for general graphs.) Note that in one dimension, a diagonal grid graph is the same as a grid graph.

Upper bounds

Lemma 54 *The fastest worst-case speed-up that could be achieved in a 1-D grid graph is* $\sigma \leq B$.

Proof: We adopt an adversarial position and demonstrate that under any blocking, we can force a page fault every B steps in the graph. We start anywhere (we arbitrarily call it vertex 0) and always walk to the right. We keep a potential function $\phi(t)$ which will increase by B every time we cause a page fault, and decrease by one for every step to the right we take. Let $p(t)$ be the number of page faults caused after we have taken t steps, and let $\phi(t)$ be the value of the potential function after t steps. We want to show that at all times,

$$p(t) \geq \frac{t}{B}, \tag{7.3}$$

or in other words that

$$\sigma = \frac{t}{p(t)} \leq B.$$

By definition,

$$\phi(t) = Bp(t) - t, \tag{7.4}$$

so rearranging equation (7.4), we get

$$p(t) = \frac{t}{B} + \frac{\phi(t)}{B}. \tag{7.5}$$

Comparing (7.3) and (7.5), we see that we can satisfy (7.3) by showing that for all t, $\phi(t) \geq 0$. But $\phi(t) - 1$ is an upper bound on how many covered cells are to the right of t, since it increases by $B - 1$ every time a page fault is caused and decreases by 1 every time a covered cell is reached. Since the number of covered cells to the right is non-negative and $\phi(t)$ is at least that large, it follows that $\phi(t) \geq 0$. □

We can also give a simple upper bound proof in the case where we have a finite 1-D grid graph.

Lemma 55 *The fastest worst-case blocking speed-up that could be achieved in a 1-D grid graph containing* ρM *vertices,* $\rho > 1$, *is*

$$\sigma \leq \frac{\rho}{\rho - 1}B - \frac{B}{(\rho - 1)M}.$$

Proof: Assume that we are at one end of the grid graph and that there are at most M vertices in memory. We will follow a path that will traverse to the other end of the grid graph. There are $(\rho - 1)M$ vertices not in memory, which means that we will cause at least $(\rho - 1)M/B$ page faults in $\rho M - 1$ steps. Taking the ratio of steps to page faults gives the result. □

Notice that when $\rho \to \infty$, the upper bound of Lemma 55 reduces to that of Lemma 54.

Lower bounds

It is possible to achieve the upper bound using the obvious strategy of putting B vertices in one block, the next B vertices in the next block, and so on.

Lemma 56 *There is a blocking of the 1-D grid graph with $s = 1$ that achieves a worst-case speed-up of $\sigma \geq B$, as long as $M \geq 2B$.*

Proof: Let $B_i = \{k \mid Bi \leq k < B(i+1)\}$ and consider the obvious blocking $\mathcal{B} = \{B_i \mid i \in \mathbf{Z}\}$. (See Figure 7.7a.) The blocks are non-overlapping sequences of B consecutive vertices. We show that in the steady state, we can service at least B steps in the graph every time a page fault is generated. Assume that we cause a page fault by stepping to vertex k. Without loss of generality, assume that we have stepped from vertex $k-1$; the same argument applies in the other direction. Then $k = Bi$ for some i and we can service the request by bringing in block B_i. Since $k-1$ was in memory, we know that block B_{i-1} had been read in at some previous time. We retain block B_{i-1}. Now, in order to cause another page fault, we must move from vertex k to either vertex $k-B-1$ or vertex $k+B$ since all the intervening nodes are in memory. Thus, it requires a minimum of B steps to cause another page fault. □

It is also possible to alleviate the memory requirement to $M \geq B$ if we allow the storage blow-up s to be 2 and let the speed-up decline to $B/2$ by overlapping two sets of the blockings used in the proof of Lemma 56 offsetting one by $B/2$ with respect to the other.

7.6.2 2-dimensional grid graphs

Let us turn now to 2-D grid graphs, where the results are not trivial.

Upper bound

Lemma 57 *In a 2-D grid graph, the fastest worst-case speed-up that could be achieved is $\sigma \leq 2\sqrt{B}$.*

Proof: This proof proceeds in a similar fashion to that of Lemma 54. This time, we consider an infinitely long, $\sqrt{B}$-wide corridor extending to the right. We divide t, the number of steps taken, into two components, t_x and t_y. We will let t_x be an index of how many columns we have traversed in the corridor (again, we will always be moving to the right), and t_y will be the total number of steps we have taken in the y direction. As before, we want to show that

$$p(t) \geq \frac{t}{2\sqrt{B}}. \tag{7.6}$$

We show that

$$t_x \leq \sqrt{B}p(t) \tag{7.7}$$
$$t_y \leq \sqrt{B}p(t) \tag{7.8}$$

so that $t = t_x + t_y \leq 2\sqrt{B}p(t)$, implying (7.6). Our scheme is actually quite simple: we show that there is an uncovered cell in the corridor within the next (amortized) $\sqrt{B}$ columns (including the current column) and then an additional $\sqrt{B} - 1$ moves in the y direction will suffice to step to any uncovered vertex in that column. As before, we define a potential function $\phi(t)$ that increases by B for every page fault we cause. This time, however, we decrement only when t_x increases, and we decrement by $\sqrt{B}$. Thus,

$$\phi(t) = Bp(t) - \sqrt{B}t_x. \tag{7.9}$$

$\phi(t)$ is an upper bound on the number of vertices covered in the channel to the right of (and including) the current column, since at most B vertices to the right can be newly covered when we cause a page fault, and by definition each step to the right leaves behind $\sqrt{B}$ covered cells. Rearranging (7.9), we get that

$$t_x = \sqrt{B}p(t) - \frac{\phi(t)}{B}. \tag{7.10}$$

Comparing (7.7) and (7.10), we see that we only need to show that $\phi(t) \geq 0$ to establish (7.7). But the interpretation of $\phi(t)$ ensures this, since the number of covered columns is non-negative.

We do not need to use an amortized argument to show (7.8), since a page fault is guaranteed to occur at least every $\sqrt{B} - 1$ steps in the y direction. □

Lower bounds

We can find a blocking that comes close to the above limit, if we allow a storage blow-up $s = 2$. First we need a definition.

Definition 10 By a *tessellation*, we mean a collection of regions such that the union of all the regions is the entire space and the intersection of any pair of regions never includes a point in the interior of either.

For simplicial tessellations, it is often required that any intersection between two simplices be a simplex of each; we make no such requirement in our definition.

Lemma 58 *There exists a blocking in a 2-D grid graph with $s = 2$ and an on-line paging algorithm that has a worst-case speed-up $\sigma \geq \sqrt{B}/4$, assuming that $M \geq 2B$.*

Proof: Consider a tessellation of the plane by blocks of size $\sqrt{B} \times \sqrt{B}$. Let the blocking be two such tessellations, offset by $\sqrt{B}/2$. Figure 7.6 demonstrates the blocking, where

Figure 7.6: A blocking of vertices in a 2-D grid with storage blow-up $s = 2$.

one tessellation is shown with solid lines and the other with dashed lines. Formally, we consider the blocking $\mathcal{B} = \{B_{ij} : i, j \in \mathbf{Z}\}$, where

$$B_{ij} = \left\{(k,\ell) : \left\lfloor \frac{i\sqrt{B}}{2} \right\rfloor \leq k < \left\lfloor \frac{i\sqrt{B}}{2} \right\rfloor + \sqrt{B}, \left\lfloor \frac{j\sqrt{B}}{2} \right\rfloor \leq \ell < \left\lfloor \frac{j\sqrt{B}}{2} \right\rfloor + \sqrt{B}\right\}.$$

Consider a step that causes a page fault. Without loss of generality, let the step be to the right out of the shaded block in Figure 7.6. By leaving the shaded block in memory and bringing in the appropriate boxed block, we can make sure that it will take at least $\sqrt{B}/4$ additional steps to produce another page fault. □

It is also possible to prove a lower bound on the speed-up even if there is no storage blow-up, but it is not quite as good. We will see in the next subsection that there is a good reason for this. Note also that the memory requirement is a little more stringent, requiring $M \geq 3B$ instead of $M \geq 2B$.

Lemma 59 *There exists a blocking in a 2-D grid graph with $s = 1$ and an on-line paging algorithm that has a worst-case speed-up $\sigma \geq \sqrt{B}/6$, assuming that $M \geq 3B$.*

Proof: Consider the blocking of Figure 7.7b, where each square has dimensions $\sqrt{B} \times \sqrt{B}$. The slowest speed-up occurs when the pathfront emerges at the place where three of the squares meet; in two steps it is possible to cause two page faults. However, the next page fault after those two will require at least $\sqrt{B}/2$ additional steps, so that we take $\sqrt{B}/2 + 2$ steps to cause 3 page faults, for a speed-up of $\sqrt{B}/6 + 2/3$. □

7.6.3 d-dimensional grid graphs and diagonal grid graphs

We can also show upper and lower bounds on higher-dimensional grid graphs and diagonal grid graphs. The 1-D and 2-D results are special cases of the result for d dimensions, but it was useful to start there to develop our intuitions. However, unexpected results to come from generalizing to multiple dimensions, like the fact

Figure 7.7: A blocking of vertices in grid graphs with storage blow-up $s = 1$, in (a) 1-D, (b) 2-D, and (c) 3-D. The sides of each block have length $B^{1/d}$.

that our upper and lower bounds are no longer within a constant factor if we block using isothetic (all sides parallel to the coordinate axes) hypercubes, and that going from $s = 1$ to $s = 2$ gives provably more than a constant factor improvement in σ in any blocking using isothetic hypercubes. Furthermore, it does not appear possible to create an optimal algorithm for higher-dimensional grid graphs with any reasonable storage blow-up (see Section 7.4); on the other hand, there is an optimal algorithm for higher-dimensional diagonal grid graphs with a storage blow-up of 2.

Upper bounds

We give a special-case argument for d-dimensional grid graphs that improves upon the general $8r^+(B)$ upper bound by a constant factor of $4/e \approx 1.47$.

Lemma 60 *In a d-dimensional grid graph, the fastest worst-case speedup that could be attained is $dB^{1/d}$.*

Proof: The argument for this proof is just like that of Lemma 57. This time, we have an infinite corridor to the positive first dimension with size

$$\underbrace{B^{1/d} \times \cdots \times B^{1/d}}_{d-1}.$$

We again divide t into d components, $t_1, \ldots, t_d$. We amortize in dimension 1, but not in the others, to show independently that

$$t_i \leq B^{1/d} p(t), \qquad 1 \leq i \leq d. \tag{7.11}$$

so that

$$t = \sum_{1 \leq i \leq d} t_i \leq dB^{1/d} p(t)$$

giving the desired result

$$p(t) \geq \frac{t}{dB^{1/d}}.$$

Again, we step to an uncovered location in the corridor such that t_1 is increased the minimum amount. This time we define

$$\phi = Bp(t) - B^{(d-1)/d} t_1, \tag{7.12}$$

which again is an upper bound on the number of vertices in memory that have their first coordinate at least t_1. Rearranging (7.12), we get

$$t_1 = B^{1/d} p(t) - \frac{\phi(t)}{B^{(d-1)/d}},$$

showing that the first coordinate of (7.11) is satisfied as long as $\phi(t) \geq 0$, which it is by its interpretation. The other dimensions all need to take fewer than $B^{1/d}$ steps per page fault to move to the empty space in the corridor. □

On the other hand, we can prove a tighter upper bound for diagonal grid graphs.

Lemma 61 *In a d-dimensional diagonal grid graph, the fastest worst-case speedup that could be attained is* $2B^{1/d}$.

Proof: The argument for this proof has points of commonality with both that of Lemma 57 and that of Lemma 60. We consider an infinite corridor to the positive first dimension with size

$$\underbrace{B^{1/d} \times \cdots \times B^{1/d}}_{d-1}.$$

This time, we divide t into 2 components, t_1 and t_2. We amortize in dimension 1, but not in dimension 2, to show independently that

$$t_i \leq B^{1/d} p(t), \qquad 1 \leq i \leq 2. \tag{7.13}$$

so that

$$t = t_1 + t_2 \leq 2B^{1/d} p(t),$$

giving the desired result

$$p(t) \geq \frac{t}{2B^{1/d}}.$$

Again, we step to an uncovered location in the corridor such that t_1 is increased the minimum amount. This time we define

$$\phi = Bp(t) - B^{(d-1)/d} t_1, \tag{7.14}$$

which again is an upper bound on the number of vertices in memory that have their first coordinate at least t_1. Rearranging (7.14), we get

$$t_1 = B^{1/d} p(t) - \frac{\phi(t)}{B^{(d-1)/d}},$$

showing that the first coordinate of (7.13) is satisfied as long as $\phi(t) \geq 0$, which it is by its interpretation. Unlike the situation with d-dimensional grid graphs, however, the diagonals ensure that all the other dimensions can reach the empty space in the corridor within $B^{1/d}$ steps per page fault, since all the other dimensions can be moving one step closer to the empty space simultaneously at each step. □

Lower bounds

In a similar fashion, we can get lower bounds on d-dimensional grid graphs and diagonal grid graphs.

Lemma 62 *There exists a blocking of a d-dimensional grid graph or a d-dimensional diagonal grid graph with* $s = 2$ *and an on-line paging algorithm that produces a worst-case speed-up of* $\sigma \geq \frac{1}{4} B^{1/d}$, *as long as* $M \geq 2B$.

Proof: This proof proceeds similarly to that of Lemma 58. We let our blocks be of size $B^{1/d}$ in each dimension and consider two overlapping grids where the corners of one grid correspond to the centers of the other grid. Exactly the same argument shows that when a page fault occurs, there is always some block that can be brought in so that the next page fault will take at least time $\frac{1}{4}B^{1/d}$ to occur. □

This algorithm is optimal within a factor of 8 for diagonal grid graphs. However, for grid graphs there is a discrepancy of $4d$ between the upper bound of Lemma 60 and this lower bound of Lemma 62. As we will see in the next lemma, it is the upper bound that is correct, and if we permit the storage blow-up to be as large as B, we can attain a blocking speed-up of at least $\frac{1}{2e}dB^{1/d}$, which is off from the upper bound by a factor of at most $2e$.

Lemma 63 *There is a blocking with $s = B$ that gives a worst-case speed-up of $\frac{1}{2e}dB^{1/d}$ in a d-dimensional grid graph.*

Proof: Lemma 49 gives a blocking with $s = B$ that achieves a blocking speed-up of $r^-(B)$. From Equation (7.1), $r^-(B) = \frac{1}{2e}dB^{1/d}$. □

Thus, the blocking given in the proof of Lemma 49 is better than the best known one with a constant storage blow-up ($s = 2$). The storage blow-up needed for this blocking is quite large, however, and it is not possible to describe the blocking simply. Using the storage reduction techniques of Theorems 28 and 30, we can reduce the storage blow-up to

$$s \leq \min\left\{\frac{6e}{d}B^{(d-1)/d}, 4^d\right\}$$

while giving up at most a factor of 2 in the blocking speed-up.

Blocking with Isothetic Hypercubes and $s = 1$

Intuitively, the problem with using isothetic hypercubes as blocks to tessellate the space, as in Lemma 62, is that the blocks do not correspond well with the neighborhoods of points that are within a certain distance r of some central point. In 2-D, for example, the neighborhoods are diamond-shaped. In 3-D, they are octohedra. In general, the shape of the neighborhoods will be the geometric dual of the isothetic hypercubes: take the centerpoint of each face of the d-dimensional hypercube and take the convex hull of the resulting points. The unfortunate property of using the geometric duals as neighborhoods is that when $d \geq 3$, the neighborhoods do not tessellate neatly.

We do not get even as good a result as Lemma 62 with isothetic hypercubes when we are not allowed to use storage blow-up greater than one. The following lemma gives a lower bound on the speed-up in grid graphs when no storage blow-up is permitted.

Lemma 64 *There is a blocking with $s = 1$ in a d-dimensional grid graph or a d-dimensional diagonal grid graph and an on-line paging algorithm that asymptotically achieves a worst-case speed-up of $\sigma \geq B^{1/d}/d^2$ as long as $M \geq (d+1)B$.*

Figure 7.8: A tessellation of d-space using isothetic unit hypercubes must consist of a series of shear hyperplanes.

Proof: This proof follows along the same lines as that of Lemma 59. Let p be the smallest prime such that $p \geq d$. (By the prime number theorem, with high probability $p \leq d + O(\log d)$.) We will block using regular isothetic hypercubes that have sides equal to $B^{1/d}$. We will tessellate d-dimensional space by stacking up tessellations of $(d-1)$-dimensional space extruded along the dth dimension. We will use the same $(d-1)$-dimensional tessellation in each layer, but we will offset adjacent layers by $1/p$ in the first dimension, $2/p$ in the second dimension, etc., to $(d-1)/p$ in the $(d-1)$st dimension, to arrange the tessellation so that not too many blocks meet at any given point. Figure 7.7 shows the tessellation for $1 \leq d \leq 3$. This stacking has been set up so that at no point do more than $d+1$ hypercubes meet; the construction ensures that a point where d hypercubes meet in $d-1$ dimensions is always in the middle of a face along the dth dimension. It similarly controls that places where $d-1$ hypercubes meet in $d-1$ dimensions never stack onto a place that 2 hypercubes meet, and so on. Thus, we can cause at most d page faults in consecutive moves, as long as we keep all the blocks meeting at a point in memory—this leads to the restriction that $M \geq (d+1)B$; to cause an additional page fault will always take at least $B^{1/d}/p$ additional steps. Thus the speed-up is at least

$$\frac{d + B^{1/d}/p}{d+1,}$$

which gives the result when p is approximated by $d + \log d$. □

Notice that this bound is off from the general grid graph upper bound by a factor of d^3. In fact, when storage blow-up is not permitted, we can prove a more stringent upper bound on any blocking that tessellates the d-dimensional space using isothetic hypercubes of size $B^{1/d}$ in each dimension; this algorithm is only off by a factor of d from the tighter upper bound. First we need a couple of topological lemmas regarding tessellations of d-space by unit hypercubes.

Lemma 65 *Any tessellation of d-space using isothetic unit hypercubes can be decomposed into layers of tessellations of $(d-1)$-space extruded along the dth dimension.*

Proof: Consider the two squares C_1 and C_2 in Figure 7.8 as projections into two dimensions of two hypercubes whose intersection is a $(d-1)$-dimensional region that is not a complete face of either hypercube. Clearly, the dotted cube C_3 is not an acceptable placement for an additional hypercube that touches C_2, since we know that the hypercubes tessellate the space, but no unit hypercube could fit in the niche between cubes C_1 and C_3. Thus, cube C_3 must abut both cubes C_1 and C_2. Continuing this argument shows that the solid hyperplane perpendicular to v must be a shear plane. Likewise, the hyperplanes on both sides of the $(d-1)$-dimensional tessellations must be shear planes, establishing the lemma. □

This lemma enables us to prove the next one, which we apply to get the more stringent upper bound for the blocking where we do not permit storage blow-up. We need a definition first.

Definition 11 A *complex of degree D* in a tessellation is a point that is incident upon D distinct regions of the tessellation.

Lemma 66 *Any tessellation of d-space using unit hypercubes contains a complex of degree at least* $d+1$.

Proof: The proof is by induction on the dimension d. In one dimension, there clearly must be points where two unit lines meet, forming a complex of degree 2. Assume that the lemma is true for $d-1$ dimensions. We know from Lemma 65 that the d-dimensional tessellation can be decomposed into layers of extruded $(d-1)$-dimensional tessellations, and by the inductive hypothesis, that there is some complex of degree d in each of these tessellations. When we arrange two adjacent layers together, the best we can do to limit the degree of complexes is to position a complex of degree d in one layer incident upon the $(d-1)$-dimensional face of the next layer. This process, however, produces a complex of degree $d+1$, maintaining the inductive hypothesis. □

Of course, in Lemmas 65 and 66, it does not matter that each hypercube has sides of unit length; the important fact is that all the hypercubes are identically the same size, so we can apply the lemma to the problem of finding a tessellation of a d-dimensional grid graph using hypercubes with sides that are all $B^{1/d}$. We set up our tessellation in such a way that the boundaries of the hypercubes always pass between integer points of the grid graph, so that we can maintain a storage blow-up of 1.

Lemma 67 *The fastest speed-up that could be achieved by decomposing a d-dimensional grid graph into blocks of size* $B^{1/d}$ *on a side is*

$$\sigma \leq \frac{B^{1/d}+d}{d+1}.$$

Proof: By Lemma 66, there must be a $(d+1)$-complex in the tessellation underlying our block decomposition. In a $(d+1)$-complex in a d-dimensional space, it is always possible for a path to visit all $d+1$ of the blocks that are "incident" in consecutive moves. (This fact can be proved by induction on the number of dimensions.) Thus,

if we extend the path to a point that is on a $(d+1)$-complex where we have not yet brought in any of the incident blocks, we can cause d page faults in d steps. It will take at most $B^{1/d}$ steps to move to another $(d+1)$-complex where only one block is in memory, to return to the starting situation. This fact means that we cause $d+1$ page faults every $B^{1/d}+d$ steps. □

This upper bound is a factor of $d/4$ worse than the one attained with isothetic hypercubes by using a storage blow-up $s=2$.

7.7 Conclusions

We have demonstrated matching upper and lower bounds for the blocking speed-up in searching of complete d-ary trees and d-dimensional grid graphs. We have also shown matching upper and lower bounds for classes of general graphs for which the maximum and minimum B-radii are within a constant factor of one another. Table 7.1 summarizes these results.

One of the issues we explored was the degree to which storage blow-up was necessary in attaining optimal blocking speed-ups. For complete trees and d-dimensional grid graphs, we were able to show the existence of optimal algorithms with a degree of storage blow-up that was independent of the block size (although there is still a dependence on the dimension for grid graphs). For general graphs, the degree of storage blow-up required to get optimal blocking speed-up by our algorithms is in general dependent upon the block size. In the case of d-dimensional grid graphs blocked using only isothetic hypercubes, were were able to show that even going from a storage blow-up of 1 to 2 results in provably worse performance for the worst-case blocking speed-up, reflected in Table 7.1 by the fact that the lower bound for $s=2$ is larger than the upper bound for $s=1$ as long as $d>4$. Thus, some degree of storage blow-up is necessary to get good blocking speed-ups. This observation implies that these techniques would not be very useful for applications that require any significant amount of updating of the data being stored.

There are clearly important questions that are still open.

1. Is it possible to prove that there are optimal lazy algorithms for the strong model?
2. What is the best possible speed-up attainable when $s=1$? Is there an algorithm to achieve the speed-up?
3. Are there upper bound techniques that take into account the storage blow-up?
4. Is it possible to reduce the storage blow-up to some constant and still achieve an optimal speed-up?
5. How do the results given here apply to directed graphs, instead of undirected graphs. This extension is particularly important for object-oriented databases.

Type of graph	Upper bound	Lower bound	s	$\frac{M}{B}$
Complete d-ary tree	$2\frac{\log B}{\log d}$	$\frac{\log B}{2\log d}$	2	≥ 1
1-D grid graphs	B	B	1	≥ 2
		$\frac{B}{2}$	2	≥ 1
2-D grid graphs	$2\sqrt{B}$	$\frac{1}{6}\sqrt{B}$	1	≥ 3
		$\frac{1}{4}\sqrt{B}$	2	≥ 2
d-D grid graphs	$dB^{1/d}$	$\frac{1}{2e}dB^{1/d}$	B	≥ 1
		$\frac{1}{4e}dB^{1/d}$	$\frac{6e}{d}B^{(d-1)/d}$	≥ 1
		$\frac{1}{4e}dB^{1/d}$	4^d	≥ 1
d-D grid graphs (isothetic hypercubes)	$dB^{1/d}$	$\frac{1}{4}B^{1/d}$	2	≥ 2
	$\frac{1}{d}B^{1/d}$	$\frac{1}{d^2}B^{1/d}$	1	$\geq d+1$
d-D diagonal grids	$2B^{1/d}$	$\frac{1}{4}B^{1/d}$	2	≥ 2
general graphs with ρM vertices	$r^+(M)$	$r^-(B)$	B	≥ 1
	$2r^-(M)$	$\frac{1}{2}r^-(B)$	$\frac{3B}{r^-(B)}$	≥ 1
	$2\frac{\rho}{\rho-1}B$	$\frac{1}{2}r^-(B)$	$\frac{B}{k-(\lfloor r^-(B)/4 \rfloor)}$	≥ 1
	$\left(2\frac{M}{B}+2\right)r^+(B)$			
	$8r^+(B)$			

Table 7.1: Summary of blocking results for external graph searching.

6. Is it possible to get matching upper and lower bounds for non-uniform classes of graphs? Intuitively, if there are a few vertices with a small B-radius, but they are separated by a large distance, say $r^+(M)$, then the adversary cannot take too much advantage of them if $B \ll M$.

Chapter 8

Conclusions

In this dissertation, we have considered the problem of minimizing the input/output bottleneck. We have looked at the problems of sorting in a wide variety of memory models, doing lattice computations in VLSI, and external graph searching. In all of these cases, we have found that specific attention must be paid to the issue of I/O in order to get optimal I/O bounds. Since the I/O time dominates all the problems considered, getting optimal I/O bounds is tantamount to achieving the optimal overall time bound.

We have found that allowing parallel I/O subsystems is an effective way of minimizing the total I/O time. In the case where we have parallel disks, it becomes necessary from the standpoint of scalability to allow the processors to be parallelized, since eventually a single processor would not be able to keep up with the demands of specifying what location to read from each disk.

The research contributions of this thesis are these:

1. We have given two different optimal algorithms for sorting on parallel disks, both of which are completely deterministic and simpler than the previously published randomized solution.

2. We have adapted the Balance Sort algorithm to sort optimally and deterministically in a wide range of parallel memory hierarchies, including a parallel disk model that also can exploit processor parallelism fully with up to $\min(M \log(M/B)/\log M, M \log\log M/\log M)$ processors.

3. We have given the first known optimal sorting algorithms for UMH hierarchies with bandwidth functions $b(\ell) = 1$ and $b(\ell) = \rho^{-c\ell}$. We have given a good, though not provably optimal algorithm for bandwidth function $b(\ell) = 1/(\ell + 1)$. All the algorithms are deterministic.

4. We have shown how to minimize the I/O bottleneck in VLSI computations of lattice computations, giving matching upper and lower bounds on the I/O efficiency.

5. We have given optimal algorithms for blocking and page replacement in external graph searching for B-uniform classes of graphs.

There are some areas for further research that this thesis opens up.

1. Is there a way of sorting in uniform memory hierarchies with bandwidth function $b(\ell) = 1/(\ell + 1)$ using time $O(N \log N)$? Alternatively, is there a way of proving a tight lower bound? Conceivably an $O(N \log N)$ algorithm could be developed using the Fusion Tree techniques to remove the extra $\log \log N$ factor [FrW].

2. Is it possible to prove that the simple balancing algorithm is optimal?

3. Are there deterministic algorithms for parallel memory hierarchies that can work with weaker interconnections between the processors than a CRCW-PRAM?

4. Can the techniques used for the lattice computations be generalized to higher-dimensional lattices or different processor connectivity patterns?

5. Can the techniques used for the lattice computations be generalized to show I/O bounds for neural network computations in VLSI?

6. Is there a way of blocking in grid graphs that gives optimal speed-up but does not require a storage blow-up that is exponential in the dimension?

7. Is it possible to find a less global graph property than $r^+(B)$ and $r^-(B)$ in external graph searching for showing upper and lower bounds?

8. Is it possible to take into account the storage for the edges in blocking optimally for external graph searching?

9. What are the repercussions in external graph searching of insisting that the storage blowup is unity?

Appendix A

Proof of Validity of Discrete Minimization Formula

In Section 6.2, we found that the I/O overhead for the multigeneration sweep architecture was

$$\psi(n, g) = \frac{2(n - g + 2)}{g(n - 2g + 2)}.$$

From (6.8), we have the value of g that minimizes ψ:

$$\tilde{g}_{\min}(n) = (n + 2)\xi,$$

where

$$\xi = 1 - \frac{\sqrt{2}}{2}.$$

Let $R(x)$ be the rounding function, defined by

$$R(x) = \lfloor x + 1/2 \rfloor. \tag{A.1}$$

We want to prove the assertion that taking the nearest integer function of $\tilde{g}_{\min}$ provides an integer value at which the minimum occurs, in other words

$$g_{\min}(n) = R(\tilde{g}_{\min}(n)) = R((n + 2)\xi). \tag{A.2}$$

This formula could produce the wrong result if the fractional part of $(n + 2)\xi$ falls in some "window of vulnerability" around $1/2$, so that the actual discrete minimum occurs at one integer value, but R rounds to the other integer value. The basic idea behind the proof of validity of (A.2) is to compute the window of vulnerability and then show that there is no smallest value of n for which the fractional part falls within that window.

Proving an exact closed form for the optimal number of generations to compute is not strictly necessary to show that the architecture meets the lower bound asymptotically to within a constant factor. However, from a practical standpoint, it is satisfying

Figure A.1: Top of the Stern-Brocot tree.

to have a closed-form formula that tells how many generations to compute for a particular value of n. Furthermore, the fact that there exists a closed-form equation is quite unexpected, as we hope to show in this subsection. The technique of using a Stern-Brocot tree to prove such an exact result is novel.

It is known that the set of fractional parts of all multiples of any irrational number β is dense on the unit interval $(0, 1)$, which is to say that given any $0 < \gamma < 1$ and $\epsilon > 0$,

A.1 The Stern-Brocot Tree

The proofs of the theorem depend heavily on the properties of the Stern-Brocot (S-B) tree [GKP]. This section explains how to derive the tree and gives the properties it has that are important to the proof.

The S-B tree starts with the fractions $0/1$ and $1/0$ and derives the next level by inserting between every pair of fractions m/n and m'/n' the fraction $(m+m')/(n+n')$. Figure A.1 shows the top part of the tree. The fractions that evaluate to 0 and ∞ are not really part of the tree; they are merely seeds to get the tree started. The S-B tree has several important properties:

1. All fractions that appear in the tree are in reduced form.

2. All reduced-form fractions appear in the tree.

3. An inorder traversal of the tree gives the fractions in ascending order.

4. We can treat the S-B tree as a binary search tree. If "L" means go down the left branch of the tree and "R" means go down the right branch, then all real numbers can be mapped to a unique element of the regular language $[LR]^\infty$. For rational numbers, that is, those that can be given a finite representation, the

unique infinite representation is the finite part followed by RL^∞. The element of $[LR]^\infty$ so generated is called the *S-B expansion* of a number.

5. If b is an irrational number, then the fractions generated in its S-B expansion are the simplest rational approximations to b in the sense that if m/n is an approximation to b, there exists an S-B expansion m'/n' such that $m' \leq m$, $n' \leq n$ and m'/n' is between m/n and b.

A.2 The Lemmas

These are the lemmas that lead up to the main theorem of this appendix. The first lemma establishes that $\psi(n, g)$ is asymmetric about its minimum.

Lemma 68 *Let $h(n, \delta)$ be the symmetric difference of $\psi(n, g)$ about its minimum for $\delta > 0$, that is,*

$$h(n, \delta) = \psi(n, \tilde{g}_{\min}(n) + \delta) - \psi(n, \tilde{g}_{\min}(n) - \delta).$$

Then $h(n, \delta) > 0$ for all integer $n > 0$ and $0 < \delta \leq 1/2$.

Proof: This is mostly a matter of algebra. We find that

$$h(n, \delta) = \frac{16\delta^3}{\left((n+2)^2(\sqrt{2}-1)^2 - 4\delta^2\right)\left((n+2)^2(\sqrt{2}-1)^2 - 2\delta^2\right)}.$$

We can set the partial derivative with respect to δ equal to 0 to find the minima and maxima. When we do this, we find a double root at zero, two imaginary roots, and roots at

$$\delta_\pm = \pm\frac{1}{4}\sqrt{\sqrt{33}-3}\left(2-\sqrt{2}\right)(n+2) \approx \pm 0.243(n+2),$$

of which we take the positive root, since we are only interested in $\delta > 0$. $\delta_+ > 1/2$ for all $n > 0$. It also represents a maximum in the curve, since

$$\begin{aligned} \frac{\partial^2 h(n, \delta_+)}{\partial \delta^2} &= -3\sqrt{\sqrt{33}-3}\left(67\sqrt{33}+385\right)\left(7\sqrt{2}+10\right)(n+2)^{-3} \\ &\approx -76142(n+2)^{-3}, \end{aligned}$$

which is negative for all $n > 0$. Since for all $n > 0$, $h(n, 0) = 0$, and the first local extremum in the curve is a maximum at $h(n, \delta)$ with $\delta > 1/2$, we conclude that $h(n, \delta) > 0$ for all $0 < \delta \leq 1/2$. □

Now we can show that the nearest integer can only produce an incorrect result if the fractional part of ξn is slightly greater than $1/2$ for some n.

Lemma 69 *Formula (A.2) for $g_{\min}$ can only produce an incorrect result if Frac$(\xi n) = 1/2 + \epsilon$, for some $\epsilon > 0$.*

Figure A.2: Behavior of ψ near its minimum.

Proof: Figure A.2 shows the behavior of ψ about its minimum for $m < \tilde{g}_{\min} < m+1$. If $\psi(n, g)$ were exactly symmetric about its minimum, then $R(\tilde{g}_{\min}(n))$ would always produce the discrete minimum of $\psi(n, g)$, where R is as defined in (A.1). From Lemma 68, we know that $\psi(n, \tilde{g}_{\min}+\delta) > \psi(n, \tilde{g}_{\min}-\delta)$, so that if the minimum falls at a fractional part that is $1/2 + \epsilon$, $\psi(n, m)$ may be less than $\psi(n, m+1)$. No similar problem exists if $\epsilon < 0$. □

Thus, there are "windows of vulnerability" if the fractional part of ξn is slightly larger than $1/2$. The next lemma shows how large these windows of vulnerability are.

Lemma 70 *If m is any integer, then (A.2) will produce the wrong result if*

$$m + \frac{1}{2} < \tilde{g}_{\min}(n) < m + \frac{1}{2} + \epsilon(m),$$

for some n, where

$$\epsilon(m) = \frac{1}{2}\left(\sqrt{2} - 1\right)\left(\sqrt{4m^2 + 4m + 2} - (2m + 1)\right). \tag{A.3}$$

Proof: A window of vulnerability exists when $\tilde{g}_{\min}(n)$ falls between $m + 1/2$ and the point such that $\psi(n, m) = \psi(n, m + 1)$ for some integer m and n. Given a value of m, we determine n from the equation

$$\tilde{g}_{\min}(n) = (n + 2)\xi = m + \frac{1}{2} + \epsilon,$$

or equivalently

$$n = \left(2 + \sqrt{2}\right)\left(m + \frac{1}{2} + \epsilon\right). \tag{A.4}$$

The condition $\psi(n,m) = \psi(n,m+1)$ leads to

$$n^2 + 2n(1-2m) + 2m(m-3) = 0.$$

Plugging in the value of n from (A.4) gives

$$4\left(3+2\sqrt{2}\right)\epsilon^2 + 4\left(1+\sqrt{2}\right)(2m+1)\epsilon - 1 = 0.$$

The values of ϵ solving this equation are

$$\epsilon(m) = \frac{1}{2}\left(\sqrt{2}-1\right)\left(\pm\sqrt{4m^2+4m+2} - (2m+1)\right).$$

The negative root is negative for all $m > 0$, so the positive root must be the correct one. □

It is instructive to do the asymptotics on $\epsilon(m)$ to see how large the windows of vulnerability are. A simple derivation reveals that

$$\epsilon(m) = \frac{1}{8}\left(\sqrt{2}-1\right)m^{-1} - \frac{1}{16}\left(\sqrt{2}-1\right)m^{-2} + O(m^{-3}). \tag{A.5}$$

The windows of vulnerability thus start out small and shrink asymptotically to zero.

Now, we adopt a different tack to find out which multiples of ξ could possibly have fractional parts that fall within the windows of vulnerability. The next lemma gives the S-B expansion of ξ.

Lemma 71 *The S-B expansion for ξ is $L(LLRR)^\infty$.*

Proof: The expansion begins with L, since $\xi < 1$. Let those fractions that would underestimate ξ according to the expansion above (those with an "R" in their next digit) be denoted by m_k/n_k and those that would overestimate ξ be denoted by p_k/q_k. We can derive expressions for m_k, n_k, p_k, and q_k by using recurrence relations. See Figure A.3. The table at the left in the figure indicates which subscript of which variables to which each fraction in the tree corresponds. For example, $p_2/q_2 = 1/3$. For example, the recurrence for m_k and n_k when k is odd is given by the simultaneous recurrence

$$\begin{aligned}
m_1 &= 1; \\
m_k &= 3m_{k-2} + 4p_{k+1}; \\
n_1 &= 4; \\
n_k &= 3n_{k-2} + 4q_{k+1}; \\
p_2 &= 1; \\
p_k &= 2m_{k-3} + 3p_{k-2}; \\
q_2 &= 3; \\
q_k &= 2n_{k-3} + 3q_{k-2}.
\end{aligned}$$

Figure A.3: Stern-Brocot expansion of $1 - \sqrt{2}/2$.

From this recurrence, we can get generating functions for m_k and n_k when k is odd:

$$\begin{aligned} m_{\text{odd}}(z) &= \sum_{k \text{ odd}} m_k z^k = \frac{z^3 + z}{z^4 - 6z^2 + 1}, \\ n_{\text{odd}}(z) &= \sum_{k \text{ odd}} n_k z^k = \frac{4z}{z^4 - 6z^2 + 1}, \end{aligned}$$

which yield for $k \geq 1$

$$\begin{aligned} m_{2k-1} &= \frac{1}{2}\left(\left(1+\sqrt{2}\right)^{2k-1} + \left(1-\sqrt{2}\right)^{2k-1}\right); \\ n_{2k-1} &= \frac{1}{2}\left(\left(4+3\sqrt{2}\right)\left(1+\sqrt{2}\right)^{2k-2} + \left(4-3\sqrt{2}\right)\left(1-\sqrt{2}\right)^{2k-2}\right). \end{aligned}$$

Similarly, we get for $k \geq 1$

$$\begin{aligned} m_{2k} &= \frac{1}{4}\left(\left(4+3\sqrt{2}\right)\left(1+\sqrt{2}\right)^{2k-2} + \left(4-3\sqrt{2}\right)\left(1-\sqrt{2}\right)^{2k-2}\right); \\ n_{2k} &= \frac{1}{2}\left(\left(7+5\sqrt{2}\right)\left(1+\sqrt{2}\right)^{2k-2} + \left(7-5\sqrt{2}\right)\left(1-\sqrt{2}\right)^{2k-2}\right). \end{aligned}$$

Let us define

$$\Delta_k = \xi n_k - m_k, \tag{A.6}$$

which is the fractional part of ξn_k. It is straightforward to show that

$$\begin{aligned} \Delta_{2k-1} &= \left(3-2\sqrt{2}\right)^k, \qquad \text{for } k \geq 1; \\ \Delta_{2k} &= \xi\left(3-2\sqrt{2}\right)^k, \qquad \text{for } k \geq 1. \end{aligned}$$

Since these quantities are always positive, m_k/n_k always underestimates ξ. Similarly, it can be shown that p_k/q_k always overestimates ξ. Thus, these fractions must constitute the S-B expansion of ξ. □

The next lemma tells about the convergence properties of m_k/n_k.

Lemma 72 *The fractions m_k/n_k converge monotonically to ξ.*

Proof: Monotonic convergence is not an automatic property of S-B expansions of numbers. Let Δ_k be as in (A.6). Let δ_k be defined by $\delta_k = \xi - m_k/n_k$. Then $\delta_k = \Delta_k/n_k$. We get

$$\frac{\delta_{2k-1}}{\delta_{2k}} = \frac{\left(7+5\sqrt{2}\right)\left(1+\sqrt{2}\right)^{2k-1} + \left(7-5\sqrt{2}\right)\left(1-\sqrt{2}\right)^{2k-1}}{\left(1+\sqrt{2}\right)^{2k} + \left(7-5\sqrt{2}\right)\left(1-\sqrt{2}\right)^{2k-1}}.$$

This ratio converges rapidly to $3+2\sqrt{2} \approx 5.282$ and is always greater than 5.28. Similarly,

$$\frac{\delta_{2k}}{\delta_{2k+1}} = \frac{\left(1+\sqrt{2}\right)^{2k+2} + \left(7-5\sqrt{2}\right)\left(1-\sqrt{2}\right)^{2k+1}}{\left(1+\sqrt{2}\right)^{2k} + \left(41-29\sqrt{2}\right)\left(1-\sqrt{2}\right)^{2k-1}}.$$

This ratio also converges rapidly to $3+2\sqrt{2}$ and is always greater than 5.28. Since these ratios are both greater than one, these fractions converge monotonically. □

We want the set of numbers b_k for which the fractional part $b_k\xi$ approaches $1/2$ from the top more closely than $n\xi$ for any other $n < b_k$. It is easy to see from the generating functions that when k is odd, m_k is odd and n_k is even. If we define $\Delta_k = \xi n_k - m_k$, then

$$\xi\frac{n_{2k-1}}{2} - \frac{m_{2k-2}-1}{2} = \frac{1}{2} + \frac{\Delta_{2k-1}}{2},$$

where the two fractions on the left-hand side are integers and Δ_{2k-1} gets exponentially small. Thus, the series of numbers for $k \geq 1$

$$b_k = \frac{n_{2k-1}}{2} = \frac{1}{4}\left(\left(4+3\sqrt{2}\right)\left(3+2\sqrt{2}\right)^{k-1} + \left(4-3\sqrt{2}\right)\left(3-2\sqrt{2}\right)^{k-1}\right), \qquad (A.7)$$

would seem to be a good candidate for the numbers we are seeking. The next lemma proves that this is actually the case.

Lemma 73 *Let us define*

$$\chi(n) = Frac(\xi n - 1/2), \qquad (A.8)$$

so that $\chi(n)$ is the amount by which the fractional part of ξn exceeds $1/2$ (or a value greater than $1/2$ if the fractional part does not exceed $1/2$). The values of b_k defined in (A.7) have the property that

$$\chi(b_k) = \min_{1 \leq n \leq b_k} \{\chi(n)\}.$$

Proof: Assume that there exists an $n < b_k$ such that $\chi(n) < \chi(b_k)$. By definition, there exists an m such that

$$\xi n - m = \frac{1}{2} + \chi(n).$$

Doubling this equation leads to

$$\xi(2n) - (2m+1) = 2\chi(n).$$

Thus, the fraction $(2m+1)/2n$ is a rational approximation that underestimates ξ. By property 5 of the S-B tree, there must be an underestimator to ξ in its S-B expansion, m_j/n_j, with $n_j \leq 2n$, that is at least as good as $(2m+1)/2n$. Also, $2n < 2b_k = n_{2k-1}$, so that $j < 2k-1$. But now we have two fractions in the S-B expansion of ξ, of which the one with the smaller index approximates ξ better, contradicting Lemma 72. Thus, the assumption that there was such a value of n was incorrect, and the lemma is proved. □

The only remaining fact needed to prove the theorem is that $\chi(b_k) \geq \epsilon(m(b_k))$ for all k, where $m(n)$ is defined by $m(n) = \lfloor \xi n \rfloor$.

Lemma 74 *We have* $\chi(b_k) = \epsilon(m(b_k))$.

Proof: We know that

$$\chi(b_k) = \frac{\Delta_{2k-1}}{2} = \frac{1}{2}\left(3 - 2\sqrt{2}\right)^k.$$

We also know from Lemma 73 that

$$m(b_k) = \frac{m_{2k-1} - 1}{2}.$$

Plugging into (A.3), we get

$$\begin{aligned}\epsilon(m(b_k)) &= \frac{1}{4}\left(\sqrt{2}-1\right) \\ &\quad \left(\sqrt{\left(1+\sqrt{2}\right)^{4k-2} + 2 + \left(1-\sqrt{2}\right)^{4k-2}} - \left(1+\sqrt{2}\right)^{2k-1} - \left(1-\sqrt{2}\right)^{2k-1}\right).\end{aligned}$$

The item in the square root is a perfect square, so

$$\begin{aligned}\epsilon(m(b_k)) &= \frac{1}{4}\left(\sqrt{2}-1\right) \\ &\quad \left(\left(1+\sqrt{2}\right)^{2k-1} - \left(1-\sqrt{2}\right)^{2k-1} - \left(1+\sqrt{2}\right)^{2k-1} - \left(1-\sqrt{2}\right)^{2k-1}\right) \\ &= \frac{1}{2}\left(3-2\sqrt{2}\right)^k.\end{aligned}$$

The expressions for $\chi(b_k)$ and $\epsilon(m(b_k))$ are thus equal. □

A.3 The Theorem

We are now ready to prove the theorem.

Theorem 32 *The formula for $g_{\min}$ above gives an integer value that minimizes $\psi(g, n)$.*

Proof: The values of $\tilde{g}_{\min}$ comprise all the multiples of an irrational number ξ. The nearest integer function R will give the wrong answer only if the fractional part of that multiple of ξ is sufficiently close to $1/2$ that R will round it one way, but the minimum would occur by rounding it the other. Lemma 69 shows that, for positive n, this event will only happen if the fractional part is slightly larger than $1/2$. If we let $m = \lfloor \xi n \rfloor$ then Lemma 70 computes the window of vulnerability $\epsilon(m)$, that is, the maximum amount by which the fractional part of a failing value of ξn can exceed $1/2$. This window of vulnerability is a monotonically decreasing function of m. Let N be the set of all n for which $R(\tilde{g}_{\min}(n)) \neq g_{\min}(n)$. Then if $N \neq \emptyset$, it has a smallest element n'. Define $\chi(n)$ as in (A.8). For any $n'' < n'$, $\chi(n') < \chi(n'')$ since otherwise $\xi n''$ would fall within its larger window of vulnerability, contradicting the assumption that n' is the smallest failing multiple. So we need to consider only those values of n such that $\chi(n)$ is smaller than for any preceding value. Lemma 73 proved the explicit form of those numbers b_k. Finally, Lemma 74 demonstrated that each b_k falls exactly at the edge of a window of vulnerability, which means that

$$\psi(b_k - 2, g_{\min}(b_k - 2)) = \psi(b_k - 2, g_{\min}(b_k - 2) - 1),$$

so that rounding to either side produces a minimum. This concludes the proof of the theorem. □

A.4 The Significance of this Result

It is known that the set of fractional parts of all multiples of any irrational number β is dense on the unit interval $(0, 1)$, which is to say that given any $0 < \gamma < 1$ and $\epsilon > 0$, there exists an integer n such that the fractional part of βn is within ϵ of γ. It turns out that the fractional parts are uniformly distributed along the unit interval. Let us consider the following probabilistic argument about whether the discrete minimum equation is true for all values of n: We assume that in any interval m to $m + 1$, there is a probability of $\epsilon(m)$ that a multiple of ξ falls within the ϵ-window. If we define

$$\lambda = \frac{1}{16}\left(\sqrt{2} - 1\right),$$

the probability that all of the ϵ-windows is missed is, from (A.5),

$$\Pr\{\text{everywhere correct}\} \leq \prod_{m \geq 1} \left(1 - 2\lambda m^{-1} + \lambda m^{-2}\right).$$

Taking logs, we get

$$\begin{aligned}\log(\Pr\{\text{everywhere correct}\}) &\leq \sum_{m\geq 1} \log\left(1 - 2\lambda m^{-1} + \lambda m^{-2}\right) \\ &\leq \sum_{m\geq 1} \left(-2\lambda m^{-1} + \lambda m^{-2}\right) = -\infty.\end{aligned}$$

Therefore, we find that $\Pr\{\text{everywhere correct}\} = 0$.

So it seems the fact the discrete minimization formula is everywhere correct is a bit of a surprise: even though the fractional parts of the multiples of ξ are uniformly distributed on the unit interval, the probabilistic argument is not valid because the fractional parts of the multiples of ξ do not approach $1/2$ from the top until the ϵ-window has shrunk just enough to be missed.

Appendix B

Evidence for the Optimality of the Simple Balancing Algorithm

In Chapter 3, we mentioned that there is evidence that the simple balancing algorithm, Algorithm 2, is within a constant factor of optimal in balancing the buckets among the D disks so that the buckets can be read efficiently in parallel. On the theory that many things are proved by standing on the shoulders of others, this Appendix is to let the interested researcher know what progress has already been made towards proving the optimality of this algorithm.

Even the optimal algorithm may require as many as $\lceil N/D \rceil + S - 1$ parallel reads, since none of the buckets may have a size that is a multiple of D. We want to compare our greedy strategy to the optimal balancing strategy, which can read all the records at once and write them out in an order that minimizes the number of parallel reads as summed over all the buckets. We would like to prove that the greedy algorithm above always comes within a factor of 2 of the optimal strategy, as long as the number of buckets $S \leq \sqrt{N/2D}$. To do this, we will try to show that no arrangement of records on the disk can cause the imbalance to increase quickly. Since we are interested in worst-case behavior, we will often adopt an adversarial argument where we try to increase the imbalance as quickly as possible by placing the records on the disk.

First we need some definitions. Let $y_b^{\max} = \max_d \{y_{bd}\}$ and $y^{\max} = \max_b \{y_b^{\max}\}$. We consider that x, y, and $y^{\max}$ are all functions of t, the track number, but we usually omit the explicit dependence for clarity of notation.

B.1 Strategies for Proving Optimality

We would like to be able to show that all the buckets can be read in on the next pass in $O(N)$ time, or in other words, that the distribution strategy is within a constant factor of optimal. There are two strategies that we have devised that could be used to show that the distribution strategy always comes within a constant factor of the optimal: the time strategy and the bucket-limited strategy. These strategies are based upon the following conjectures.

Conjecture 1 $y^{\max}(t) \leq ct^{\alpha}$ *where* $\alpha \leq 1/2$.

Conjecture 2 $y^{\max}(t) \leq cS$.

Conjecture 1 can be equivalently stated that it requires at least a quadratic number of tracks before any given value of $y^{\max}$ can be attained in the skew matrix. We actually suspect that the number of tracks needed is $((y^{\max})^3 + 5y^{\max})/6$ (more on this later). For Conjecture 2, we suspect the constant c is $1/2$.

Let the track number t range between 1 and T, so that $T = \lceil N/D \rceil$. Then the optimal number of parallel I/O's required for reading all S buckets is at least T (it could be as high as $T + S - 1$ if none of the buckets has a number of records in it that is a multiple of D).

In computing the number of parallel reads needed by the algorithm, we divide the quantity into two parts: that required to read the balanced part of the buckets and that required for the unbalanced part. The balanced part of the buckets can require at most T parallel reads. Since there are only S buckets, if each bucket had an imbalance of $y^{\max}$, the total number of parallel reads

$$R \leq T + Sy^{\max}.$$

If we could prove either of the conjectures, we could bound the contribution of the unbalanced parts of the buckets.

Conjecture 1 would give the overall bound if we bound $S \leq c'T^{1/2}$, so that the total number of parallel reads would be

$$R \leq T + Sy^{\max} = T + ScT^{1/2} \leq \left(1 + \frac{cc'}{\sqrt{2}}\right) T,$$

since $S \leq \sqrt{T/2}$. This number is within a factor of $1 + cc'/\sqrt{2}$ of optimal. Conjecture 2 gives $R \leq T + cS^2$ which gives a constant factor above optimal as long as $S = O(T^{1/2})$.

B.2 Preliminaries

In this subsection, we give the background definitions and some simple lemmas along with their proofs.

Definition 12 Let $\mathcal{B} = \{1, \ldots, S\}$ be the set of buckets. Then a *track* τ is a D-length vector $[\tau_1, \ldots, \tau_D]$ such that $\tau_i \in \mathcal{B}$ for $1 \leq i \leq D$.

Note that in this definition of a track, we identify a record with the bucket to which it belongs, since that is all that matters with respect to the algorithm.

Definition 13 Let y be a skew matrix. Then $y^{\max} = \max_{b,d} \{y_{bd}\}$ is called the *attainment* of y. $y_b^{\max} = \max_d \{y_{bd}\}$ is called the *imbalance* of bucket b. Let $B \subset \mathcal{B}$. Then $y_B^{\max} = \max_{b \in B, d} \{y_{bd}\}$ is the *imbalance of set* B.

Definition 14 We write $y \stackrel{\tau}{\mapsto} z$ to mean that there is some min-cost assignment of buckets in track τ that converts the skew matrix y to skew matrix z. We say that y *yields* z *under* τ. In general, z is not unique.

Definition 15 Given $y \stackrel{\tau}{\mapsto} z$. If $y_b^{\max} < z_b^{\max}$ then we say that bucket b is *promoted* under τ.

Definition 16 Given skew matrix y_{bd} and track τ. Let $B(\tau)$ be the set of buckets represented in τ, i.e., $B(\tau) = \bigcup_i \{\tau_i\}$. Then we say that τ causes a *local promotion* if there exists a skew matrix z_{bd} such that $y_{bd} \stackrel{\tau}{\mapsto} z_{bd}$ and $z_{B(\tau)}^{\max} > y_{B(\tau)}^{\max}$. We say that τ causes a *global promotion* if $z^{\max} > y^{\max}$.

We only require that there be some min-cost matching that increases $y_{B(\tau)}^{\max}$ or $y^{\max}$ since the algorithm does not specify any particular min-cost matching and we are interested in worst-case behavior. Our goal is to find a limit on how quickly $y^{\max}$ can promote, since it is a measure of how unbalanced the worst bucket is.

Definition 17 A track τ is called *birepresentative* if $|B(\tau)| = 2$.

Definition 18 A configuration y *admits of a (locally) promoting track* if there exists some track τ such that for some z for which $y \stackrel{\tau}{\mapsto} z$, $z^{\max} > y^{\max}$ ($z_{B(\tau)}^{\max} > y_{B(\tau)}^{\max}$).

We will sometimes refer to a skew matrix as a *configuration*. Here are a few simple lemmas. In what follows, we assume that y is the skew matrix corresponding to histogram matrix x. We remind the reader of the following lemmas, proved in Chapter 3

Lemma 75 *Every bucket b contains at least one zero in the skew matrix.*

Lemma 76 *The disk sums, $y_d = \sum_b y_{bd}$, are all equal.*

Lemma 77 *Any track promoting disk d results in at most one bucket b being promoted on disk d.*

There are some other simple lemmas we can show.

Lemma 78 *If $y \stackrel{\tau}{\mapsto} z$, then $|y_{bd} - z_{bd}| \leq 1$ for all b, d.*

Proof: All the elements of the histogram matrix either stay the same or increase by one. The lemma then follows from the definition of the skew matrix. □

Definition 19 We say that a bucket b *fills all its zeroes* on a track if an element of b is assigned to each disk d for which $y_{bd} = 0$.

Lemma 79 *A local promotion can occur on track τ comprising bucket set $B(\tau)$ only if there exists a bucket b such that an element of b is assigned to a disk d for which $y_{bd} = y_{B(\tau)}^{\max}$ and b does not fill all its zeroes.*

Proof: In order for a local promotion to occur, an element of some bucket b must be assigned to a disk d such that $y_{bd} = y_{B(\tau)}^{\max}$. But if bucket b fills all its zeroes, then $\min_d \{x_{b,d}\}$ increases by 1 also, so y_{bd} remains constant and no promotion occurs on bucket b. If every bucket b containing a $y_{B(\tau)}^{\max}$ fills all its zeroes, then a local promotion does not occur. □

B.3 The Two-Disk Pebble Game.

We start by considering the special case where $D = 2$. If a bucket has values (a_1, a_2) in the skew matrix on disks 1 and 2 respectively, then we can define a bijection $f : \mathbf{Z}^+ \times \mathbf{Z}^+ \rightarrow \mathbf{Z}$ as follows:

$$f(a_1, a_2) \longrightarrow \begin{cases} a_1 & \text{if } a_1 > 0 \\ -a_2 & \text{otherwise} \end{cases}$$

The function f is a bijection because every bucket has a zero either on disk 1 or on disk 2 as a result of Lemma 75, so that at most one of a_1 or a_2 is non-zero.

We can now consider how the min-cost algorithm assigns elements to disks when $D = 2$. There are only three cases to consider:

1. Both elements read represent the same bucket. In this case, the skew matrix remains unchanged.

2. Two buckets with the same sign are represented. Then the unbalance of the bucket with the smaller unbalance is increased and the unbalance of the other is decreased.

3. Two buckets with opposite signs are represented. Then the unbalance of both buckets is decreased.

These cases can be succinctly enumerated by considering each bucket to be a pebble on a number line. There is exactly one move that can take place:

$$(i, j) \longrightarrow (i + 1, j - 1) \quad \text{if } i \leq j, \tag{B.1}$$

where the fact that this mapping uses pairs indicates that pairs of pebbles are moved. (The fact that that we must move pebbles in pairs reflects the fact that the skew matrix remains unchanged if both elements represent the same bucket.)

We can now express the adversarial problem of trying to get a large imbalance as an equivalent problem of trying to put a pebble on some positive (or negative) integer k starting with an infinite supply of pebbles on node 0, using only applications of Rule B.1. We would like it to be difficult to get a pebble onto k, and in particular, we would like to show at least a quadratic lower bound.

We begin, however, by showing an upper bound on this problem by proposing a simple algorithm to get a pebble on node k.

Lemma 80 *There exists an algorithm to place a pebble on any node $k > 0$ within $(k^3 + 5k)/6$ moves.*

Proof: The algorithm is to find the largest-numbered node having two or more pebbles on it (this will initially be node 0, with its infinite supply), and apply Rule B.1 to two pebbles at that node. Table B.1 gives the number or pebbles on each node for the first few moves of the algorithm. It can be shown by induction that this algorithm

Move	Node			
	1	2	3	4
0	0			
1	1			
2	2			
3	0	1		
4	1	1		
5	2	1		
6	0	2		
7	1	0	1	
8	2	0	1	
9	0	1	1	
10	1	1	1	
11	2	1	1	
12	0	2	1	
13	1	0	2	
14	1	1	0	1

Table B.1: The first moves of the Two-Disk Pebble Game algorithm.

eventually places a pebble on every node $k > 0$. The induction proceeds by noticing that when a pebble is first placed on node $k-1$, there is one pebble each on nodes $0, \ldots, k-3, k-1$, while node $k-2$ has no pebbles on it. An additional $k-2$ moves results in all the nodes except $k-3$ having a single pebble on them, and so on, so that in $\sum_{1 \le i \le k-2} i$ additional moves, every node up to $k-1$ has a single pebble on it. An additional k moves places a pebble on node k, leaving the exact configuration we assumed in our inductive hypothesis. We therefore get a recurrence

$$\begin{aligned} T(k) &= T(k-1) + \sum_{1 \le i \le k-1} i + 1 \\ T(0) &= 0 \end{aligned}$$

for which the solution is $T(k) = (k^3 + 5k)/6$. □

It is hard to imagine how any algorithm could do better than this, but the problem of proving a cubic lower bound has so far been unsolved. Notice that the given algorithm had the property that all of the moves were of the form (i, j), where $i = j$, even though dissimilar moves (those for which $i < j$) are allowed by Rule B.1. There is a reason for this.

Theorem 33 *No minimal schedule for placing a pebble on node k contains any dissimilar moves.*

Proof: We show this theorem by taking any schedule $\mathcal{S}$ containing dissimilar moves and producing a shorter schedule $\mathcal{S}'$ that has at least the same attainment. Thus, if we

Other pebble	Ⓛ unlabeled	Ⓛ labeled	Ⓛ′ (= Ⓛ − 1) unlabeled	Ⓛ′ (= Ⓛ − 1) labeled
$k < Ⓛ - 1$	$k + 1$	$Ⓛ - 1$	$k + 1$	$Ⓛ - 2$
$k = Ⓛ - 1$	$Ⓛ$	$Ⓛ - 1$	$Ⓛ$	$Ⓛ - 2$
$k = Ⓛ$	$Ⓛ - 1$	$Ⓛ + 1$	*	*
$k > Ⓛ$	$k - 1$	$Ⓛ + 1$	$k - 1$	$Ⓛ$

* Simply swapping the labels and cutting out the move maintains the invariant.

Table B.2: Where to move pebble Ⓛ′ when pebble Ⓛ is moved.

Other pebble	Ⓤ unlabeled	Ⓤ labeled	Ⓤ′ (= Ⓤ + 1) unlabeled	Ⓤ′ (= Ⓤ + 1) labeled
$k < Ⓤ$	$k + 1$	$Ⓤ - 1$	$k + 1$	$Ⓤ$
$k = Ⓤ$	$Ⓤ + 1$	$Ⓤ - 1$	*	*
$k = Ⓤ + 1$	$Ⓤ$	$Ⓤ + 1$	$Ⓤ$	$Ⓤ + 2$
$k > Ⓤ + 1$	$k - 1$	$Ⓤ + 1$	$k - 1$	$Ⓤ + 2$

* Simply swapping the labels and cutting out the move maintains the invariant.

Table B.3: Where to move pebble Ⓤ′ when pebble Ⓤ is moved.

have a schedule attaining k that has a dissimilar move in it, we can produce a shorter schedule that attains at least k, so the original schedule could not have been minimal, since we will have

$$\begin{aligned} \text{length}(\mathcal{S}') &< \text{length}(\mathcal{S}) \\ \text{attainment}(\mathcal{S}') &\geq \text{attainment}(\mathcal{S}) \end{aligned}$$

Let $\mathcal{S}'$ be the same as $\mathcal{S}$ until the first move (i, j) in the schedule for which $i < j$. $\mathcal{S}'$ will leave out move (i, j) and will never add any moves not present in $\mathcal{S}$, guaranteeing that $\text{length}(\mathcal{S}') < \text{length}(\mathcal{S})$. In schedule $\mathcal{S}$, mark the pebble moved from node i with Ⓛ (for lower) and the pebble moved from node j with Ⓤ (for upper). In schedule $\mathcal{S}'$, mark one of the pebbles on i with Ⓛ′ and one of the pebbles on j with Ⓤ′. We start out with the invariants $Ⓛ' = Ⓛ - 1$ and $Ⓤ' = Ⓤ + 1$. (Here we are using a label to stand for the node on which the labeled pebble is located.) We continue to maintain these invariants until such point (if any) as Ⓛ and Ⓤ cross each other in $\mathcal{S}$ by using the moves on the right hand side of Table B.2 to show where to move pebble Ⓛ′ when pebble Ⓛ as moved and likewise the moves of Table B.3 when pebble Ⓤ is moved. Any move not involving Ⓛ or Ⓤ is identical in $\mathcal{S}'$. The only stipulations on how labels move in schedule $\mathcal{S}$ is that on move $(Ⓛ, Ⓛ)$, the label moves up and on move $(Ⓤ, Ⓤ)$ the label moves down. There are now three situations.

(1) Ⓛ and Ⓤ never cross. In this case, the subset of pebbles that $\mathcal{S}'$ has on nodes Ⓤ

and above is always the same as for $\mathcal{S}$, except that $\textcircled{U'} = \textcircled{U} + 1$. Since the attainment comes from this subset, attainment($\mathcal{S}'$) $\geq$ attainment($\mathcal{S}$).

(2) $\textcircled{L}$ and $\textcircled{U}$ cross in such a way that $\textcircled{L} = \textcircled{U} + 1$. In that case, $\mathcal{S}'$ is in exactly the same configuration as $\mathcal{S}$ at that point, since $\textcircled{L'} = \textcircled{L} - 1$ and $\textcircled{U'} = \textcircled{U} + 1$:

$$\begin{array}{rcc} \mathcal{S}: & \textcircled{U} & \textcircled{L} \\ \mathcal{S}': & \textcircled{L'} & \textcircled{U'} \end{array}$$

so $\mathcal{S}'$ can just finish the rest of schedule $\mathcal{S}$ (ignoring labels).

(3) $\textcircled{L}$ and $\textcircled{U}$ cross in such a way that $\textcircled{L} = \textcircled{U} + 2$. Here is the situation:

$$\begin{array}{rlccc} \mathcal{S} & \text{before:} & & \textcircled{L} & \\ & & & \textcircled{U} & \\ & \text{after:} & \textcircled{U} & & \textcircled{L} \\ \mathcal{S}' & \text{before:} & \textcircled{L'} & & \textcircled{U'} \end{array}$$

So $\mathcal{S}'$ can leave out the crossing move and wind up in the same configuration as $\mathcal{S}$, as in the previous case. Thus, for cases (2) and (3), attainment($\mathcal{S}'$) = attainment($\mathcal{S}$). □

We are now ready to prove our quadratic lower bound for the two-disk case.

Theorem 34 *When $D = 2$, it takes $\Omega(k^2)$ tracks to attain k.*

Proof: We argue this using the pebble game formulation. By Theorem 33, we need not consider any schedules with dissimilar moves, since we are interested in lower bounds. For each value $0 \leq m < k$, consider the last time a pebble was promoted from node m to $m + 1$. In order to do that promotion, at least two pebbles must have been present at m, one of which never goes past m again by definition of the fact that this is the last promotion. Hence, that pebble is never used in any of the promotions from $m + 1$ onward, so new pebbles will be needed. But it required at least m moves to get the never-again-used pebble up to node m, so the total work is at least $\sum_{0 \leq m < k} m = \Omega(k^2)$. □

This theorem is equivalent to proving Conjecture 1 for the case where $D = 2$, so that we know Algorithm Balance_Buckets will be within a constant factor of optimal.

B.4 What we know when $D > 2$.

The situation gets a lot more complicated when $D > 2$. It is not easy to represent the algorithm as a pebble game for the reasons that (1) more than two pebbles would be involved at every move, (2) a pebble can be promoted on a disk without any other pebble being demoted on that disk, and (3) a pebble can be demoted on a disk without promoting any other pebble. Fortunately, (2) cannot happen indefinitely, since every time it occurs, there is a finite resource (the number of zeroes) that is consumed.

First, however, we can give an upper bound to the minimum number of tracks needed to attain k.

Lemma 81 *A value of k can be attained in $(k^3+5k)/6$ moves for any $D>1$.*

Proof: We follow the same strategy as for Lemma 80. Recall that since we are trying to show worst-case behavior, we need only show that there exists some min-cost matching that causes the sought-after attainment. Without loss of generality, we seek to attain k on disk 1 by having one bucket for each bucket in the two-disk argument. A bucket of the form $(i,0)$ in the two-disk case will correspond to a bucket $(i,0,\ldots,0)$ in the multi-disk case; a bucket of the form $(0,1)$ will correspond to $(0,1,\ldots,1)$. The correspondence is already present in the initial configuration, so we only need to show that the correspondence can be maintained. All of the moves in the proof for Lemma 80 are of the form

$$\left.\begin{array}{cc} \underline{i} & 0 \\ i & \underline{0} \end{array} \longrightarrow \begin{array}{cc} i+1 & 0 \\ i-1 & 0 \end{array}\right\} \text{for } i \geq 1$$

or

$$\begin{array}{cc} \underline{0} & 0 \\ 0 & \underline{0} \end{array} \longrightarrow \begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array}$$

We can clearly replace those with

$$\left.\begin{array}{cccc} \underline{i} & 0 & \ldots & 0 \\ i & \underline{0} & \ldots & \underline{0} \end{array} \longrightarrow \begin{array}{cccc} i+1 & 0 & 0 & 0 \\ i-1 & 0 & 0 & 0 \end{array}\right\} \text{for } i \geq 1$$

and

$$\begin{array}{cccc} \underline{0} & 0 & \ldots & 0 \\ 0 & \underline{0} & \ldots & \underline{0} \end{array} \longrightarrow \begin{array}{cccc} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 1 \end{array}$$

by giving the promoted bucket one representative and the demoted one $D-1$ representatives. The shown matchings are clearly min-cost. Hence, we can attain k in $(k^3+5k)/6$ moves as in the two-disk case. □

The next theorem gives necessary and sufficient conditions for a birepresentative promoting track to exist.

Theorem 35 *A configuration y admits of a b,b'-birepresentative locally promoting track τ iff (1) there exists a disk d such that $y_{bd} = y_{b'd} = y^{\max}_{\{b,b'\}}$ and (2) there exists a disk d' such that $y_{bd'} = y_{b'd'} = 0$.*

Proof: Our parlance for conditions (1) and (2) will be to say that buckets b and b' *share a* $y^{\max}$ and *share a zero*, respectively.

($\Longrightarrow$) Without loss of generality, consider any matching in which bucket b is locally promoted. We show that any matching that promotes bucket b is not min-cost (or even locally min-cost) unless bucket b' shares a $y^{\max}$ and a zero. Let $B=\{b,b'\}$.

In order for b to be locally promoted, an element of b must be assigned to some disk d for which $y_{bd} = y^{\max}_B$ by Lemma 79. Also by Lemma 79, bucket b has a zero on some disk d' that is not filled. Look at the partial skew matrix:

$$\begin{array}{ccc} & d & d' \\ b & \underline{y^{\max}_B} & 0 \\ b' & a_1 & \underline{a_2} \end{array}$$

where underlines indicate assignments. Now a necessary condition for the matching to be min-cost is that $y_B^{\max} + a_2 \leq a_1$, or equivalently, $a_1 - a_2 \geq y_B^{\max}$. By definition, $0 \leq a_1, a_2 \leq y_B^{\max}$, so the only way to satisfy this condition is if $a_1 = y_B^{\max}$ and $a_2 = 0$. But then buckets b and b' share both a max and a zero, as claimed.

($\Longleftarrow$) To prove this direction, we must compute a track that allows a min-cost local promotion; in other words, we must determine how many elements to assign to buckets b and b' to ensure that there is a min-cost promoting matching. For every disk d for which $y_{bd} \neq y_{b'd}$, put an element of the bucket with the lower value into the track. Of the remaining elements (of which there are at least two), we assign one to bucket b and the rest to bucket b'. Clearly, any matching in which every disk d for which $y_{bd} \neq y'_b d$ is matched to the bucket with the lower cost will be a min-cost matching. At least one such matching will promote bucket b by matching its $y_B^{\max}$ without filling any shared zeroes. □

We can also show a necessary condition for a promoting track containing any number of representatives.

Lemma 82 *A (not necessarily birepresentative) track τ is a locally promoting track only if there exist buckets b and b' that share both a $y_{B(\tau)}^{\max}$ and a zero.*

Proof: This proof follows along the same lines as the forward proof of Theorem 35. Consider a matching that promotes bucket b and let b' be the bucket that assigns an element to some disk where bucket b has a zero (there must be one by Lemma 79). The same argument as before shows that the matching cannot be min-cost unless $y_{b'd} = y^{\max}$ and $y_{b'd'} = 0$. □

Since having shared zeroes is so important to local promotions (including global promotions), it is useful to know how zeroes can be created.

Lemma 83 *The number of zeroes a bucket b has in the skew matrix does not increase unless b fills all its zeroes.*

Proof: Look at the row for bucket b in the assignment matrix x. A zero occurs in the skew matrix whenever $x_{bd} = \min_{d'}\{x_{bd'}\} \equiv x_b^{\min}$. Since the elements in the assignment matrix are monotonically increasing, the only way that a zero could be introduced on some disk d for which $x_{bd} > x_b^{\min}$ is for $x_b^{\min}$ to increase, which entails filling all the existing zeroes. □

Bucket-limited theorems

It is an open question whether the attainment is even finite for fixed S and general D, let alone linear as postulated in Conjecture 2. There are some special cases of S for which we can prove a finite attainment, however.

Lemma 84 *The maximum attainment possible when $S = 1$ is 0.*

Proof: The same bucket has to be written to all disks, increasing both the min and max for that bucket. □

Lemma 85 *The maximum attainment possible when $S = 2$ is 1.*

Proof: Let $\mathcal{C}_0$ denote the starting configuration and let $\mathcal{C}_1$ denote the configuration

$$\mathcal{C}_1 : \begin{array}{c} \\ b_1 \\ b_2 \end{array} \overbrace{\begin{array}{ccc} 1 & \ldots & 1 \\ 0 & \ldots & 0 \end{array}}^{r_1} \overbrace{\begin{array}{ccc} 0 & \ldots & 0 \\ 1 & \ldots & 1 \end{array}}^{r_2}$$

where $r_1 + r_2 = D$. Any monorepresentative track leaves the skew matrix unchanged. The first birepresentative track will give configuration $\mathcal{C}_1$ up to a permutation of the disks. There are no buckets b and b' that share a zero, so no promoting track is possible by Lemma 82. Any birepresentative tracks from configuration $\mathcal{C}_1$ lead either to $\mathcal{C}_1$ again (since at least one bucket will fill all its zeroes and the other will fill only zeroes) or $\mathcal{C}_0$ (if bucket b_1 has exactly r_2 representatives). Hence, we can represent the possibilities as a finite state machine. □

It is possible to extend this proof technique to $S = 3$, resulting in a finite state machine with four states and thirteen arcs. The exhaustive evaluation of all possibilities is tedious, however, and what becomes clearest is that this proof technique does not scale well.

Intuitively, the bucket b_1 with the highest attainment had some other bucket b_2 that it needed to get it to that attainment (by Lemma 82), and bucket b_2 needed some different bucket b_3, so that there is a linear limit to the attainment when only S buckets are provided. This assumes that there can not be some finite set of buckets that "ratchet" one another up infinitely. If that could be proved, then we would have our result.

Birepresentative theorems

One approach to proving the bound would be to show that there is a track-minimal attainment for any k that uses only birepresentative tracks, and then to give a lower bound for how fast birepresentative tracks can attain a given value. There are several lemmas that pertain to this approach.

Lemma 86 *Any configuration that admits of a locally promoting track admits of a birepresentative locally promoting track.*

Proof: Lemma 82 states that any configuration that admits of a locally promoting track τ has buckets b and b' that share both a $y^{\max}_{B(\tau)}$ and a zero. But then, by the backward direction of Theorem 35, there exists a birepresentative track τ' comprising buckets b and b' that will cause the local promotion. □

Unfortunately, this lemma leaves open whether or not we can combine such birepresentative local promotions to produce a track-minimal attainment.

Lemma 87 *One half of all shared zeroes are filled on a b, b'-birepresentative track.*

Proof: Let d be any disk for which buckets b and b' share a zero. Something must be assigned to disk d, so that means either bucket b or b' fills its zero. (Note that the zero could "reappear" if the bucket fills all its zeroes.) □

Lemma 88 *Either bucket b or b' fills all its unshared zeroes on a b, b'-birepresentative track.*

Proof: Assume that neither fills all its unshared zeroes. Then we have some situation

$$\begin{array}{rcc} b: & 0 & \underline{a_1} \\ b': & \underline{a_2} & 0 \end{array}$$

By definition of the matching being min-cost, $a_1 + a_2 \leq 0$, so that $a_1 = a_2 = 0$. But then they are not unshared zeroes. □

One of the cases that might concern us if a bucket with a lower attainment can keep promoting a bucket with a higher attainment, without being demoted itself. The following lemma shows that this process cannot happen.

Lemma 89 *Assume that $y_{b'd} < y_{bd}$. Then bucket b' can introduce zeroes to b in a birepresentative track without demoting y_{bd} only if (1) $y_{b'd} = y_{bd} - 1$ and (2) for all disks d' for which a zero is introduced, $y_{b'd'} = 0$.*

Proof: Here is the situation in the partial skew matrix that adds a zero to bucket b on disk d' without demoting bucket b on disk d:

$$\begin{array}{rccc} & d & d' & d'' \\ b: & \underline{y_{bd}} & 1 & \underline{0} \\ b': & y_{b'd} & \underline{y_{b'd'}} & y_{b'd''} \end{array}$$

We know that $y_{bd} + y_{b'd'} \leq y_{b'd} + 1$. Since $y_{b'd} < y_{bd}$, this is only possible if $y_{b'd} = y_{bd} - 1$ and $y_{b'd'} = 0$. □

General theorems

It might be possible to gain a handle on $y^{\max}$ by considering the disk sum in the skew matrix $c(y)$ since clearly $y^{\max} \leq c(y) \leq Sy^{\max}$.

Lemma 90 *Let $c(y)$ be the (unique) disk sum of skew matrix y, and let h be the number of buckets all of whose zeroes are filled when $y \stackrel{\tau}{\mapsto} z$. Then $c(z) = c(y) + 1 - h$.*

Proof: Look at any disk d. Let x be the histogram matrix corresponding to skew matrix y. x_{bd} will increase by one for exactly one bucket b. In the absence of any buckets filling all their zeroes, this will increase $c(y)$ by one. Consider now any bucket b' that fills all its zeroes. There are two cases: (1) if $y_{b'd} > 0$ then filling the zeroes will cause $z_{b'd} = y_{b'd} - 1$ if $b \neq b'$, and counteracts the placing of the elements on d if $b = b'$. (2) If $y_{b'd} = 0$, then by definition of the fact that b fills all its zeroes, $b = b'$, so the increase is counteracted. In every case, $c(z)$ is decreased below $c(y) + 1$ by one for every bucket that fills all its zeroes. Thus, $c(z) = c(y) + 1 - h$. □

It is possible to prove that at least one zero is filled by every track (although the zero may reappear). This approach might be useful in the bucket-limited strategy (trying to show Conjecture 2), since the zeroes comprise the exhaustible resource.

Theorem 36 *If every row in a non-negative matching matrix z contains a zero, then a min-cost matching will match at least one row to a column where the cost is zero.*

Proof: There are actually two ways to prove this theorem, a simple one and a complicated one. We give both proofs because the complicated proof provides a foundation for understanding the proof of Theorem 37.

(Simple proof.) A min-cost matching in a skew matrix is isomorphic to doing min-cost matching on a complete bipartite graph where every node on the left has at least one edge of zero cost. Given any matching in such a graph that contains no zero-cost edges, we can find a matching with a smaller cost. Start at any node on the right and trace back over its matched edge. Then follow some zero edge back to the right. Continue this process until you reach some node on the right a second time. At this point, we will have a loop that is a path alternating between zero and non-zero cost edges. The non-zero cost edges are all in the matching, but we could replace those with all the zero-cost edges to create another perfect matching with a smaller cost.

(Complicated proof.) The complicated proof is the same argument as the simpler one, but in a different guise. Assume that we are given a $D \times D$ matrix and a min-cost matching. Permute the rows and columns so that the matching appears along the main diagonal (there are many ways of doing this). If any diagonal element is zero, then we have filled a zero and we are done. Assume therefore that the diagonal has no zeroes. We now show that if $D-1$ of the rows contain a zero, the last row cannot have one, contradicting our assumption that all rows contain a zero. We do this by showing that there is a permutation of the rows and columns leaving the matching on the main diagonal such that $z[i,j] > 0$ for all $1 \leq j \leq i$. In particular, this will mean that row D can contain only non-zero elements.

We construct our permutation by induction on the row number n in such a way that (1) $z[i,j] > 0$ for all $1 \leq j \leq i \leq n$ and (2) $z[i,i+1] = 0$ for all $1 \leq i < n$. For the base case of the induction, we show that the conditions hold for $n = 2$, since condition (2) does not make sense when $n = 1$. We have assumed that $z[1,1] > 0$ and that row 1 has a zero. Without loss of generality, let $z[1,2] = 0$ by swapping rows and columns if needed while keeping the matching on the main diagonal. We have assumed that $z[2,2] > 0$, so to show our initial conditions, we only need to show that $z[2,1] > 0$. By definition of a min-cost matching, however, $z[2,1] \geq z[1,1] + z[2,2] > 0$.

Now assume that conditions (1) and (2) hold up through $n-1$. We have the situation shown in Figure B.1. Here, the bullets refer to elements that are greater than zero. Row $n-1$ has a zero somewhere, so without loss of generality let it be in column n. We can now show that row n has only non-zero elements in it up through column n. We assumed that $z[n,n] > 0$ since it is part of the matching, being on the main diagonal. Consider any element $z[n,i]$ for which $i < n$ and look at the submatrix indicated by the dashed lines. In order for the partial matching in the submatrix to be

Figure B.1: Why a mincost matching must match a zero.

min-cost, we must have

$$z[n,i] + \sum_{i \le d \le n-1} z[d,d+1] = z[n,i] + 0 \ge \sum_{i \le d \le n} z[d,d] \ge n - i + 1 > 0,$$

or the boxed elements in the submatrix will have a smaller sum than the underlined ones. Embedded within this inequality is our induction step, $z[n,i] > 0$.

Continuing the induction, we find that row D cannot have a zero, contradicting the assumption that every row must have a zero. Therefore, at least one diagonal element must be zero. □

We would like to be able to use Theorem 36 in a bucket-limited framework to say that once we get down to S zeroes, we can not increase our total imbalance any more because the next track will have to fill all the zeroes of at least one bucket. Unfortunately, however, filling all the zeroes could introduce more zeroes into that bucket, allowing further promotions.

Definition 20 We write $\mathrm{Prom}(b, d, \tau, y)$ to mean that there is a min-cost matching that promotes bucket b on disk d by track τ from configuration y.

Definition 21 Assume that we have a track τ and that $b \in B(\tau)$. Then we call $B_S(b, d, \tau) = \{b' \mid y_{b'd} \ge y_{bd}, b' \in B(\tau)\}$ the *superior set* of b.

We can show that if a track forces a promotion of a bucket on some disk, then a zero is filled for some bucket in its superior set with respect to that disk and track. First we prove a simpler assertion.

Lemma 91 *Assume Prom(b, d, τ, y). Then for every disk d' for which $y_{bd'} = 0$, d' is matched to some $b' \in B_S(b, d, \tau)$. The cost of this part of the match is at most $y_{b'd} - y_{bd}$.*

Proof: We have this situation:

$$\begin{array}{rcc} & d & d' \\ b: & \underline{y_{bd}} & 0 \\ b': & y_{b'd} & \underline{y_{b'd'}} \end{array}$$

Since $y_{bd} + y_{b'd'} \leq y_{b'd}$, we know that $y_{b'd} \geq y_{bd}$, meaning that $b' \in B_S(b, d, \tau)$. Also, $y_{b'd'} \leq y_{b'd} - y_{bd}$. □

We generalize this lemma still further in the next theorem, which uses the same argument as the complicated proof for Theorem 36.

Theorem 37 *Assume Prom(b, d, τ, y). Then there exists a bucket $b' \in B_S(b, d, \tau)$ that fills a zero.*

Proof: We use a diagonalization argument. Assume without loss of generality that $b = d = 1$, and assume that no $b' \in B_S(b, d, \tau)$ fills a zero. We will permute our rows and columns so that our matching winds up on the main diagonal. Bucket b has an unfilled zero (by definition of being promoted); permute it to disk 2 and permute the bucket that matches on new disk 2 to bucket 2. Then we have

$$\begin{array}{cc} \underline{y_{11}} & 0 \\ y_{21} & \underline{y_{22}} \end{array}$$

We know that $y_{21} \geq y_{11} + y_{22}$ so $b_2 \in B_S(b_1, d, \tau)$. By assumption then, $y_{22} > 0$.

Inductively, we have a state where

$$\begin{array}{cccccc} \underline{y_{11}} & 0 & & & & \\ y_{21} & \underline{y_{22}} & 0 & & & \\ y_{31} & y_{32} & \underline{y_{33}} & 0 & & \\ \vdots & & & \ddots & & \\ y_{i-1,1} & & & & \underline{y_{i-1,i-1}} & 0 \\ y_{i1} & & & & & \underline{y_{ii}} \end{array}$$

where $y_{kj} > 0$ for all $1 \leq j \leq k \leq i$ and $y_{k1} > y_{11}$ for all $2 \leq k \leq i$ (in other words, $b_k \in B_S(b_1, d, \tau)$ for all $1 \leq k \leq i$). We show that we can extend this configuration to $i + 1$. Swap disks so that bucket i's first zero is on disk $i + 1$ and swap buckets so that the matched bucket on disk $i+1$ is bucket $i+1$. It is easy to see that $b_{i+1} \in B_S(b_1, d, \tau)$, so by assumption, $y_{i+1,i+1} > 0$. By the same argument as before,

$$y_{i+1,k} \geq \sum_{k \leq j \leq i+1} y_{jj} > 0$$

for all $1 \leq k \leq i + 1$. Since each $y_{jj} > 0$, $y_{i+1,k} > 0$.

Continuing the induction, eventually we will run out of buckets (when we have a $D \times D$ matrix of the above form), and bucket D will have no zeroes — a contradiction. Hence at least one of the buckets had to fill a zero. Since all the buckets are in $B_S(b, d, \tau)$, the theorem is proved. □

It is not necessarily true that a zero in the superior set is filled for every unfilled zero in the promoted bucket, as the following counterexample shows:

$$\begin{array}{ccc} \underline{1} & 0 & 0 \\ 10 & \underline{1} & 0 \\ 10 & 10 & \underline{0} \end{array}$$

B.5 Empirical results.

I have written two programs for simulating the results of Algorithm Balance_Buckets: the first program does a complete search of every possible combination of tracks from the initial state, to try to elicit worst-case behavior of the algorithm in a bucket-limited framework; the second program generates a random permutation and distributes the buckets to disks according to the algorithm to see what the average-case behavior is.

B.5.1 Worst-case behavior

The first program accepts as inputs two parameters: the number of disks D, and the number of buckets S. It starts by creating a $D \times S$ excess matrix initialized to all zeroes. Then, it tries all possible tracks from each configuration to produce a complete tree of all the configurations comprising the search space.

The biggest difficulty in the program was trying to determine when two configurations were equivalent. We desire to consider configurations equivalent if we can obtain one from the other by some permutation of its rows and columns. Ideally, we could convert each configuration to some canonical form and then do an element-by-element comparison to check for equality. Unfortunately, this approach is apparently not possible. To see this, consider the special case where the configurations have only values drawn from the set $\{0, 1\}$. In this case, the problem is equivalent to the graph isomorphism problem, viewing the matrix as an adjacency graph (although not all possible graphs are represented, since each bucket must have at least one zero in it by Lemma 5 — I doubt if this restriction matters), a problem that is known to have no polynomial-time solution as long as D$\neq$ND. Traces of the program showed that the majority of the running time was involved in this routine, so I made a special effort to speed it up as much as possible. I used a hash table for saving the configurations to see if a given configuration had already been encountered. To do this, I needed a hash function that guaranteed that any two configurations that might be identical were hashed to the same address. Simply sorting all the elements in the configuration and then hashing resulted in too many different configurations having with the same hash value. Eventually, I sorted the values of the buckets to produce a series of bucket hash keys (called a signature) and then sorted and combined the bucket hash keys to produce the global hash key. Even so, for $D = 3, S = 6$, the 4000 or so configurations were distributed among only about 1000 slots (of 8192) in the hash table. However, these optimizations produced a speedup of a factor of 50 or more in some cases.

As an example of the hashing difficulties, consider the histogram of the hash table entries when $D = 3, S = 6$:

#configurations	0	1	2	3	4	5	6	7	8	9	10	11
#hash values	7197	325	154	128	101	49	50	44	29	8	23	11

#configurations	12	13	14	15	16	17	18	19	20	23	29	36
#hash values	16	21	6	6	1	5	8	2	2	2	3	1

For example, there were three hash values, each of which had 29 configurations in them. There are three possible sources of collisions in the final hash table: (1) different sets of bucket values being hashed to the same bucket hash key, (2) different configurations having the same signature, and (3) different signatures having the same global hash value. I sincerely doubt that source 1 contributed to the overall crowding of the final hash table. To separate out the effects of sources 2 and 3, I checked to see how many signatures were in each hash table slot:

#signatures	0	1	2	3
#hash values	7197	957	36	2

So there were only two hash values to which three different configuration signatures were mapped and the vast majority of occupied hash values comprised only a single configuration signature. Thus, it was the inability of the signatures to distinguish between different configurations that was the limiting factor in the overall performance of the algorithm.

A second problem with the simulation is that the search space was not quite complete by virtue of the fact that only a single min-cost matching was considered for each (configuration,track) pair. Conceivably, some of the alternative min-cost matchings could have resulted in some configurations not obtained by the program, though I am reasonbly certain this is not the case.

Now, let us look at the results of the simulations. Table B.4 gives the maximum attainment, the minimum number of tracks to attain the maximum, the maximum disk sum, the total number of configurations obtained, and the CPU time in seconds for all the values of S and D for which the simulation completed.

The maximum attainment for a given number of buckets seems to be limited by $A(S) = \lfloor \frac{S}{2} \rfloor$, independent of the number of disks, and the maximum disk sum is always at least $\frac{1}{2}A(S)(A(S)+1)$. The number of configurations and consequently the running time appears to be growing at least exponentially in both S and D.

A curiosity in Table B.4 is that the minimum number of tracks needed to attain a given value may decrease as the number of buckets increases, as can be seen when $D = 2$ in going from 8 to 9 buckets and from 10 to 11 buckets. Table B.5 gives more complete information on how many tracks were needed to attain a given value with a given number of buckets. In all the cases we were able to run, the minimum number of tracks needed to get a certain attainment is dependent only upon the number of buckets and not at all on the number of disks. Empirically, increasing the number of disks does not change anything in terms of the minimum speed at which an attainment is reached, and in particular, it is plausible that the number of tracks needed for a track-minimal attainment given an infinite number of buckets may be independent of the number of disks.

		Max. attainment D			Min. tracks D			Max. disk sum D		
		2	3	4	2	3	4	2	3	4
	3	1	1	1	1	1	1	1	2	4
	4	2	2	2	3	3	3	3	4	5
	5	2	2	2	3	3	3	3	4	5
S	6	3	3	≥ 3	8	8	≥ 8	6	8	
	7	3	≥ 3		7	≥ 7		6		
	8	4			17			10		
	9	4			16			10		
	10	5			33			15		
	11	5			33			15		

		# configurations D			CPU time D		
		2	3	4	2	3	4
	3	2	4	5	0.0	0.0	0.1
	4	6	36	207	0.1	1.4	23.3
	5	7	84	1132	0.1	7.5	417.8
S	6	29	4060		1.2	1392.7	> 5321.5
	7	37			3.1	> 5748.1	
	8	193			44.5		
	9	269			254.9		
	10	1629			5157.7		
	11	2400					

Table B.4: Simulation results in the bucket-limited experiments.

Attainment		1			2			3			4	5
		D			D			D			D	D
		2	3	4	2	3	4	2	3	4	2	2
	3	1	1	1	—	—	—	—	—	—	—	—
	4	1	1	1	3	3	3	—	—	—	—	—
	5	1	1	1	3	3	3	—	—	—	—	—
S	6	1	1	1	3	3	3	8	8	8	—	—
	7	1	1		3	3		7	7		—	—
	8	1			3			7			17	—
	9	1			3			7			16	—
	10	1			3			7			15	33
	11	1			3			7			14	30

Table B.5: The minimum number of tracks needed to attain a given value.

$S = 8$		$S = 9$		$S = 10$		$S = 11$	
4	0	4	0	4	0	4	0
2	0	2	0	2	0	2	0
1	0	1	0	1	0	1	0
0	2	0	2	0	2	0	1
0	2	0	2	0	1	0	1
0	2	0	1	0	1	0	1
0	1	0	1	0	1	0	1
0	0	0	1	0	1	0	1
		0	0	0	1	0	1
				0	0	0	1
						0	0

Table B.6: Configurations when a value of 4 is attained with $D = 2$.

However, the number of tracks needed in Table B.5 to obtain the maximum attainment does sometimes depend upon the number of buckets available, though there is always some critical number of buckets after which the number of tracks needed does not decrease any more as the number of buckets is increased. If we are interested in exploring the track-limited framework, it would seem that we need to make certain that there are enough buckets available to achieve the track-minimal attainment. How do we know if we have the critical number of buckets? It is possible to tell by inspecting the configuration when the maximum attainment is achieved in the $D = 2$ case. For example, Table B.6 shows the configurations when a value of 4 is attained for different values of S.

It is obvious that what is happening is that when the number of buckets is less than the critical value, additional moves must be taken to replenish the supply of pebbles on the zero node of the number line, to phrase things in terms of the $D = 2$ pebble

game. Thus, we wind up requiring extra moves to get pebbles to nodes that are less than -1. Accordingly, we know that $S = 11$ represents the critical number of buckets for a track-minimal attainment of 4, since no bucket has more than an imbalance of 1 on Disk 2. Even given the configuration where 4 was attained with $S = 8$, it is possible to extrapolate that three more buckets would be needed for the critical value.

The critical number of buckets as far as we can tell from this Table B.5 (together with the configuration when 5 is attained) are:

Attainment	1	2	3	4	5
Minimum # buckets	2	4	7	11	16

where it appears from extrapolating that minimum $= y^{\max}(y^{\max} + 1)/2 + 1$.

It is also possible to extrapolate the value of the minimum number of tracks needed for an attainment given a configuration that attains the value using a sub-critical number of buckets. When $S = 8, D = 2$, for example, the value 4 is attained in 17 tracks, 3 of which were needed for causing extra promotions on Disk 2; hence 14 is the minimum number of tracks required even with an infinite number of buckets. It is possible to rewrite the program slightly to expand the range for which we can infer the minimum number of tracks required for a given attainment. In this variation of the program, each bucket is initialized to have a 1 on Disk 1 and a 0 on all other disks. One track is charged for each bucket. This is the equivalent of saying that we started with $2S$ buckets and combined them in pairs to promote one on Disk 1 and the other on the remainder of the disks. Since when $D = 2$, we know that a track-minimal attainment will never involve any buckets that are non-zero on Disk 2, we can ignore these extra S buckets. Our program is initialized so that the initial track count is S, since that is how many tracks it would take to promote S buckets to have a 1 on Disk 1. When we do this, we can construct the following table of the minimum number of tracks to attain various values:

Attainment	1	2	3	4	5	6	7
Minimimum # tracks	1	3	7	14	25	41	63

In each case, the minimum number of tracks is $((y^{\max})^3 + 5y^{\max})/6$, indicating that it is probably the quadratic lower bound proof that needs improving rather than the cubic upper bound.

All of the theoretical work was done in terms of bounding $y^{\max}$, which places an obvious limit of $Sy^{\max}$ on the total unbalance as summed over all the buckets. I ran some experiments to find out just what the actual total unbalance was; the results are shown in Table B.7. It appears in comparing the maximum unbalance with $Sy^{\max}$ that achieving a total of unbalance of $Sy^{\max}$ requires that D be above some critical value that is a function of S. The few critical values we can see so far are summarized:

S	3	4	5
Critical # disks	2	3	4

from which it appears that the critical number of disks may be $S - 1$.

		Min. tracks			Max. unbalance			$Sy^{\max}$		
		D			D			D		
		2	3	4	2	3	4	2	3	4
	3	1	1	1	2	3	3	3	3	3
	4	5	7	6	6	8	8	8	8	8
	5	4	8	7	6	9	10	10	10	10
S	6	14	27		12	16		18	18	
	7	12			12			21	21	
	8	30			20			32		
	9	26			20			36		

Table B.7: The total unbalance achieved as a function of S and D.

		Normal # configurations			Birepresentative # configurations		
		D			D		
		2	3	4	2	3	4
	3	2	4	5	2	4	5
S	4	6	36	207	6	35	206
	5	7	84	1132	7	84	1128
	6	29	4060		29	4058	

Table B.8: The number of configurations reachable for normal and birepresentative tracks.

I also wanted to check a couple of hypotheses. The first was the supposition that there exists a track-minimal attainment for any values of D and S that uses only birepresentative tracks. This hypothesis is clearly trivial unless $D > 2$. All of the results that I was able to simulate did in fact give the same number of tracks for a minimal attainment. However, the total number of configurations reachable was slightly smaller when only birepresentative tracks were used, as shown in Table B.8. Thus, this experiment does not give a very strong indication in either direction whether a track-minimal birepresentative attainment exists.

The other hypothesis I was interested in checking was whether we needed to pay the cost of finding a globally min-cost matching, or whether a min-cost matching that is a local minimum with respect to swaps of assignments would suffice. Of all the lemmas and theorems in this paper, only Theorems 36 and 37 require a global min-cost algorithm. When $D = 2$, a local min-cost is a global min-cost, so I ran my experiments using $D = 3$. In the case where $S = 6$, the local min-cost achieved a total column sum of 9, as opposed to 8 for a global min-cost. Similarly, the number of configurations reachable was 4321 instead of 4060. So it seems that using a local min-cost might affect the worst-case performance of the algorithm.

B.5.2 Average-case behavior

To determine what kind of performance the algorithm is expected to have in practice, I wrote a second program that generated a random permutation of N elements and then tried to distribute the buckets evenly among the D disks, assuming that the elements were distributed among the S buckets as evenly as possible (i.e., each bucket contained either $\lfloor \frac{N}{S} \rfloor$ or $\lceil \frac{N}{S} \rceil$ elements). I then ran this algorithm over a systematic data set:

$$\begin{aligned} S &= \{5, 10, 20, 50, 100, 200\} \\ D &= \{5, 10, 20, 40, 80\} \\ T &= \{50, 100, 250, 500, 1000, 2000\}. \end{aligned}$$

Here, T = number of tracks, so that $N = DT$. For speed of running, the algorithm used only a local min-cost function, so the actual algorithm with a real min-cost could quite possibly be better than the simulation indicates. The program was also able to compare the actual distribution with the optimal one and compute the ratio. The worst ratio to the optimal was $\{S, D, T\} = \{100, 80, 100\}$, which yielded a ratio of 1.775. This particular set of values is unrealistic in the sense that each bucket comprised a single track (so the ratio can be interpreted as saying that 77% of the buckets were spread between two tracks). If we say that each bucket should comprise at least two tracks, then the worst ratio occurred when $\{S, D, T\} = \{50, 80, 100\}$, with a ratio of 1.354. Finally, if we impose the very reasonable restriction that $S \leq \sqrt{T}$, also considered in the theoretical part of the study, then the worst ratio obtained was when $\{S, D, T\} = \{10, 40, 100\}$, with a ratio of 1.042. Thus, the algorithm seems to balance the buckets extremely well in practice, with a total unbalance never more than 5% above optimal. It is expected that using a real min-cost instead of an approximate one would yield even better results.

Bibliography

[AAC] Alok Aggarwal, Bowen Alpern, Ashok K. Chandra, and Marc Snir, "A Model for Hierarchical Memory," *Proceedings of 19th Annual ACM Symposium on Theory of Computing* (May 1987), 305–314.

[ACSa] Alok Aggarwal, Ashok K. Chandra, and Marc Snir, "Hierarchical Memory with Block Transfer," *Proceedings of 28th Annual IEEE Symposium on Foundations of Computer Science* (October 1987), 204–216.

[AgP] Alok Aggarwal and James Park, "Notes on Searching in Multidimensional Monotone Arrays," *Proceedings of 29th Annual IEEE Symposium on Foundations of Computer Science* (October 1988), 497–512.

[AgV] Alok Aggarwal and Jeffrey Scott Vitter, "The Input/Output Complexity of Sorting and Related Problems," *Communications of the ACM* (September 1988), 1116–1127.

[Akl] S. G. Akl, *Parallel Sorting Algorithms*, Notes and Reports in Computer Science and Applied Mathematics #12, Academic Press, Inc., Orlando, 1985.

[AlR] Romas Aleliunas and Arnold L. Rosenberg, "On Embedding Rectangular Grids in Square Grids," *IEEE Transactions on Computers* C-31 (1982), 907–913.

[ACF] Bowen Alpern, Larry Carter, and Ephraim Feig, "Uniform Memory Hierarchies," *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science* (October 1990), 600–608.

[ACSb] Bowen Alpern, Larry Carter, and Ted Selker, "Visualizing Computer Memory Architectures," *Proceedings of the First Annual IEEE Conference on Visualization* (October 1990), 107–113.

[Amd] G. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *Proceedings AFIPS 1967 Spring Joint Computer Conference* (April 1967), 483–485.

[Bai] David H. Bailey, "FFTs in External or Hierarchical Memory," *J. Supercomputing* 4 (1990), 23–35.

[Ber] Claude Berge, *Graphs and Hypergraphs* (second edition), North-Holland Publishing Company, Amsterdam, 1976.

[BKT] P. Berman, H. J. Karloff, and G. Tardos, "A Competitive 3-Server Algorithm," *Proceedings of the First Annual IEEE Symposium on Discrete Algorithms* (January 1990), 280–290.

[BDH] Dina Bitton, David J. DeWitt, David K. Hsiao, and Jaishankar Menon, "A Taxonomy of Parallel Sorting," *Computing Surveys* 16 (September 1984), 287–318.

[BFP] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan, "Time Bounds for Selection," *J. Computer and System Sciences* 7 (1973), 448–461.

[BIR] Allan Borodin, Sandy Irani, Prabhakar Raghavan, and Baruch Schieber, "Competitive Paging with Locality of Reference," *Proc. of the 23rd ACM Symposium on Theory of Computing* (May 1991), 249–259.

[Car] David A. Carlson, "Using Local Memory to Boost the Performance of FFT Algorithms on the CRAY-2 Supercomputer," *J. Supercomputing* 4 (1990), 345–356.

[CGK] Peter Chen, Garth Gibson, Randy H. Katz, David A. Patterson, and Martin Schulze, "Two Papers on RAIDs," U. C. Berkeley, UCB/CSD 88/479, December 1988.

[Che] Peter M. Chen, "An Evaluation of Redundant Arrays of Disks using an Amdahl 5890," U. C. Berkeley, UCB/CSD 89/506, May 1989.

[CKP] M. Chrobak, H. Karloff, T. Payne, and S. Vishwanathan, "New Results on Server Problems," *Proceedings of the First Annual IEEE Symposium on Discrete Algorithms* (January 1990), 291–300.

[CRS] F.R.K. Chung, A. L. Rosenberg, and Lawrence Snyder, "Perfect Storage Representations for Families of Data Structures," *SIAM J. Algorithms and Discrete Methods* 4 (1983), 548–565.

[Col] Richard Cole, "Parallel Merge Sort," *SIAM J. Computing* 17 (August 1988), 770–785.

[Coo] Stephen A. Cook, "An Observation on Time-Storage Trade Off," *Proceedings of the 5th Annual ACM Symposium on Theory of Computation* (May 1973), 29–33.

[Cor] Thomas H. Cormen, "Fast Permuting on Disk Arrays," *Proc. Brown/MIT Conference on Advanced Research in VLSI and Parallel Systems* (March 1992), 58–76.

[Cyp] Robert Cypher, "Theoretical Aspects of VLSI Pin Limitations," IBM Almaden Research Center, Research Report RJ 7115 (67362), November 1989.

[CyP] Robert Cypher and C. Greg Plaxton, "Deterministic Sorting in Nearly Logarithmic Time on the Hypercube and Related Computers," *Journal of Computer and System Sciences* (to appear), also appears in *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, (May 1990), 193–203.

[DEL] Richard A. DeMillo, Stanley C. Eisenstat, and Richard J. Lipton, "Preserving Average Proximity in Arrays," *Communication of the ACM* 21 (1978), 228–231.

[Dew] A. K. Dewdney, "Computer Recreations," *Scientific American* 252 (May 1985), 18–30.

[Eve] Shimon Even, "Parallelism in Tape-Sorting," *Communications of the ACM* 17 (April 1974), 202–204.

[FKL] Amos Fiat, Richard M. Karp, Michael Luby, Lyle A. McGeoch, Daniel D. Sleator, and Neal E. Young, "Competitive Paging Algorithms," Carnegie-Mellon University, CMU-CS-88-196, November 1988.

[Flo] Robert W. Floyd, "Permuting Information in Idealized Two-Level Storage," in *Complexity of Computer Computations*, R. Miller and J. Thatcher, ed., Plenum, 1972, 105–109.

[FrW] Michael L. Fredman and Dan E. Willard, "BLASTING Through the Information Theoretic Barrier with FUSION TREES," *Proc. of the 22nd Annual ACM Symposium on the Theory of Computing* (May 1990), 1–7.

[Fri] D. Bitton Friedland, "Design, Analysis, and Implementation of Parallel External Sorting Algorithms," U. Wisconsin — Madison, Ph.D. Dissertation, 1981.

[GaJ] Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., New York, 1979.

[GHK] Garth Gibson, Lisa Hellerstein, Richard M. Karp, Randy H. Katz, and David A. Patterson, "Coding Techniques for Handling Failures in Large Disk Arrays," U. C. Berkeley, UCB/CSD 88/477, December 1988.

[GiS] David Gifford and Alfred Spector, "The TWA Reservation System," *Communications of the ACM* 27 (July 1984), 650–665.

[GoS] Mark Goldberg and Thomas Spencer, "Constructing a Maximal Independent Set in Parallel," *SIAM J. Discrete Math* 2, 322–328.

[GKP] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik, in *Concrete Mathematics*, Addison-Wesley, Reading, MA, 1989, Chapter 4.

[HoK] Jia-Wei Hong and H. T. Kung, "I/O Complexity: The Red-Blue Pebble Game," *Proc. of the 13th Annual ACM Symposium on the Theory of Computing* (May 1981), 326–333.

[Ihl] Edmund Ihler, "Bounds on the Quality of Approximate Solutions to the Group Steiner Problem," in *Graph-Theoretic Concepts in Computer Science, Proceedings of the 16th International Workshop WG '90*, G. Goos and J. Hartmanis, eds., Lecture Notes in Computer Science #484, Springer-Verlag, Berlin, 1991, 109–118.

[IsS] Amos Israeli and Y. Shiloach, "An Improved Parallel Algorithm for Maximal Matching," *Information Processing Letters* 22 (1986), 57–60.

[Jil] W. Jilke, "Disk Array Mass Storage Systems: The New Opportunity," Amperif Corporation, September 1986.

[Knu] Donald E. Knuth, in *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[KSS] Steven D. Kugelmass, Richard Squier, and Kenneth Steiglitz, "Performance of VLSI Engines for Lattice Computations," *Complex Systems* 1 (October 1987), 939–965.

[Lei] Tom Leighton, "Tight Bounds on the Complexity of Parallel Sorting," *IEEE Transactions on Computers* C-34 (April 1985), 344–354, also appears in *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, (April 1983), 71–80.

[LiV] Eugene E. Lindstrom and Jeffrey Scott Vitter, "The Design and Analysis of BucketSort for Bubble Memory Secondary Storage," *IEEE Transactions on Computers* C-34 (March 1985), 218–233.

[LED] R. J. Lipton, S. C. Eisenstat, and R. A. DeMillo, "Space and Time Hierarchies for Classes of Control Structures and Data Structures," *J. Association for Computing Machinery* 23 (1976), 720–732.

[Luba] Michael Luby, "A Simple Parallel Algorithm for the Maximal Independent Set Problem," *SIAM J. Computing* 15 (1986), 1036–1053.

[Lubb] Michael G. Luby, "Removing Randomness in a Parallel Computation Without a Processor Penalty," International Computer Science Institute, TR-89-044, July 1989, also appears in *Proceedings of the 29th Annual IEEE Symposium on the Foundations of Computer Science*, (October 1988), 162–173.

[LuP] F. Luccio and L. Pagli, "Sequential Computation Based on Pipelined Access to Memory," *Proceedings of the 27th Annual Allerton Conference on Communication, Control, and Computing* (September 1989), 702–711.

[Mag] Ninamary Buba Maginnis, "Store More, Spend Less: Mid-Range Options Around," *Computerworld* (November 16, 1987), 71–82.

[Man] Swaminathan Manohar, "Supercomputing with VLSI," Brown Univ., Ph.D. Thesis, 1989.

[MaT] Norman Margolus and Tommaso Toffoli, "Cellular Automata Machines," *Complex Systems* 1 (October 1987), 967–993.

[MGS] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal* 9 (1970), 78–117.

[McS] Lyle A. McGeoch and Daniel D. Sleator, "A Strongly Competitive Randomized Paging Algorithm," Carnegie-Mellon University, CMU-CS-89-122, March 1989.

[NoVa] Mark H. Nodine and Jeffrey Scott Vitter, "Greed Sort: An Optimal External Sorting Algorithm for Multiple Disks," Brown University, CS-91-20, August 1991, also appears in shortened form in "Large-Scale Sorting in Parallel Memories," *Proc. 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, Hilton Head, SC (July 1991), 29–39.

[NoVb] Mark H. Nodine and Jeffrey Scott Vitter, "Optimal Deterministic Sorting on Parallel Memory Hierarchies," Department of Computer Science, Brown University, CS-92-38, August 1992.

[NoVc] Mark H. Nodine and Jeffrey Scott Vitter, "Optimal Deterministic Sorting on Parallel Disks," Department of Computer Science, Brown University, CS-92-08, August 1992.

[PGK] David A. Patterson, Garth Gibson, and Randy H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proceedings ACM SIGMOD Conference* (June 1988), 109–116.

[PeS] James L. Peterson and Abraham Silberschatz, in *Operating System Concepts*, Addison-Wesley Publishing Company, Reading, MA, 1983.

[Pou] William Poundstone, *The Recursive Universe: Cosmic Complexity and the Limits of Scientific Knowledge*, Contemporary Books, Chicago, IL, 1985.

[PLK] K. Wojtek Przytula, Wei-Ming Lin, and V.K. Prassana Kumar, "Partitioned Implementation of Neural Networks in Mesh Connected Array Processors," *VLSI Signal Processing IV: Proceedings of IEEE VLSI For Signal Processing Workshop* (November 1990), 106–115.

[RaS] Prabhakar Raghavan and Marc Snir, "Memory Versus Randomization in On-line Algorithms," in *Proc. of the 16th International Colloquium on Automata, Languages, and Programming*, G. Goos and J. Hartmanis, eds., Lecture Notes in Computer Science #372, Springer-Verlag, Berlin, July 1989, 687–703.

[RaR] Sanguthevar Rajasekaran and John H. Reif, "Optimal and Sublogarithmic Time Randomized Parallel Sorting Algorithms," *SIAM J. Computing* 18 (1989), 594–607.

[ReW] Gabriele Reich and Peter Widmayer, "Beyond Steiner's Problem: A VLSI Oriented Generalization," in *Graph-Theoretic Comcepts in Computer Science: Proceedings of the 15th International Workshop WG '89*, G. Goos and J. Hartmanis, eds., Lecture Notes in Computer Science #411, Springer-Verlag, New York, 1990, 196–210.

[Rosa] Arnold L. Rosenberg, "Preserving Proximity in Arrays," *SIAM J. Comput.* 4 (1975), 443–460.

[Rosb] Arnold L. Rosenberg, "Data Encodings and Their Costs," *Acta Informatica* 9 (1978), 273–292.

[Rosc] Arnold L. Rosenberg, "Encoding Data Structures in Trees," *J. Association for Computing Machinery* 26 (1979), 668–689.

[RoS] Arnold L. Rosenberg and Lawrence Snyder, "Bounds on the Costs of Data Encodings," *Math. Systems Theory* 12 (1978), 9–39.

[SaV] John Savage and Jeffrey Scott Vitter, "Parallelism in Space-Time Tradeoffs," in *Advances in Computing Research, Volume 4*, F. P. Preparata, ed., JAI Press, 1987, 117–146, Also appears in *Proceedings of the International Workshop on Parallel Computing and VLSI*, Amalfi, Italy (May 1984), P. Bertolazzi and F. Luccio, ed., Elsevier Science Press, 1985, 49–58.

[Sil] Gabriel M. Silberman, "Delayed-Staging Hierarchy Optimization," *IEEE Transactions on Computers* C-32 (November 1983), 1029–1037.

[SlT] Daniel D. Sleator and Robert E. Tarjan, "Amortized Efficiency of List Update and Paging Rules," *Communications of the ACM* 28 (February 1985), 202–208.

[TML] A. P. Thakoor, A. Moopenn, John Lambe, and S. K. Khanna, "Electronic Hardware Implementations of Neural Networks," *Applied Optics* 26, 5085–5092.

[ToM] Tomasso Toffoli and Norman Margolus, *Cellular Automata Machines: A New Environment for Modeling*, MIT Press, Cambridge, MA, 1987.

[Ull] Jeffrey D. Ullman, in *Principles of Database and Knowledge-Base Systems*, Computer Science Press, Rockville, MD, 1988.

[UlY] Jeffrey D. Ullman and Mihalis Yannakakis, "The Input/Output Complexity of Transitive Closure," *Annals of Mathematics and Artificial Intelligence* (1991), 331–360.

[Uni] University of California at Berkeley, "Massive Information Storage, Management, and Use (NSF Institutional Infrastructure Proposal)," Technical Report No. UCB/CSD 89/493, January 1989.

[ViN] Jeffrey Scott Vitter and Mark H. Nodine, "Large-Scale Sorting in Uniform Memory Hierarchies," *Journal of Parallel and Distributed Computing* (January 1993), also appears in shortened form in "Large-Scale Sorting in Parallel Memories," *Proc. 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, Hilton Head, SC (July 1991), 29–39.

[ViSa] Jeffrey Scott Vitter and Elizabeth A. M. Shriver, "Optimal Disk I/O with Parallel Block Transfer," *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing* (May 1990), 159–169.

[ViSb] Jeffrey Scott Vitter and Elizabeth A. M. Shriver, "Algorithms for Parallel Memory I: Two-Level Memories," Brown University, CS-90-21, September 1990.

[ViSc] Jeffrey Scott Vitter and Elizabeth A. M. Shriver, "Algorithms for Parallel Memory II: Hierarchical Multilevel Memories," Brown University, CS-90-22, September 1990.

[WHA] Mark Walker, Paul Hasler, and Lex Akers, "A CMOS Neural Network for Pattern Association," *IEEE Micro* 9 (October 1989), 68–74.