Parallel Out-of-Core Sorting: The Third Way

Geeta Chaudhry
Dartmouth College

Dartmouth College Computer Science Department Technical Report TR2004-517

Parallel Out-of-Core Sorting: The Third Way

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

in

Computer Science

by

Geeta Chaudhry

DARTMOUTH COLLEGE

Hanover, New Hampshire

March 12th, 2004

Examining Committee:

(chair) Thomas H. Cormen

Doug McIlroy

Bill McKeeman

Lars Arge

Carol Folt
Dean of Graduate Students

Abstract

Sorting very large datasets is a key subroutine in almost any application that is built on top of a large database. Two ways to sort out-of-core data dominate the literature: merging-based algorithms and partitioning-based algorithms. Within these two paradigms, all the programs that sort out-of-core data on a cluster rely on assumptions about the input distribution.

We propose a third way of out-of-core sorting: oblivious algorithms. In all, we have developed six programs that sort out-of-core data on a cluster. The first three programs, based completely on Leighton's columnsort algorithm, have a restriction on the maximum problem size that they can sort. The other three programs relax this restriction; two are based on our original algorithmic extensions to columnsort. We present experimental results to show that our algorithms perform well. To the best of our knowledge, the programs presented in this thesis are the first to sort out-of-core data on a cluster without making any simplifying assumptions about the distribution of the data to be sorted.

To my parents.

Acknowledgments

First and foremost, I thank Tom Cormen, my thesis advisor, in whom I have gained a great friend, colleague, and mentor. You have given me a perfect mixture of guidance and freedom. Thanks for always encouraging me to think out of the box, and at the same time ensuring that I make consistent progress in research. Thank you for your patience every time you listened to some of my crazy ideas before pointing out the flaws (that you most probably saw right away)! I can not possibly exaggerate your role in making me into a much better writer. You made my experience at Dartmouth as well as my experience as a PhD student a strongly positive one.

Thanks to both Doug McIlroy and Bill McKeeman, the internal committee members of my thesis, for contributing great insights to my work. I also thank you for sharing the enthusiasm for my research area, it meant a lot to me. Doug, special thanks to you for the numerous long discussions over the course of my research.

I thank Lars Arge, for agreeing to be on my committee, and for taking the time to come here all the way from Duke. I also thank you for your insightful comments about various written sections of my thesis.

Several people have helped me in the process of completing my thesis research;

I thank them all. Special thanks to Elena Davidson, Michael Fromberger, Elizabeth Hamon, David Kotz, Cliff Stein, Tim Tregubov, Len Wisniewski, and Tony Yan.

Several of my teachers have had a profound influence on my attitude toward learning, research, scholarship, and teaching. Tom, my advisor, Prasad Jayanti, Achyut Roy, and Hari Sahasrabuddhe, I thank you all. You have played a very important role in shaping my academic attitude.

I thank two of my high school teachers - Kiran Bist, and Uma Jain, for always encouraging me to participate in every single activity in school, and for making me believe that I could do whatever I wanted. Kiran didi - special thanks to you for encouraging me to take part in all those debate competitions every year, in spite of my obnoxious reluctance to work for them.

I want to take this opportunity to thank some people without whom this journey would not be the same. Sabina, Shivani, Vinu, and Viraj, I thank you for being my closest and best friends through these years. Without you, I might have forgotten what a beautiful thing true friendship is.

There is no way I could have done this without the love and encouragement of my parents. Amma and Papa, I thank you for trusting me, believing in me, and teaching me the art of living. Your role in getting me so far is beyond words. Mama and Tata, my husband's parents, I thank you for your loving concern and support through the last few years. Raju bhaiya, my elder brother, I thank you for your love, your confidence in me and all the inexplicably strong support you provided when I came to USA. Shailja bhabhi, my adorable sister-in-law, thanks for being my friend and sister, at some of the most important times of my life. Anand, what can I say about you, you are my brother, friend, critic, guide, and confidant all in one; I thank you for always being there for me.

It has been several years of work, this thesis. Srdjan, my husband, you were the single most important person in making this time the best most wonderfully joyous time of my life. I thank you with all my love, for being a part of my life.

Contents

Chapter 1

Introduction

We explore the use of oblivious sorting algorithms to sort out-of-core data on distributed-memory clusters. This chapter discusses our motivation for using oblivious algorithms, touches on some related work, and summarizes the contributions.

Out-of-core sorting refers to the problem of sorting when the amount of data exceeds the capacity of main memory. Such out-of-core data then typically resides on one or more disks. As a result, the time required to transfer data between memory and disks (relative to the cost of computation) can be a major performance bottleneck. For large-scale applications, therefore, it is essential to design and engineer algorithms that minimize both CPU times and disk-I/O times [Vit91]. Furthermore, in a multiprocessor environment, the algorithms also need to minimize communication time.

Sorting very large datasets is a key subroutine in almost any application that is built on top of a large database [SGV03]. Geographical information systems, seismic modeling, and web-search engines are a few example applications that store, sort, and search through enormous amounts of data. The study of out-of-core sorting goes at least as far back as 1956 [Fri56]. In addition to being a database operation in its own right, sorting is a key subroutine in several other database operations [Ull99a, Ull99b]. We describe some of these operations in Section 1.4.

Our goal is to design robust out-of-core sorting algorithms for a distributed-memory cluster and to develop their implementations using off-the-shelf software for both I/O and communication. Because distributed-memory clusters [ACPtNt95, BBH99, BG02] offer a good price-performance ratio, they have become a popular platform for high-performance computing.

Two ways to sort out-of-core data dominate the literature: merging-based algorithms and partitioning-based algorithms. An algorithm belonging to either of these paradigms has the property that, when adapted to the out-of-core setting of a distributed-memory cluster, the resulting I/O and communication patterns vary depending on the data to be sorted. In other words, the I/O and communication patterns are not known in advance. Because the patterns are not known in advance, it becomes difficult for the programmer to plan an efficient implementation; much of the programming effort is spent in handling data that might lead to "bad" I/O and communication patterns. In fact, except for the work presented, all the programs that sort out-of-core data on a cluster make simplifying assumptions about the data, thereby avoiding the issue of data instances that might lead to bad I/O and communication patterns.[1]

We propose a third way of out-of-core sorting: oblivious algorithms.[2] As discussed later in this chapter, the paradigm of oblivious algorithms offers some distinct advantages over the existing paradigms. Not surprisingly, it has some down-

[1] All of these programs are partitioning-based.

[2] Knuth discusses oblivious algorithms as sorting networks in Chapter 5 of [Knu98].

sides as well. Here, we document and analyze several sorting programs that we have developed using this third paradigm. In all, we have developed six programs that sort out-of-core data on a cluster. Three of them are based completely on columnsort [Lei85]; they differ in their adaptation to the out-of-core setting. The first three programs have a restriction on the maximum problem-size that they can sort. The other three programs relax this restriction; two are based on our original algorithmic extensions to columnsort. To the best of our knowledge, the programs presented here are the first to sort out-of-core data on a cluster without making any simplifying assumptions about the distribution of the data to be sorted. We do assume that all records are the same size and that the time to compare two records depends only on the record size.

1.1 Why oblivious algorithms?

It is a significant task to implement a fast out-of-core sorting algorithm for a cluster. Overlapping I/O, communication, and computation often forms the bulk of the complexity of this task. We believe that oblivious algorithms, as compared with both merging-based and distribution-based algorithms, are much more amenable to implementations that allow efficient overlapping.

Any out-of-core application on a cluster of machines typically has three significant costs: I/O, communication, and computation. For good performance, it is not enough to keep each of these individual costs low. For example, it is possible for an implementation A to take $a + b + c$ units of time to complete, where a is the optimum I/O time, b is the optimum communication time, and c is the optimum computation time. If there were no interdependence among the three operations, then A could overlap the three operations, since they use three different resources: disk, network, and the CPU. Achieving maximum overlap would allow A to complete in $\max(a, b, c)$ units of time. Of course, there is always a significant amount of interdependence among I/O, communication, and computation, making it very difficult to achieve maximum overlap.[3]

The amount of overlap actually achieved in an out-of-core sorting program depends on two factors: the opportunity for overlap in the underlying algorithm, and the software used to achieve this overlap. For a given algorithm, it is really an implementation issue whether the program can actually achieve the maximum overlap, but choosing an algorithm that has ample opportunity for overlap is a design issue.

The input to an out-of-core sorting algorithm is a file of *records*, where each record contains a *key* according to which the records are to be sorted. An algorithm whose I/O and communication patterns depend on the input key distribution makes the task of overlapping difficult. Such an algorithm has the additional challenge of beating bad key distributions. When the I/O and communication patterns depend on the input, it is hard to predict (at coding time) the best way to overlap I/O, communication, and computation. Not surprisingly, the three existing implementations of out-of-core sorting for a cluster—all distribution-based—lack the robustness that any practical algorithm must have. These implementations assume that the keys are uniformly distributed. To quote the authors of NOW-Sort [ADADC$^+$97]:

> In this study, we make a number of simplifying assumptions about the distribution of key values . . . Clearly, implementations used for sorting

[3]When we say maximum overlap, we mean as much overlap as possible, for a given algorithm on a given input.

datasets in the real world would need to be more robust.

An *oblivious sorting algorithm* is a comparison-based sorting algorithm in which the sequence of comparisons is predetermined [Lei92]. In other words, the result of a comparison has no effect on the indices of elements compared later on. Knuth [Knu98, pp. 220–221] uses the term *homogeneous* for such comparison sequences. When adapted to an out-of-core setting, the I/O and communication patterns of such an algorithm should be independent of the input keys. As mentioned in Section 5.3.4 of [Knu98], for large problem sizes, an oblivious algorithm may have to do asymptotically more comparisons than a non-oblivious algorithm. Our algorithms, however, are only partially oblivious; the in-core sorting components of our implementations are not oblivious.[4] Restricting the non-obliviousness to the in-core computation parts allows our algorithms to have predetermined I/O and computation patterns, yet to perform fewer comparisons than a fully oblivious algorithm.

1.2 Related work: The first two ways

As one would expect for a fundamental problem like sorting, the research community has investigated it extensively. Several thorough papers catalog out-of-core algorithms [Arg97, ATV01, CFM$^+$98, CGG$^+$95, GTVV93, Vit01]. This section summarizes previous work in developing out-of-core sorting algorithms under both the merging-based and the partitioning-based paradigms. Chapter 8 presents a more detailed discussion.

[4] In an oblivious algorithm, any oblivious subroutine a can be replaced by a non-oblivious subroutine b, as long as b produces the same output as a.

1.2.1 Merging-based algorithms

Merging-based algorithms are out-of-core versions of the more familiar in-core merge sort. There are two main challenges in designing a merging-based, out-of-core, sorting algorithm for a distributed-memory cluster. The primary challenge is to ensure good data locality when the sorted runs are merged. Even on a uniprocessor system, the sequence of disk reads and writes performed during merging depends on the input keys. The second challenge is to parallelize the merging of data in a distributed memory; we know of no efficient algorithm that does so. It is not surprising, therefore, that the known implementations [BV02, DS03, Pea99, RJ00] of merging-based algorithms run only on a single processor. Iyer and Dias [ID90] present experimental data showing that, even on tightly-coupled processors, data skew can seriously affect the performance of external merge sort.

1.2.2 Partitioning-based algorithms

The paradigm of partitioning-based algorithms is based upon adapting the sequential quicksort algorithm to an out-of-core setting. When designing a partitioning-based, out-of-core, sorting algorithm, the challenge is to ensure that the amount of data in each partition is approximately equal. An uneven partitioning requires load balancing, which in turn requires additional communication and possibly additional I/O. It is difficult to find the exact partitioning elements (leading to a perfectly load-balanced partition) in an I/O-efficient way [Vit01].

Even when the partitions are approximately equal in size, the I/O and communication patterns are highly dependent on the keys. Suppose that there are P processors and that each has an internal memory that can hold M/P records, where

M is the size of the memory, in records, of the entire system. Since the data is out-of-core, the distribution step proceeds in a number of rounds, where each round distributes one *memoryload* (i.e., M records, with M/P records per processor). In the memoryload of a given round, it is impossible to know in advance the number of records that belong to the partition of any given processor. Let S_i be the number of records that belong to the partition mapped to processor i. For any i, the value of S_i affects both the memory usage and the communication pattern, since processor i needs to receive its partition and it needs to have enough local memory to store it. Also, S_i affects the I/O pattern, since processor i needs to write out S_i records back to disk.

The dependence of I/O and communication patterns is undesirable for two reasons. First, it makes it difficult for the programmer to plan the best way to overlap I/O, communication, and computation. Second, if $S_i \gg M/P$ for some i, communication and I/O become highly unbalanced.

The literature describes several partitioning-based algorithms. Some of them are deterministic [AV88, BHJ96, HJB98, NV93], and others are randomized [DNS91, HBJ98, HJ97, VS94]. In any case, an out-of-core sorting algorithm using these methods must have a provision for recovering from a bad partition. To date, we do not know of any out-of-core sorting implementation for clusters that uses any of these methods.

1.3 The third way: Contributions

In marked contrast to the two dominant paradigms of merging and partitioning, both of which have unpredictable I/O and communication patterns, we offer the third paradigm of oblivious algorithms. With this paradigm, we have achieved our goal: robust[5] out-of-core sorting on a distributed-memory cluster using off-the-shelf software. An oblivious algorithm in an out-of-core setting leads to predetermined I/O and communication patterns, allowing excellent opportunity for both planning and achieving good overlap among I/O, communication, and computation. We have begun to explore this third paradigm by designing and implementing several algorithms that sort out-of-core data on clusters. Figure 1.1 summarizes the chronology of our six implementations. The following list summarizes the key contributions of our work:

- As a first step, we have designed and implemented three out-of-core sorting programs [CC02, CCW01] for distributed-memory clusters: *non-threaded 4-pass columnsort*, *4-pass columnsort*, and *3-pass columnsort*. Each of these three programs is based on Leighton's columnsort [Lei85], an oblivious sorting algorithm. Non-threaded 4-pass columnsort, our first implementation, is discussed in detail in [CCW01], where we compare it with NOW-Sort [ADADC$^+$97]. The 4-pass columnsort implementation is a threaded version of non-threaded 4-pass columnsort. The 3-pass columnsort implementation improves 4-pass columnsort by reducing the number of passes. Each *pass* consists of reading each record once from disk, working on it in memory, and writing it back to disk. Chapter 3 discusses these implementations in detail. In the rest of the thesis, we use the term *preliminary implementations* to refer to these first three implementations.

[5]By a robust algorithm, we mean an algorithm that is robust against bad key distributions.

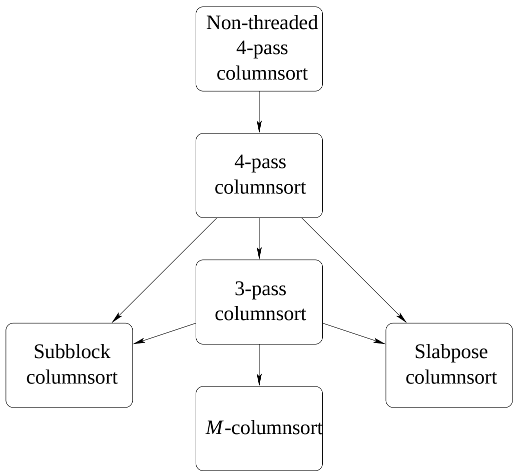

Figure 1.1: Our six out-of-core sorting programs. An arrow from Program A to Program B implies that Program B was developed as an extension or modification of Program A. Except for non-threaded 4-pass columnsort, all the programs use threads to achieve asynchrony.

Over the course of last few years, we have conducted several experiments of our programs. We present four sets of results, from experiments on four different platforms. We shall refer to these platforms as *Sun-A*, *Sun-B*, *Borg*, and *Jefferson*.[6] On Sun-A and Sun-B, we found non-threaded 4-pass columnsort to be competitive with NOW-Sort [CCW01]. Also, as discussed in [CC02] 4-pass columnsort reduces the running time to approximately half of the non-threaded version. The reduction of the number of passes down to three in 3-pass columnsort showed mod-

[6]The platforms Sun-A and Sun-B were at the Sun Microsystems office in Burlington, Massachusetts. Borg and Jefferson are two Beowulf clusters that belong to ISTS (Institute for Security Technology Studies) and the Computer Science Department at Dartmouth, respectively.

est performance gains (5–9%) on Sun-A and Sun-B. On jefferson, however, 3-pass columnsort is about 25% faster than the 4-pass version. As we shall discuss later, this difference is a result of Jefferson being more IO-bound as compared to the two Sun clusters.

Although our preliminary implementations are robust, they suffer from a restriction on the maximum problem size that they can sort. Columnsort is an 8-step algorithm that sorts N records arranged as an $r \times s$ mesh, where $N = rs$, subject to the following two restrictions:

1. The *divisibility restriction*: s divides r.
2. The *height restriction*: $r \geq 2s^2$.

In our adaptations of columnsort to an out-of-core setting, we set r to be M/P, where M is the overall memory size, in records, over the entire cluster, and P is the number of processors in the cluster. Therefore, M/P is the number of records that a single processor can hold in its memory. We refer to this setting of r, the height of the mesh, to M/P as the *height interpretation*. Together, the height restriction and the height interpretation lead to the following *problem-size restriction*, which limits the maximum number of records N that we can sort:

$$N \leq \frac{(M/P)^{3/2}}{\sqrt{2}} . \qquad (1.1)$$

The height restriction is an algorithmic issue, i.e., it is a part of the underlying algorithm. The height interpretation, on the other hand, is an implementation issue. We have attempted to relax the problem-size restriction by attacking both the height restriction and the height interpretation, employing algorithmic and practical means respectively. The next four items in the list of contributions summarize the results of these efforts.

- We show that the divisibility restriction is unnecessary.
- We present *subblock columnsort*, an oblivious algorithm that relaxes the height restriction by a factor of $\sqrt{s}/2$, to $r \geq 4s^{3/2}$, by adding two steps to columnsort. With a height interpretation of $r = M/P$, we get a problem-size restriction of
$$N \leq \frac{(M/P)^{5/3}}{4^{2/3}} . \tag{1.2}$$
This improvement in problem size can be quite substantial in an out-of-core setting. For most current systems ($M/P \geq 2^{12}$ records), this change enables us to more than double the largest problem size. The two additional steps of subblock columnsort add an extra pass to the three passes of 3-pass columnsort.[7] Since we tested subblock columnsort only on Borg and Jefferson (both IO-bound), it did not come as a surprise that subblock columnsort, with its 4 passes, runs almost as fast as 4-pass columnsort. In other words, subblock columnsort is about 33% slower compared to 3-pass columnsort.
- *M-columnsort* changes the height interpretation from $r = M/P$ to $r = M$, thus leading to the improved problem-size restriction
$$N \leq \frac{M^{3/2}}{2} . \tag{1.3}$$
On a cluster with 16 processors, with $M/P = 2^{19}$ records, this change al-

[7] Since 3-pass columnsort is the fastest of the preliminary implementations, we use it as a baseline for all subsequent programs.

lows us to sort up to one terabyte of data, assuming a record size of 64 bytes. Although M-columnsort adds no extra passes, it does incur substantial amounts of communication and additional computation when compared to 3-pass columnsort. On both Borg and Jefferson, M-columnsort performed slower than 3-pass columnsort due to its increased computation and communication. On Borg, M-columnsort runs faster than subblock columnsort in all cases. On Jefferson, on the other hand, M-columnsort is much slower than subblock columnsort, since Jefferson is not quite as IO-bound as Borg is.

- Finally, we present *slabpose columnsort*, a new, oblivious 10-step algorithm that relaxes the problem-size restriction without adding any communication or I/O costs to 3-pass columnsort. Slabpose columnsort relaxes the height restriction to $r \geq (2s^2/k)(\lceil k^2/s \rceil + 1)$, where $k < s$. In our implementation, we set $k = P$. With a height interpretation of $r = M/P$, we get a problem-size restriction of

$$N \leq \begin{cases} (M/P)^{3/2} \cdot \dfrac{\sqrt{P}}{2} & \text{if } s/P \geq P, \\ \dfrac{(M/P)^2}{4P} & \text{if } s/P < P . \end{cases}$$

  Chapter 5 discusses the implications of this problem-size restriction in detail. In most cases, it is an improvement over the original problem-size restriction.

  Experimental results on Jefferson show that, as expected, the relaxed problem-size restriction of slabpose columnsort comes at almost no loss in performance: slabpose columnsort runs almost as fast as 3-pass columnsort.

- Our implementations use only standard, off-the-shelf software: MPI [SOHL$^+$98, GHLL$^+$98] for communication and UNIX file system calls for I/O.

- There are no assumptions required about the keys. In fact, the I/O and communication patterns are oblivious to the keys in each of our programs. To the best of our knowledge, our implementations are the first to sort out-of-core data on a distributed-memory cluster without making any assumptions about the input keys.

- We parallelize all disk I/O operations across all the disks in the cluster. That is, we have engineered our implementation to make sure that all reads and writes are evenly distributed across the nodes in the cluster. Furthermore, the reads and writes at a given node are evenly distributed across all the disks owned by that node.

- The output appears in the standard striped ordering used by the Parallel Disk Model (PDM) [VS94]; Appendix A.1 describes a variation of the PDM that we use. PDM ordering balances the load for any consecutive set of records across processors and disks as evenly as possible. A further advantage to producing sorted output in PDM ordering is that our algorithm can be used as a subroutine in other PDM algorithms. (See [Vit01] for a compendium of PDM algorithms.) To the best of our knowledge, our programs are the first multiprocessor sorting algorithms whose output is in striped PDM order.

1.4 Some applications of out-of-core sorting

We first describe some applications that handle out-of-core data, and then we discuss sorting as an important subroutine for several database operations.

An increasing number of computer and scientific applications are data-intensive; they need to efficiently manage large data volumes. Geographical information systems, seismic modeling, multimedia applications, and scientific computation are a few examples. Szalay et al. [SGV03] discuss the challenges of mining petabytes of data. The Sloan Digital Sky Survey [SKT$^+$00] and the Large Synoptic Survey Telescope [Tys02] are two examples of recent experiments that deal with astronomy datasets of sizes ranging from several terabytes up to a few petabytes. Web search engines need to store and search through enormous amounts of Web data.

Effective management of large volumes of data often necessitates the use of database management systems that provide good performance while manipulating large datasets. Graefe [Gra93] presents a thorough survey of algorithms for executing queries over large databases.

Apart from being a database operation in itself [Gra93, section 10.2], sorting is a key subroutine in several database operations; we mention a few of these operations here. Many query algorithms require *aggregation*, i.e., bringing together items that are alike in some defined way and then performing some operations on them. Efficient aggregation often requires sorting on a given set of attributes. Another database operation that directly uses sorting is *deduplication*, i.e., removal all but one from every set of data items that are similar. If the data items are sorted, duplicate removal requires a single scan of the data. Not surprisingly, most algorithms for duplicate removal use sorting [Gra93, Section 4.2]. *Relational-join* [VG84] is another widely used database operation. The main steps in computing a join of two relations based on a given set of attributes are sorting (based on the join attribute) the records of each relation, and then doing a parallel scan of the two sorted lists to collect all records that have similar values for the join attribute. Some other database operations that may benefit from a sort subroutine are *projection*, which is computing a subset of the records, and *bulk-loading*, which refers to the creation of an index structure, say a B-tree, for a large dataset.

The remainder of this thesis is organized as follows. Chapter 2 presents the original columnsort algorithm, along with our result showing that the divisibility restriction is unnecessary. In Chapter 3, we report our preliminary implementations of out-of-core columnsort, along with experimental results on Sun-A and Sun-B. Chapter 4 describes both subblock columnsort and M-columnsort, along with experimental results on Borg. Chapter 5 presents the slabpose columnsort algorithm and implementation details. Chapter 6 presents a simple technique for computing a lower bound on the execution time of a given run of our program. Chapter 7 analyzes recent results, from experiments on Jefferson. Chapter 8 discusses related work, focusing on the two dominant paradigms in the literature. Finally, Chapter 9 offers concluding remarks and some future work.

Chapter 2

Columnsort

This chapter presents Leighton's original columnsort algorithm: an oblivious algorithm that sorts correctly under certain restrictions. Along the way, we make observations that will improve our out-of-core implementations. Next, using the 0-1 Principle, we give a simple proof of its correctness. We also show that the divisibility restriction of columnsort is unnecessary.

2.1 The original columnsort algorithm

Columnsort sorts a given $r \times s$ mesh, subject to the following two restrictions:

1. s must be a divisor of r (the divisibility restriction), and
2. $r \geq 2s^2$ (the height restriction).[1]

Columnsort operates in eight steps to sort the mesh of all $N = rs$ values in column-major order. Each of steps 1, 3, 5, and 7 sorts each column. Each of steps

[1]Leighton's original paper had a slightly more relaxed height restriction: $r \geq 2(s-1)^2$. The $r \geq 2s^2$ restriction matches better with proof methods in this thesis.

$$\begin{bmatrix} 0 & 9 & 18 \\ 1 & 10 & 19 \\ 2 & 11 & 20 \\ 3 & 12 & 21 \\ 4 & 13 & 22 \\ 5 & 14 & 23 \\ 6 & 15 & 24 \\ 7 & 16 & 25 \\ 8 & 17 & 26 \end{bmatrix} \begin{array}{c} \xrightarrow{\text{Step 2}} \\ \xleftarrow{\text{Step 4}} \end{array} \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \\ 9 & 10 & 11 \\ 12 & 13 & 14 \\ 15 & 16 & 17 \\ 18 & 19 & 20 \\ 21 & 22 & 23 \\ 24 & 25 & 26 \end{bmatrix}$$

Figure 2.1: The operations of steps 2 and 4 of columnsort, shown for $r = 9$ and $s = 3$.

$$\begin{bmatrix} 0 & 9 & 18 \\ 1 & 10 & 19 \\ 2 & 11 & 20 \\ 3 & 12 & 21 \\ 4 & 13 & 22 \\ 5 & 14 & 23 \\ 6 & 15 & 24 \\ 7 & 16 & 25 \\ 8 & 17 & 26 \end{bmatrix} \begin{array}{c} \xrightarrow{\text{Step 6}} \\ \xleftarrow{\text{Step 8}} \end{array} \begin{bmatrix} -\infty & 5 & 14 & 23 \\ -\infty & 6 & 15 & 24 \\ -\infty & 7 & 16 & 25 \\ -\infty & 8 & 17 & 26 \\ 0 & 9 & 18 & \infty \\ 1 & 10 & 19 & \infty \\ 2 & 11 & 20 & \infty \\ 3 & 12 & 21 & \infty \\ 4 & 13 & 22 & \infty \end{bmatrix}$$

Figure 2.2: The operations of steps 6 and 8 of columnsort.

2, 4, 6, and 8 performs a specific fixed permutation.

- As Figure 2.1 shows, treating the mesh as an $r \times s$ mesh, step 2 transposes the mesh and then reshapes the resulting $s \times r$ mesh back into an $r \times s$ mesh by taking each row of r entries and mapping it to a consecutive set of r/s rows. We refer to this permutation as the *transpose-and-reshape* permutation.

- Step 4 performs the inverse of the permutation performed in step 2: treat each consecutive set of r/s rows as a single row in an $s \times r$ mesh, and transpose this mesh into the $r \times s$ mesh. We refer to this permutation as the *reshape-and-transpose* permutation.

- As Figure 2.2 shows, step 6 shifts each column down by $\lfloor r/2 \rfloor$ positions. That is, the bottom $\lfloor r/2 \rfloor$ elements of each column move to the top of the next column to the right, and the top $\lceil r/2 \rceil$ elements of each column move to the bottom of the column. The evacuated top $\lfloor r/2 \rfloor$ entries of the leftmost column (column 0) are filled with the value $-\infty$. A new rightmost column (column s) is created, receiving the bottom $\lfloor r/2 \rfloor$ elements of column $s-1$, and the bottom $\lceil r/2 \rceil$ entries of this new column are filled with the value ∞.
- Step 8 performs the inverse of the permutation performed in step 6: shift each column up by $\lceil r/2 \rceil$ positions.

2.2 Simplifying observations

Three observations will improve our out-of-core implementation later on:

Placement: Immediately following each of steps 2, 4, and 6, we sort each column. Therefore, we need to permute each element into only the correct column in steps 2, 4, and 6. Its placement within the column (i.e., its exact row) is immaterial.

Run: The sorts in steps 3, 5, and 7 can be implemented as multiway merge operations. That is, the entries to be sorted are arranged as several sorted runs, which need only be merged. In Leighton's basic algorithm, sorting each column in step 3 is a merge of s runs of length r/s each, and sorting each column in step 7 is a merge of 2 runs of lengths $\lfloor r/2 \rfloor$ and $\lceil r/2 \rceil$. Our implementation takes advantage in steps 3 and 7 of the formation of these runs in the previous steps, and it also uses the placement observation in step 4 to form s sorted runs of length r/s each so that step 5 is also a multiway merge.

Pairing: We can combine steps 6–8 by pairing adjacent columns. We sort the bottom $\lfloor r/2 \rfloor$ entries of each column along with the top $\lceil r/2 \rceil$ entries of the next column, placing the sorted r entries into the same positions. The top $\lceil r/2 \rceil$ entries of the leftmost column were already sorted by step 5 and can therefore be left alone, and similarly for the bottom $\lfloor r/2 \rfloor$ entries of the rightmost column.

These observations are known within the mesh-sorting community. Rajasekaran [Raj01] uses the run and pairing observations at various points in his LMM-sort algorithm.

2.3 Proof of correctness

To show that columnsort really does sort, we first need to establish that it is oblivious. Since the even-numbered steps perform fixed permutations, they are oblivious to the data. The sorting method used in the odd-numbered steps might not be oblivious, however. As pointed out by Leighton [Lei92, p. 147], "No matter how the columns are sorted, the end result will look the same." Thus, we can imagine that an oblivious sorting method was used for the odd-numbered steps, knowing that we can substitute any sorting method of our choosing. Hence, the entire algorithm is oblivious.

Because columnsort is oblivious, we can prove its correctness by means of the *0-1 Principle* [Lei92, pp. 141–142]:

If an oblivious algorithm sorts all input sets consisting solely of 0s and 1s, then it sorts all input sets with arbitrary values.

When given a 0-1 input, we say that an area of the mesh is *clean* if it consists either of all 0s or all 1s. An area that might have both 0s and 1s is *dirty*. We shall show that steps 1–4 reduce the size of the dirty area to at most half a column and that steps 5–8 complete the sorting, assuming that the dirty area is at most half a column in size. As we read 0-1 values in a prescribed order within the mesh, a *0 $\to$ 1 transition* occurs when a 0 is followed immediately by a 1; we define a *1 $\to$ 0 transition* analogously.

Lemma 2.1 *Assuming a 0-1 input, after step 3, the mesh consists of some clean rows of 0s at the top, some clean rows of 1s at the bottom, and at most s dirty rows between them.*

Proof: As Figure 2.3(a) shows, after step 1, reading from top to bottom, each column consists of 0s followed by 1s. As we read a given column from top to bottom after step 1, there is at most one transition, and it is a 0 $\to$ 1 transition.

Step 2 turns each column into exactly r/s rows, shown in Figure 2.3(b). By the divisibility restriction, r/s is an integer. If we read the mesh in row-major order, any 1 $\to$ 0 transition occurs at the end of one row and the beginning of another. Therefore, read in row-major order, each set of consecutive r/s rows has at most the one 0 $\to$ 1 transition from the corresponding column after step 1. Within each set of consecutive r/s rows, the only row that may be dirty is the row containing the 0 $\to$ 1 transition. Since there are s such sets of rows, the mesh now has at most s dirty rows altogether.

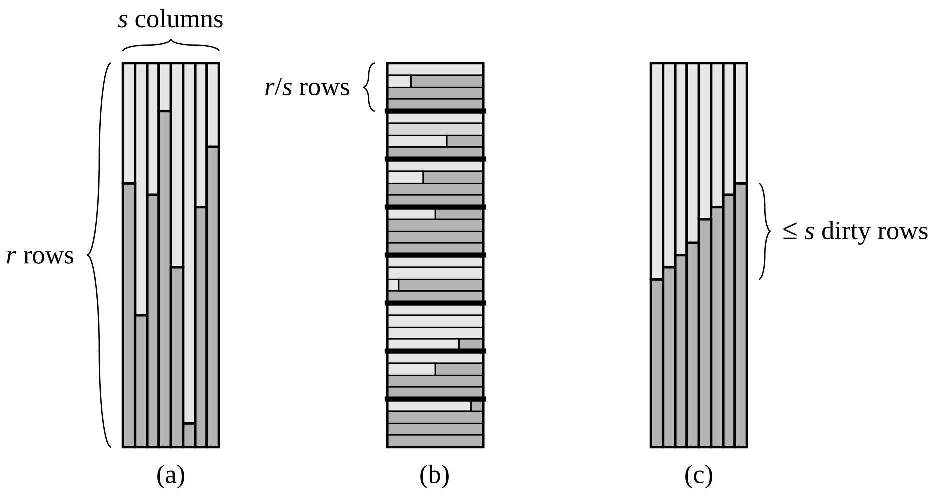

Figure 2.3: Steps 1–3 of columnsort, assuming a 0-1 input. 0s are lightly shaded and 1s are more darkly shaded. **(a)** After step 1, each column consists of 0s followed by 1s. **(b)** After step 2, each set of consecutive r/s rows has at most one $0 \to 1$ transition. There are at most s $0 \to 1$ transitions altogether. Heavy lines separate the rows formed from each column. **(c)** After step 3, the clean rows of 0s are at the top, the clean rows of 1s are at the bottom, and there are at most s dirty rows between them.

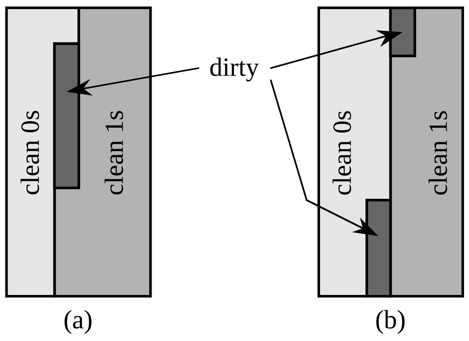

Figure 2.4: After step 4 the dirty area either **(a)** fi ts in a single column or **(b)** crosses from one column to the next.

Figure 2.3(c) shows the effect of step 3: moving the clean rows of 0s to the top rows of the mesh and the clean rows of 1s to the bottom rows of the mesh. The dirty rows, of which there are at most s, end up between the clean 0s and 1s. ■

Lemma 2.2 *Assuming a 0-1 input, after step 4, the mesh consists of some clean columns of 0s on the left, some clean columns of 1s on the right, and a dirty area of size at most s^2 between them.*

Proof: By Lemma 2.1, the dirty area after step 3 is a set of consecutive rows numbering at most s. Since each row is s columns wide, the dirty area has size at most s^2. Step 4 permutes the clean rows of 0s at the top of the mesh to be clean columns of 0s on the left, the clean rows of 1s at the bottom of the mesh to be clean columns of 1s on the right, and the dirty area has at most s^2 entries between the clean areas. ■

Corollary 2.3 *Assuming a 0-1 input, reading the mesh in column-major order after step 4, the dirty area is at most half a column in size.*

Proof: The height restriction $r \geq 2s^2$ is equivalent to $s^2 \leq r/2$. The dirty area has size at most s^2, and $r/2$ is the height of half a column. ■

Lemma 2.4 *Assuming a 0-1 input, if the mesh, read in column-major after step 4, has a dirty area that is at most half a column in size, then steps 5–8 produce a fully sorted output.*

Proof: As Figure 2.4 shows, because the dirty area is at most half a column in size, it either fits in a single column or crosses from one column into the next. If the dirty area fits in a single column, then the sorting of step 5 cleans it, and steps 6–8 do not corrupt the sorted 0-1 output. If the dirty area is in two columns after step 4, then it is in the bottom half of one column and the top half of the next. Step 5 does not change this property. Since the size of the dirty area (s^2) is an integer, $s^2 \leq r/2$ implies $s^2 \leq \lfloor r/2 \rfloor$. Therefore, step 6, which shifts each column down by $\lfloor r/2 \rfloor$ positions, ensures that the dirty area resides in one column. Step 7 then cleans the dirty area, and step 8 moves all values back to where they belong. ■

Theorem 2.5 *Columnsort sorts any input correctly.*

Proof: Immediate from columnsort being oblivious, the 0-1 Principle, Corollary 2.3, and Lemma 2.4. ■

2.4 Removing the divisibility restriction

In this section, we show that columnsort, as described in Section 2.1, sorts correctly in the absence of the divisibility restriction. We continue to assume that $r \geq 2s^2$.

As in Section 2.1, we rely on the 0-1 Principle. Because s might not be a divisor of r, however, we shall account for $1 \to 0$ transitions. The statement of the key lemma is similar to that of Lemma 2.1.

Lemma 2.6 *Assuming a 0-1 input but without the divisibility restriction, after step 3, the mesh consists of some clean rows of 0s at the top, some clean rows of 1s at the bottom, and at most s dirty rows between them.*

Proof: As in the proof of Lemma 2.1, each column has at most one transition after step 1, and it is a $0 \to 1$ transition. Because s might not divide r, however, we can see $1 \to 0$ transitions within the rows after step 2. Figure 2.5 shows this

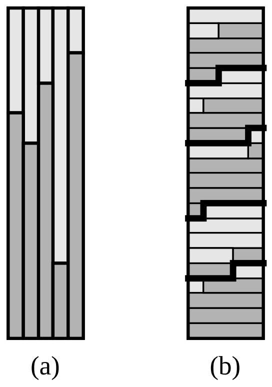

Figure 2.5: Steps 1 and 2 when s does not divide r. **(a)** After step 1. **(b)** Within the rows after step 2, there is at most one $0 \to 1$ transition for each column after step 1, and there is at most one $1 \to 0$ transition for each pair of consecutive columns. Heavy lines separate the values originally from each column.

possibility. There are still at most s $0 \to 1$ transitions, and because a $1 \to 0$ transition can occur only between pairs of consecutive columns after step 1, the number of $1 \to 0$ transitions is at most $s - 1$. Thus, as we read the mesh in row-major order after step 2, there are at most $2s - 1$ transitions of either type.

After step 2, let X be the set of dirty rows containing one $0 \to 1$ transition and no $1 \to 0$ transitions, Y be the set of dirty rows containing one $1 \to 0$ transition and no $0 \to 1$ transitions, and Z be the set of all other dirty rows (each of which must contain at least one $0 \to 1$ transition and at least one $1 \to 0$ transition).

We claim that $\max(|X|, |Y|) + |Z| \leq s$. To prove this claim, note that every row of X and Z contains at least one $0 \to 1$ transition, and so $|X| + |Z| \leq s$. Similarly, every row of Y and Z contains at least one $1 \to 0$ transition, and so $|Y| + |Z| \leq s - 1$. If $\max(|X|, |Y|) = |X|$ then $\max(|X|, |Y|) + |Z| \leq s$, and if $\max(|X|, |Y|) = |Y|$ then $\max(|X|, |Y|) + |Z| \leq s - 1$. In either case, $\max(|X|, |Y|) + |Z| \leq s$, thus proving the claim.

Now we examine the effect of step 3. As in the proof of Lemma 2.1, the clean rows of 0s move to the top and the clean rows of 1s move to the bottom. Let us consider pairs of rows in which one row of the pair is in X and the other row is in Y; there are exactly $\min(|X|, |Y|)$ such pairs of rows. When we pair them up and move rows of all 0s to the top and rows of all 1s to the bottom, we have one of the following results:

$$\begin{array}{lccccc}
 & \text{more 0s than 1s} & & \text{more 1s than 0s} & & \text{equal 0s and 1s} \\
\text{from } X: & 0\cdots0001\cdots1 & & 0\cdots0111\cdots1 & & 0\cdots0011\cdots1 \\
\text{from } Y: & 1\cdots1000\cdots0 & & 1\cdots1110\cdots0 & & 1\cdots1100\cdots0 \\
 & \Downarrow & \text{or} & \Downarrow & \text{or} & \Downarrow \\
 & 0\cdots0000\cdots0 & & 0\cdots0110\cdots0 & & 0\cdots0000\cdots0 \\
 & 1\cdots1001\cdots1 & & 1\cdots1111\cdots1 & & 1\cdots1111\cdots1 \\
 & \text{clean row of 0s} & & \text{clean row of 1s} & & \text{clean rows}
\end{array}$$

In any case, at least one clean row is formed, and so at least $\min(|X|, |Y|)$ new clean rows are created. These clean rows go to the top (if 0s) and bottom (if 1s) of the mesh.

The number of dirty rows remaining is at most $|X|+|Y|-\min(|X|, |Y|)+|Z|$. Observing that $|X|+|Y|-\min(|X|, |Y|) = \max(|X|, |Y|)$, we see that the number of dirty rows remaining is at most $\max(|X|, |Y|)+|Z|$, which, by our earlier claim, is at most s. ■

From here, we can use the same technique as in Section 2.1 to prove the following theorem:

Theorem 2.7 *Even without the divisibility restriction, columnsort sorts any input correctly.* ■

Chapter 3

Background: Preliminary Implementations

In this chapter, we present our preliminary implementations. Starting with a description of our out-of-core setting of a cluster, we summarize non-threaded 4-pass columnsort. Then we discuss the motivation for using threads, leading up to the next two preliminary implementations: 4-pass columnsort and 3-pass columnsort. We present results of our experiments on Sun-A and Sun-B and conclude by recalling the problem-size restriction.

3.1 Out-of-core setting

In this section, we describe our adaptation of Leighton's original columnsort algorithm to an out-of-core setting of a distributed-memory cluster. We first present our machine model. Next, we describe the common structure of each pass, followed by a discussion of the implementation of each pass within the common framework.

Machine model

We assume that our machine has P processors $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{P-1}$ and D disks $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{D-1}$. The processors may be on separate nodes of a cluster, they may be processors within a shared-memory node, or they may be a combination of the above. For example, Sun-B has a cluster of 4 nodes, with each node having 2 processors, for a total of 8 processors; in this configuration, each node has 1 disk, which is shared by the 2 processors on that node. More generally, we assume that each processor can access at least one disk directly.

When $D = P$, each processor accesses exactly one disk. When $D < P$, we require that there be P/D processors per node and that they share the node's disk; in this case, each processor accesses a distinct portion of the disk. In our implementation, we treat this distinct portion as a separate "virtual disk," allowing us to assume that $D \geq P$. When $D > P$, each processor has exclusive access to D/P disks, and it stripes its data across these disks in *local stripes* of BD/P records each. We say that a processor *owns* the D/P disks that it accesses.

The *striping unit*, also known in the PDM as the *block size*, is the maximum number of consecutive records that are stored on a single disk. For good performance, the striping unit should be at least one physical block on the disk. In practice, the striping unit is at least the system's physical page size. We parameterize the striping unit by the PDM parameter B. Any disk access transfers an entire block of records between the disk and the memory of some processor.

The memory of the entire system holds M records. Memory is partitioned among the P processors so that each processor can hold M/P records. In practice, a processor that holds M/P records has a physical memory that is larger than M/P

records' worth. That is because physical memory holds more than just records—software and the run-time stack, for example. Moreover, our implementations require multiple data buffers for merging (which is not in-place), communication, and overlapping I/O with other operations.

Given the machine parameters, the preliminary implementations set the columnsort mesh dimensions r and s as follows. We require a column's worth of data to fit within one processor's memory. Thus, we set $r = M/P$. Since $N = rs$, we set $s = N/r = NP/M$. In all our implementations, we assume the column size, r, to be even.

Common structure among passes

Even though the number of passes in our preliminary implementations varies, there is a structure common to all the passes. For ease of understanding, we shall describe the pass structure under the assumption that our program proceeds in four passes over the data.

Each pass reads records from one portion of each disk and writes records to a different portion of each disk. It is best to think of these portions as files. Each processor has an input file, with part of the total data to be sorted.[1] We cannot overwrite the input file, since we test every run of our program for correctness. Moreover, some applications might require the input file to stay intact. Apart from the input file, therefore, we use two files: file A and file B. Pass 1 writes to file A, pass 2 reads from file A and writes to file B, pass 3 reads from file B and writes to file A, and so on. The sorted output resides in file A or file B, depending on whether the number of passes is odd or even, respectively.

[1] In a cluster setting, each processor has an input fi le with N/P records.

C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7
0	16	32	48	64	80	96	112
1	17	33	49	65	81	97	113
2	18	34	50	66	82	98	114
3	19	35	51	67	83	99	115
4	20	36	52	68	84	100	116
5	21	37	53	69	85	101	117
6	22	38	54	70	86	102	118
7	23	39	55	71	87	103	119
8	24	40	56	72	88	104	120
9	25	41	57	73	89	105	121
10	26	42	58	74	90	106	122
11	27	43	59	75	91	107	123
12	28	44	60	76	92	108	124
13	29	45	61	77	93	109	125
14	30	46	62	78	94	110	126
15	31	47	63	79	95	111	127
$\mathcal{D}_0$	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_0$	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$

Figure 3.1: The layout used internally within out-of-core columnsort. Each square represents a block of B records. Each column is stored contiguously on the set of disks owned by a single processor. Column C_i is stored on the disks owned by processor $\mathcal{P}_{i \bmod P}$. In this example, $P = D = 4$, $B = 2$, and $s = 8$.

Each pass performs two consecutive steps of columnsort. That is, pass 1 performs columnsort steps 1 and 2, pass 2 performs steps 3 and 4, pass 3 performs steps 5 and 6, and pass 4 performs steps 7 and 8.

Figure 3.1 shows how the data is organized on the disks at the start of each pass. The data is placed so that each column is stored in contiguous locations on the disks owned by a single processor. Specifically, processor j *owns* columns j, $j + P$, $j + 2P$, and so on.

Each pass is decomposed into s/P *rounds*. Each round processes the next set of P consecutive columns, one column per processor, in five phases:

Read phase: Each processor reads a column of r records from the disks that it owns.

Sort phase: Each processor locally sorts, in memory, the r records it has just read. Next, if required, each processor rearranges the sorted data into the correct order for the next phase.

Communicate phase: Each record is destined for a specific column, depending on which even-numbered columnsort step this pass is performing. In order to get each record to the processor that owns this destination column, processors exchange records.

Permute phase: Having received records from other processors, each processor rearranges them into the correct order for writing.

Write phase: Each processor writes a set of records onto the disks that it owns. These records are not necessarily all written consecutively onto the disks, though they are written as a small number of sorted runs.

We shall see shortly how each phase differs in each pass.

How each pass works

The exact operation of three of the five phases within a round differs from pass to pass. The phases that remain roughly the same are reading and permuting. Each processor in each round, regardless of the pass, reads a column of r records by simply reading r/B consecutive blocks from the disks that it owns. This style of reading yields the best possible I/O performance. Each processor in each round permutes the final data into the writing pattern for that pass.

Now we examine how the four passes perform each of the other three phases.

Pass 1. This pass reads in each column, locally sorts it—assuming nothing about the sortedness of the data within the column—and then writes it out according to the transpose-and-reshape permutation.

Local sorting: Because we make no assumptions about the sortedness of each column or about the key distribution, we locally sort using a combination of the system `qsort` call and merge sort. We create sorted runs of a certain size k using `qsort` and then we merge these runs using merge sort. Assuming that `qsort` performs $O(k \lg k)$ comparisons on average to sort k elements, the in-core sorting complexity of each round of pass 1 is $O(r \lg r)$. The per-processor sorting complexity of pass 1, therefore, is $O((s/P)r \lg r) = O((N/P) \lg r)$.

Partial-sorting optimization: Here we briefly discuss what we refer to as the *partial-sorting optimization*. This optimization affects only the local sorting phase of the first pass of our implementation. We observe that often, the value M/P is larger than the minimum required value of the columnsort parameter r, in order for r and s to satisfy the height restriction. Let r' be the minimum column size, such that the height restriction is satisfied. In the first pass, therefore, if we sort each chunk of size r', instead of sorting the whole column of r records, the algorithm still works. This optimization reduces the overall sorting time in the first pass. On Jefferson, for a problem size of 16 GB, we found that the running time reduced by approximately 10%. For larger problem sizes, however, the improvement is less pronounced; the value of r' gets closer to r for larger problem sizes.

Communication: After locally sorting, each processor has a sorted column of r records. As in step 2 of columnsort, r/s records out of the r are destined for each of the s columns. Each processor locally permutes its r records to form s sorted runs of r/s records each. Every processor owns s/P columns, and so every processor sends s/P sorted runs to each of the other processors. Each processor, therefore, receives P messages (including a message that the processor sends to itself), with each message consisting of s/P sorted runs.

Writing: Every processor receives, for each of its s/P columns, a sorted run from each of the P processors. Therefore, every processor now has P sorted runs for each of the s/P columns that it owns. Since each run consists of r/s records, each processor performs s/P writes of the P runs, i.e., Pr/s records. In the case in which $D > P$ and $Pr/s < BD/P$—so that the amount of data being written to each column is less than the size of a local stripe—the writes are not distributed evenly across the D/P disks owned by the processor. For example, Figure 3.2 shows a situation with $s = 8$, $P = 4$, $D = 16$ (so that $D/P = 4$), and $r = 4B$. Here, $D > P$ and $Pr/s = 2B < 4B = BD/P$. The figure shows the write patterns in the first round for processor $\mathcal{P}_0$. This processor owns columns 0 and 4. It needs to write $2B$ records, or 2 blocks, to each of these columns. As Figure 3.2(a) shows, if both columns start on disk $\mathcal{D}_0$, we write to only half the disks. Figure 3.2(b) demonstrates how we use the placement observation to write to different locations within the columns in order to distribute the load evenly among the disks.

Pass 2. This pass reads in each column, locally sorts it—knowing that every column consists of s sorted runs of r/s records each—and then writes it out according to the reshape-and-transpose permutation.

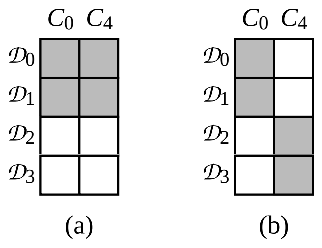

Figure 3.2: Write patterns in the first round of pass 1 for processor $\mathcal{P}_0$ when the amount of data being written to each column is less than the size of a local stripe. Shaded blocks are the ones written to if we **(a)** do not use and **(b)** do use the placement observation to distribute the load evenly.

Local sorting: We use merge sort, but because we have s sorted runs to start, we need to recurse only $\lg s$ levels rather than $\lg r$. It takes $O(r \lg s)$ time to locally sort each column in a given round, so that the total local sorting time is $O((s/P)r \lg s) = O((N/P) \lg s)$ per processor.

Communication: Communication in pass 2 is the same as in pass 1, except that immediately after local sorting, the records destined for each column are already formed into a sorted run of r/s records. Therefore, the local permutation prior to sending is skipped. As in pass 1, however, records are locally permuted after they have been received.

Writing: Pass 2 writes in exactly the same way as pass 1. It can do so even though the runs received within a given round are not destined for consecutive locations within a column according to the reshape-and-transpose permutation. These runs arrived in contiguous locations of a communication buffer, and we can write them together because, by the placement observation, it does not matter where in the column we write them. Moreover, writing these runs as we receive them makes the input to the next pass consist of sorted runs.

Pass 3. Like pass 2, this pass reads in each column and locally sorts it, knowing that every column consists of s sorted runs of r/s records each. It then writes out the records according to the shift-down-by-$r/2$ permutation.

Local sorting: We use merge sort, just as in pass 2.

Communication: The last $r/2$ records in each column are going to go to the next column. Therefore, in each round, processor $\mathcal{P}_j$ sends its last $r/2$ records to processor $\mathcal{P}_{(j+1) \bmod P}$.

Writing: In each round, each processor writes a column of r records by simply writing r/B consecutive blocks to the disks that it owns. Just like how we read, this style of writing yields the best possible I/O performance.

Pass 4. This last pass reads in each column and locally sorts it—knowing that every column consists of 2 sorted runs of $r/2$ records each—and then writes it out according to the shift-up-by-$r/2$ permutation, but in PDM order.

Local sorting: We merge the 2 sorted runs, which takes only $O(r)$ time per processor in each round. Over all rounds, local sorting takes $O(N)$ time, or $O(N/P)$ time per processor.

Communication: In each round, each processor has r records that need to go to r consecutive locations in PDM order. That is, they are striped across all D disks in the system. In PDM order, each stripe has BD records, and so the r records form r/BD consecutive stripes. In other words, the r records will go to r/B blocks in all, or r/BD blocks per disk. Since every processor owns D/P disks, $(r/BD)(D/P) = r/BP$ blocks go to each processor. Every processor sends r/P records to each processor. Each processor, therefore, receives P messages (including the message that the processor sends to itself), each message consisting of r/BP blocks. As each processor receives these blocks, it assembles them into a buffer such that the blocks that are destined for the same disk are contiguous.

Writing: Each processor has D/P disks, and the records destined for each disk are already contiguous. A set of D/P write calls finishes the job.

3.2 Non-threaded 4-pass columnsort

Non-threaded 4-pass columnsort uses asynchronous MPI and MPI-2 calls to overlap local sorting, communication, and I/O. At any particular time, processor j might be communicating records belonging to column $j + kP$, locally sorting records in column $j+(k+1)P$, reading column $j+(k+2)P$, and writing column $j+(k-1)P$. Non-threaded 4-pass columnsort does not overlap reading with either local sorting or writing, however. For implementation details, see [CCW01]. We conclude with two main features of this first implementation. These two features carry over to all our subsequent implementations:

- Our algorithm's I/O and communication patterns are oblivious to the keys; the exact sequence of I/O and communication operations depends only on the problem size and the system parameters, not on the key values. Because the I/O and communication patterns are completely oblivious to the keys, each round has no effect at all on other rounds of the same pass. That is, the rounds of a pass are fully independent of each other and therefore can execute with full asynchrony. We shall take advantage of this property in our threaded implementations.

- Our algorithm produces sorted records in the standard striped ordering used by the Parallel Disk Model (PDM). This ordering affects only the write phase of the last pass, in which each column is striped across all D disks rather than residing solely on the disks of the processor that owns it. As mentioned before, PDM ordering balances the load for any consecutive set of records across processors and disks as evenly as possible.

In the rest of the thesis, unless we use “non-threaded” in an implementation’s name, it is threaded.

3.3 4-pass columnsort: The threaded 4-pass implementation

We start this section by discussing the motivation behind implementing 4-pass columnsort, which is just a threaded version of non-threaded 4-pass columnsort. We conclude with a description of the thread structure of 4-pass columnsort.

Motivation

There are two reasons to revise our implementation to use threads. First, threads allow scheduling to be more dynamic. Second, our threaded implementation permits greater flexibility in memory usage.

Our programs use four types of machine resources: disks, the communication network, processing power, and memory. As described in the previous section, each column goes through five phases, two using I/O (the read and write phases), one using the network (the communicate phase), and the other two using the processing power (the sort and permute phases). All phases make use of memory, of course.

Non-threaded 4-pass columnsort has a single thread of control. In order to permit overlapping of I/O, communication, and processing, it relies on asynchronous MPI and MPI-2 calls in order to schedule the usage of the four types of resources. Due to the single thread of control, this implementation has to decide the overlap mechanism statically at coding time and is not fully flexible. For example, the implementation is unable to adapt automatically to a faster network, faster CPU, or faster I/O. The 4-pass columnsort implementation uses the standard pthreads package to overlap I/O, communication, and processing. It uses only synchronous MPI and MPI-2 calls.

Because non-threaded 4-pass columnsort uses static scheduling, its memory usage is static and has to be known at coding time. Consequently, this implementation is unable to adapt to an increased amount of available memory. Our (threaded) 4-pass columnsort, on the other hand, maintains a global pool of memory buffers, the number of which is set at the start of each run of the program. One can determine the optimum number of buffers for a given configuration by means of a small number of experimental runs.

Basic thread structure

In order to overlap the usage of resources in a dynamic manner, we created four threads per processor. One thread is responsible for all the disk I/O functions, one does all the interprocessor communication, one does all the sorting, and the final thread does all the permuting. We shall refer to the four threads as the I/O, communicate, sort, and permute threads, respectively. The threads operate on buffers, each capable of holding exactly r records, and which are drawn from a global pool. The threads communicate with each other via a standard semaphore mechanism. The read and write functions appear in a common thread—the I/O thread—because they will serialize at the disk anyway.[2]

Figure 3.3 illustrates how the threads work by displaying the history of a column within a given round of a given pass.

1. The I/O thread acquires a buffer b from the global pool and performs the read phase of column c by reading the column from the disk into this buffer. The I/O thread suspends while the read happens, and when the read completes, the I/O thread wakes up.

2. When the I/O thread wakes up after the read completes, it signals the sort thread, which picks up buffer b and performs the sort phase of column c on it.

3. The sort thread signals the communicate thread, which picks up buffer b and performs the communicate phase of column c on it. The communicate

[2]We found some evidence [Ham03] suggesting that it is not necessarily the best idea to use a single thread for reads and writes. Our programs now allow both choices.

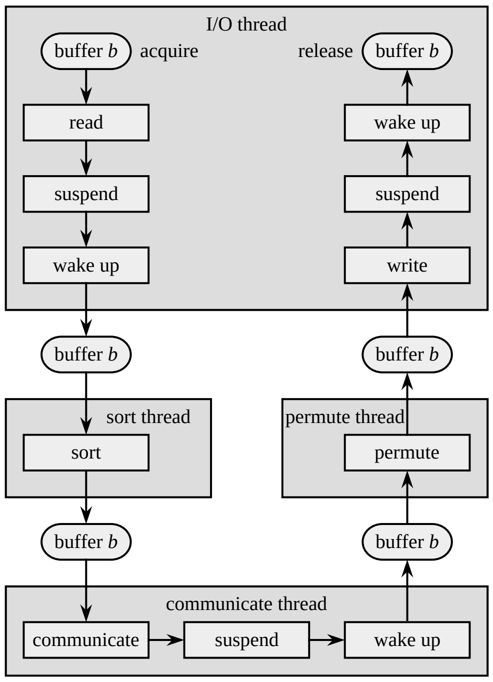

Figure 3.3: The history of a buffer *b* as it progresses within a given round of a given pass. The I/O thread acquires the buffer from the global pool and then reads into it from disk. The I/O thread suspends during the read, and when it wakes up, it signals the sort thread. The sort thread sorts buffer *b* and signals the communicate thread. The communicate thread suspends during interprocessor communication, and when it wakes up, it signals the permute thread. The permute thread then permutes buffer *b* and signals the I/O thread. The I/O thread writes the buffer to disk, suspending during the write. When the I/O thread wakes up, it releases buffer *b* back to the global pool.

phase suspends during interprocessor communication, and after communication completes, it wakes up.

4. The communicate thread signals the permute thread, which picks up buffer *b* and performs the permute phase of column *c* on it.

5. Finally, after the permute phase completes, the permute thread signals the I/O thread, which picks up buffer *b* and writes it out to disk. The I/O thread suspends during the write, and when the write completes, the I/O thread wakes up and releases buffer *b* back to the global pool.

The pthreads implementation may preempt any thread at any time. Thus, during the time that a given thread considers itself as active, it might not actually be running on the CPU.

The sort, permute, and communicate threads allocate additional buffers for their own use. Each of these threads allocates one buffer at the beginning of the program and uses it throughout the entire run. Thus, the total memory usage of the 4-pass columnsort is three buffers more than are created in the global pool.

3.4 3-pass columnsort: The threaded 3-pass implementation

This section describes how to reduce the number of passes in 4-pass columnsort, resulting in 3-pass columnsort. The key observation is the *pairing observation* from Section 2.1.

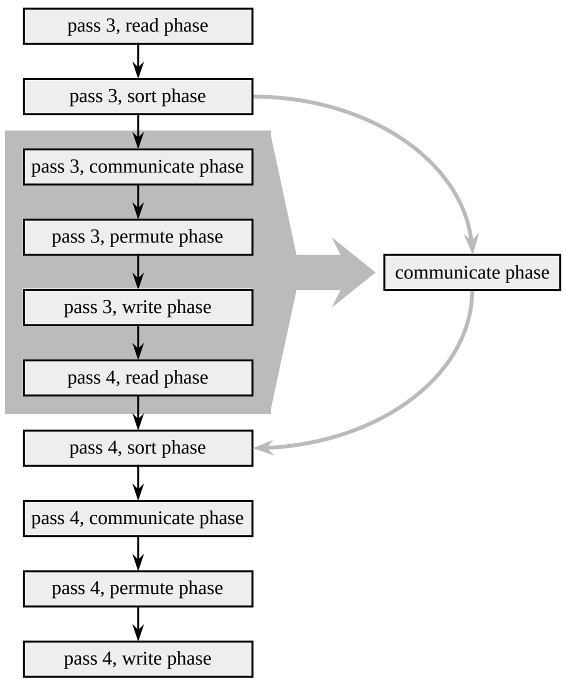

Figure 3.4: How passes 3 and 4 of 4-pass columnsort are combined into a single pass of 3-pass columnsort. The communicate, permute, and write phases of pass 3, and the read phase of pass 4 are replaced by a single communicate phase.

Basic idea

To take advantage of the pairing implementation, we combine steps 5–8 of columnsort—passes 3 and 4 in 4-pass columnsort—into one pass. Figure 3.4 shows how. In 4-pass columnsort, the communicate, permute, and write phases of pass 3, along with the read phase of pass 4, merely shift each column down by $r/2$ rows (wrapping the bottom half of each column into the top half of the next column). We replace these four phases by a single communicate phase.

In 3-pass columnsort, the first two passes are the same as in 4-pass columnsort. To further understand how the last pass works, let us examine a typical round (one that is neither the first nor the last round) in this pass. At the start of the round, some processor $\mathcal{P}_k$ contains in a buffer $r/2$ records left over from the previous round. When the round completes, some other processor $\mathcal{P}_l$, where $l = (k-1) \bmod P$, will contain $r/2$ leftover records, and it will serve as $\mathcal{P}_k$ in the next round. The round proceeds as follows:

1. Each processor except for $\mathcal{P}_k$ reads in a column, so that $P-1$ columns are read in.

2. Each processor except for $\mathcal{P}_k$ sorts its column locally.

3. With the exception of $\mathcal{P}_k$, each processor $\mathcal{P}_i$ sends the first $r/2$ of its sorted elements to processor $\mathcal{P}_{(i-1) \bmod P}$. After all sends are complete, each processor except for $\mathcal{P}_l$ holds r records ($r/2$ that it had prior to the send, and $r/2$ that it just received), and processor $\mathcal{P}_l$ holds $r/2$ records, which it keeps aside in a separate buffer to be used in the next round. This communicate phase replaces the four phases shaded in Figure 3.4.

4. Each processor except for $\mathcal{P}_l$ sorts its column locally.

5. To prepare for the write in PDM order, each processor (except $\mathcal{P}_l$) sends r/P records to every processor (including $\mathcal{P}_l$ and itself).

6. Each processor locally permutes the $r(P-1)/P$ records it has received to put them into the correct PDM order.

7. Each processor writes the $r(P-1)/P$ records to the disks that it owns.

The first and last rounds have minor differences from the middle rounds. The first round processes all P columns, and the last round may process fewer than

$P - 1$ columns. Because the first round processes P columns, the number of rounds is $1 + \lceil (s - P)/(P - 1) \rceil$.

Thread structure

In 3-pass columnsort, the first two passes have the same thread structure as in 4-pass columnsort, but the third pass is different. As we have just seen, the third pass has seven phases. As before, each phase is assigned to a single thread, except that the read and write phases are assigned to a single I/O thread. Consequently, there is one I/O thread, two communicate threads, two sort threads, and one permute thread. This thread structure raises two additional issues.

First, the number of additional buffers increases. Other than the I/O thread, each thread requires an r-record buffer. Since there are five non-I/O threads, this implementation requires five buffers more than are in the global pool. That is, 3-pass columnsort requires two buffers more than 4-pass columnsort.

Second, because there are two communicate threads, each making calls to MPI, not all MPI implementations are suitable. Some MPI implementations are unreliable when multiple threads perform communication.

3.5 Performance results

Table 3.1 summarizes performance results of our preliminary implementations, from experiments done on Sun-A and Sun-B platforms. Chapter 7 discusses more recent results on Jefferson, a Beowulf cluster. The results are for 64-byte[3] records

[3]In all our experiments, the buffer size is a power of 2. To ensure an integral number of records, we restrict ourselves to record sizes that are powers of 2 as well. The Datamation-standard for sorting uses 100-byte records; we use 64 byte records, implying that for a given volume of data, our measurements are for over 50% more records.

with an integer key at the beginning of each record. The input files are generated using the `drand` function to generate the value of each key.

Sun-A

Sun-A is a cluster of 4 Sun Enterprise 450 SMPs. We used only one processor on each node. Each processor is an UltraSPARC-II, running at 296 MHz and with 128 MB of RAM, of which we used 40 MB for holding data. The nodes are connected by an ATM OC-3 network, with a peak speed of 155 megabits per second. Each node has one disk, each an IBM DNES309170 spinning at 7200 RPM and with an average latency of 4.17 msec. All disks are on Ultra2 SCSI buses. The MPI and MPI-2 implementations are part of Sun HPC ClusterTools 4.

Sun-B

Sun-B is also a cluster of 4 Sun Enterprise 450 SMPs. Here, we used two processors on each node. Other than the disks, the components are the same as in Sun-A. This system has 8 disks, 4 of which are the same IBM disks as in Sun-A, and the other 4 of which are Seagate ST32171W also spinning at 7200 RPM and also with an average latency of 4.17 msec.

4-pass implementations: Threaded vs. non-threaded

From the columns labeled “4-pass threaded” in Table 3.1, we see that (threaded) 4-pass columnsort took only 49.8% of the time taken by the non-threaded 4-pass columnsort on Sun-A. On Sun-B, 4-pass columnsort took 57.1% as much time as the non-threaded 4-pass columnsort.

	Sun-A 4-pass non-threaded	Sun-A 4-pass threaded	Sun-A 3-pass threaded	Sun-B 4-pass non-threaded	Sun-B 4-pass threaded	Sun-B 3-pass threaded
P	4	4	4	8	8	8
D	4	4	4	8	8	8
U (MB)	160	192	256	320	384	512
V (GB)	1	1	1	2	2	2
T (minutes)	4.338	2.161	2.043	5.631	3.217	2.944

Table 3.1: Parameters and results for three implementations of our out-of-core columnsort algorithm on two clusters of Sun Enterprise 450 SMPs. The rows show number of processors (P), number of disks (D), megabytes of memory used across the entire system (U), gigabytes sorted (V), and time to sort (T) for each system. Times shown are averages of five runs.

What accounts for this significant improvement in running time? Due to the highly asynchronous nature of each of the implementations, we were unable to obtain accurate breakdowns of where any of them were truly spending their time. We were able to obtain the amounts of time that each thread considered itself as active, but because threads may be preempted, these times may not be reflective of the times that the threads were actually running on the CPU. Similarly, the timing breakdown for non-threaded 4-pass columnsort is not particularly accurate.

Our best guess is that the gains came primarily from two sources. First is increased flexibility in scheduling, which we discussed above in regard to the motivation for threaded implementations. The second source of performance gain is that the MPI calls in 4-pass columnsort are synchronous, whereas they are asynchronous in the non-threaded 4-pass columnsort. Apparently, asynchronous MPI calls incur a significant overhead. Although there is an overhead cost due to threads, the benefits of scheduling flexibility and synchronous MPI calls in 4-pass columnsort outweigh this cost.

We conducted a set of ancillary tests to verify that a program with threads and synchronous MPI calls is faster than a single-threaded program with asynchronous MPI calls. The first test overlaps computation and I/O, and the second test overlaps computation and communication. We found that by converting a single thread with asynchronous MPI calls to a threaded program with synchronous MPI calls, the computation-and-I/O program ran 7.8% faster and the computation-and-communication program ran 23.8% faster.

Moreover, there is a qualitative benefit of 4-pass columnsort. Because all calls to MPI and MPI-2 functions are synchronous, the code itself is cleaner and it is easier to modify.

Columnsort: 4-pass vs. 3-pass

By inspection of Table 3.1, we see that on Sun-A and on Sun-B, 3-pass columnsort implementation took 94.6% and 91.5%, respectively, of the time used by 4-pass columnsort. The improvement due to reducing the number of passes from four to three is not as marked as that from introducing a threaded implementation.

The observed running times lead to two questions. First, what accounts for the improvement that we see in eliminating a pass? Second, why don't we see more of an improvement?

Compared to 4-pass columnsort, 3-pass columnsort enjoys one significant benefit and one additional cost. Not surprisingly, the benefit is due to less disk I/O. The pass that is eliminated reads and writes each record once, and so 3-pass columnsort performs only 75% as much disk I/O as 4-pass columnsort. This reduced amount of disk I/O is the only factor we know of that accounts for the observed improvement.

The added cost is due to more sort phases and more communicate phases. In the 3-pass implementation, each round contains two sort and two communicate phases, for a total of $2 + 2\lceil (s - P)/(P - 1) \rceil$ sort phases and the same number of communicate phases. Together, the two rounds of 4-pass columnsort—which are replaced by the last pass of 3-pass columnsort—perform $2s/P$ sort phases and $2s/P$ communicate phases. Since the problem is out-of-core, we have $s > P$, which in turn implies that

$$\begin{aligned} 2 + 2\lceil (s - P)/(P - 1) \rceil &\geq 2 + 2\lceil (s - P)/P \rceil \\ &= 2 + 2(s/P - 1) \\ &= 2s/P \ . \end{aligned}$$

Thus, 3-pass columnsort has more sort and communicate phases than 4-pass columnsort.

The degree to which 3-pass columnsort improves upon 4-pass columnsort depends on the relative speeds of disks, processing, and communication in the underlying system. The 25% reduction in the number of passes does not necessarily translate into a 25% reduction in the overall time. That is because although the combined third pass of 3-pass columnsort writes and reads each record only once, all the other work (communication and sorting) of the two passes still has to be done. Therefore, when the last two passes of 4-pass columnsort are relatively I/O bound, we would expect 3-pass columnsort to be significantly faster than 4-pass columnsort. Conversely, when the last two passes are not I/O bound, the advantage of 3-pass columnsort is reduced. In fact, 3-pass columnsort can even be slower than 4-pass columnsort! On the clusters whose results appear in Table 3.1, the processing and network speeds are not particularly fast. Our detailed observations of the 4-pass implementation reveal that in the last two passes, the I/O and communication times are close. Hence, these passes are only slightly I/O bound, and therefore we see only a modest gain in the 3-pass implementation. On Jefferson, the most recent testbed, our implementations are mostly I/O bound; we shall see later that 3-pass columnsort is indeed significantly faster than 4-pass columnsort on Jefferson.

3.6 Problem-size restriction

Our preliminary implementations do not sort all problem sizes. They have a restriction on the maximum problem size. As mentioned in Section 1.3, the height interpretation ($r = M/P$) and the height restriction ($r \geq 2s^2$) induce an upper limit on the number of records N to be sorted:

$$N \leq \frac{(M/P)^{3/2}}{\sqrt{2}} . \tag{3.1}$$

All preliminary implementations are subject to this problem-size restriction, since they use the original columnsort algorithm, inheriting its height restriction, and they use the height interpretation of $r = M/P$.

Although the restriction may appear to be overly constraining on the largest problem size that can be sorted correctly, in practice we can still sort some fairly large files. For example, suppose we are sorting 32-byte records on a machine with 32 megabytes of memory per processor available for holding data (after reducing the available memory for software, stack, and buffers). Then $M/P = 2^{20}$, since the memory size M is in terms of 32-byte records. By the restriction (3.1), the largest file we can sort has 2^{29} records, or 2^{34} bytes. In other words, even with a moderate memory size, we can sort up to 16 gigabytes of data. Moreover, by restriction (3.1), the maximum problem size increases with $(M/P)^{3/2}$, i.e., it increases superlinearly with the amount of memory per processor.

There are two main implications of this restriction. The first implication is the obvious one: the maximum problem size has an upper bound. The second implication is one of scalability. The problem size N depends on M/P, the memory per processor, rather than on the total memory M of the system. In other words, if we double the number of processors, without changing the memory per processor, the maximum problem size does not increase.

Chapter 4

Subblock Columnsort and *M*-columnsort

In this chapter, we present the design, implementation, and experimental results of two approaches that relax the problem-size restriction: subblock columnsort and M-columnsort. Subblock columnsort is based on an algorithmic modification of the underlying columnsort algorithm; it relaxes the height restriction, thereby relaxing the problem-size restriction at the cost of an additional pass. M-columnsort is more of an engineering effort that changes the height interpretation; it relaxes the problem-size restriction at the cost of additional communication and computation. In the next chapter, we shall describe our more recent algorithm, subblock columnsort, which relaxes the problem-size restriction with no extra I/O or communication cost.

4.1 Subblock columnsort

In this section, we present subblock columnsort, prove its correctness, and discuss some aspects of its implementation. Subblock columnsort relaxes the height restriction by a factor of $\sqrt{s}/2$, to $r \geq 4s^{3/2}$, by adding two steps to columnsort. With a height interpretation of $r = M/P$, we get problem-size restriction (1.2): $N \leq (M/P)^{5/3}/4^{2/3}$.

The algorithm

To describe subblock columnsort, we first need to define what a subblock is. We will be working with $\sqrt{s} \times \sqrt{s}$ subblocks of the mesh, where each *subblock* is a contiguous set of $\sqrt{s}$ rows and $\sqrt{s}$ columns. Subblocks are aligned to the mesh, meaning that the indices of the top row and leftmost column of each subblock must be multiples of $\sqrt{s}$ (assuming 0-origin indexing). We need $\sqrt{s}$ to be an integer. The main idea behind subblock columnsort, inspired by the Revsort algorithm of Schnorr and Shamir [SS86], is to add two extra steps after step 3 of columnsort.

Subblock columnsort consists of the following ten steps:

- Do steps 1–3 of columnsort.
- Step 3.1 performs any permutation that moves all the values in each $\sqrt{s} \times \sqrt{s}$ subblock into all s columns. We shall refer to this property of a permutation on the $r \times s$ mesh as the *subblock property*.
- Step 3.2 sorts each column.
- Do steps 4–8 of columnsort.

There are several permutations that accomplish the goal of step 3.1. Since the particular permutation used will make its way into our out-of-core implementation of subblock columnsort, we need to choose it carefully. We shall describe the specific permutation chosen for step 3.1 later in this section.

Proof of correctness

We shall show that as long as $r \geq 4s^{3/2}$, the dirty area after step 4 is at most half a column in size. As in Section 2.3, the 0-1 Principle and Lemma 2.4 will complete the proof of correctness. We start with the following lemma, which states another property that even the original columnsort algorithm has.

Lemma 4.1 *Assuming a 0-1 input and assuming that the divisibility restriction holds, after step 3, the line dividing the 0s and 1s is monotonic in the sense that, as shown in Figure 4.1, it goes from left to right and bottom to top, never turning back to the left and never turning back toward the bottom.*

Proof: Because step 3 sorts the columns, we cannot see any 0 below a 1 within a given column. Thus, the dividing line cannot turn back to the left.

To see that the dividing line cannot turn back toward the bottom, it suffices to show that after step 2, each column has at least as many 0s as the column immediately to its right. In order for a column to have fewer 0s than the column to its right, there would have to be some row in which these two columns exhibited a $1 \rightarrow 0$ transition. But due to the divisibility restriction, there are no $1 \rightarrow 0$ transitions within rows after step 2. ■

The following lemma uses an argument similar to one that was first made by Schnorr and Shamir [SS86].

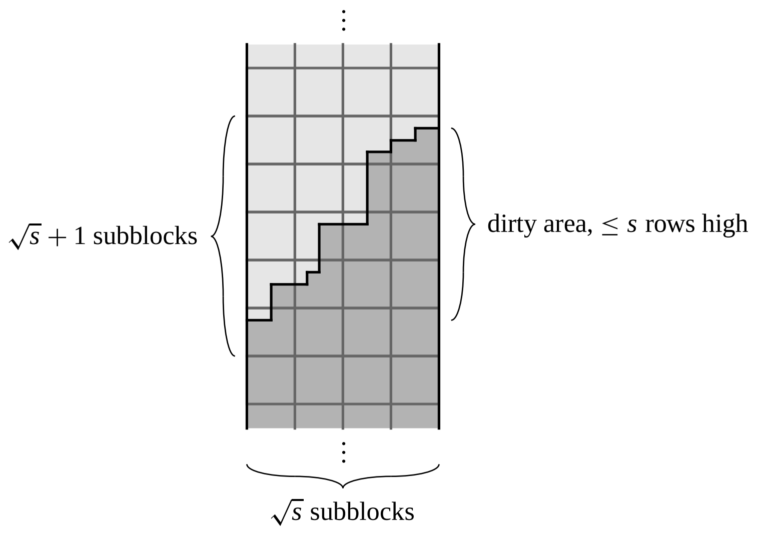

Figure 4.1: The monotonicity of the dividing line between the 0s and 1s after step 3. The dividing line passes through at most $2\sqrt{s}$ subblocks.

Lemma 4.2 *Assuming a 0-1 input, after step 3.1 of subblock columnsort, the number of 0s in any two columns differs by at most* $2\sqrt{s}$.

Proof: As Lemma 2.1 shows, after step 3 (and before step 3.1), the dirty area is confined to an area that is $s \times s$. Referring to Figure 4.1, and noting that subblocks are $\sqrt{s} \times \sqrt{s}$ in size, the dirty area is confined to an array of subblocks that is at most $\sqrt{s}+1$ high and at most $\sqrt{s}$ wide. Because the dividing line between 0s and 1s is monotonic, it passes through at most $2\sqrt{s}$ subblocks. In other words, all but $2\sqrt{s}$ subblocks are clean.

Step 3.1 distributes each subblock to all s columns. The clean subblocks distribute their 0s or 1s uniformly to each column, and thus the difference between the number of 0s in any two columns is at most the number of dirty subblocks: $2\sqrt{s}$. ■

Lemma 4.3 *Assuming a 0-1 input, after step 4 of subblock columnsort, the mesh consists of some clean columns of 0s on the left, some clean columns of 1s on the* *right, and a dirty area of size at most* $2s^{3/2}$ *between them.*

Proof: By Lemma 4.2, after sorting each column in step 3.2, the dirty area is confined to an area that is at most $2\sqrt{s}$ rows high and s columns wide. Thus, after step 3.2, the dirty area has size at most $2s^{3/2}$. As in the proof of Lemma 2.2, after step 4, there are clean columns of 0s on the left, clean columns of 1s on the right, and the dirty area is between them, but now of size at most $2s^{3/2}$. ■

Theorem 4.4 *Assuming that the divisibility restriction holds, s is a perfect square, and $r \geq 4s^{3/2}$, subblock columnsort sorts any input correctly.*

Proof: By Lemma 4.3, after step 4, the dirty area has size at most $2s^{3/2}$. Since $r \geq 4s^{3/2}$, we have that $2s^{3/2} \leq r/2$, and so the dirty area is at most half a column in size. We complete the proof by applying Lemma 2.4 and noting that because subblock columnsort is oblivious, the 0-1 Principle applies. ■

Removing the divisibility restriction from subblock columnsort

We next show that if we tighten the height restriction for subblock columnsort by 50%, so that $r \geq 6s^{3/2}$, then the divisibility restriction is unnecessary.

When we remove the divisibility restriction, there may be $1 \to 0$ transitions within rows after step 2. After step 3, therefore, there may be $1 \to 0$ transitions within rows. Figure 4.2 shows the dividing line between 0s and 1s after step 3: it is no longer monotonic like in Figure 4.1. The dividing line still cannot turn back to the left (since step 3 sorts each column), but it may turn back toward the bottom. The result is the “skyline” appearance shown in Figure 4.2.

The key to relaxing the height restriction in subblock columnsort was showing that the boundary between 0s and 1s passes through a limited number of subblocks.

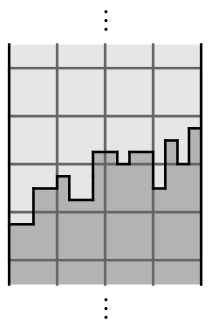

Figure 4.2: The "skyline" shape of the dividing line between the 0s and 1s after step 3 when the divisibility restriction is removed. The total length of the rising and falling edges is bounded by $2s - 1$.

When the boundary can both rise and fall, as in the skyline pattern of Figure 4.2, one might suspect that each peak and each valley could possibly be just one column wide and there might be very tall peaks and very deep valleys; in this case, the boundary could pass through all $s + \sqrt{s}$ subblocks in a region $\sqrt{s} + 1$ subblocks high and $\sqrt{s}$ subblocks wide. The following lemma shows that this scenario is overly pessimistic: the total length of the rising and falling edges is limited by $2s - 1$.

Lemma 4.5 *Assuming a 0-1 input but without the divisibility restriction, after step 3 of subblock columnsort, the line dividing the 0s and 1s has a total length of rising and falling edges that is at most* $2s - 1$.

Proof: A rising edge occurs whenever there is a $0 \to 1$ transition within a row, and a falling edge occurs whenever there is a $1 \to 0$ transition within a row. For $j = 0, 1, \ldots, s - 2$, let x_j denote the number of $0 \to 1$ transitions between columns j and $j + 1$ after step 2, and x'_j denote the number of $0 \to 1$ transitions between columns j and $j + 1$ after the columns are sorted in step 3. Define y_j

and y'_j similarly for $1 \to 0$ transitions. Letting $x' = \sum_{j=0}^{s-2} x'_j$ and $y' = \sum_{j=0}^{s-2} y'_j$, it suffices to show that $x' + y' \le 2s - 1$.

We claim that $y'_j - x'_j = y_j - x_j$ for $j = 0, 1, \ldots, s-2$. To see why, let z_j equal the number of 0s in column j, for $j = 0, 1, \ldots, s-1$; note that z_j does not change due to the sorting in step 3. Because each $1 \to 0$ transition causes column $j+1$ to have one more 0 than column j and each $0 \to 1$ transition causes column $j+1$ to have one fewer 0 than column j, we have that $z_{j+1} = z_j + y_j - x_j$ and also $z_{j+1} = z_j + y'_j - x'_j$ for $j = 0, 1, \ldots, s-2$. Subtracting z_j from each formula gives $y'_j - x'_j = z_{j+1} - z_j = y_j - x_j$, which proves the claim.

Next we claim that after step 3, for $j = 0, 1, \ldots, s-2$, at least one of x'_j and y'_j must be 0. We prove this claim by contradiction: suppose that both x'_j and y'_j are positive. Then in columns j and $j+1$, there is a row with a $0 \to 1$ transition and a row with a $1 \to 0$ transition. If the row with the $0 \to 1$ transition is the higher of the two, then column $j+1$ has a 1 above a 0, which cannot occur since it has been sorted in step 3. If instead the row with the $1 \to 0$ transition is the higher of the two rows, then column j has a 1 above a 0, which again cannot occur. Since we cannot have both a $0 \to 1$ transition and a $1 \to 0$ transition within the same row, at least one of x'_j and y'_j must be 0, thus proving the claim.

Noting that all x_j and y_j are nonnegative, $x'_j = 0$ implies $y'_j = y_j - x_j \le y_j$, and $y'_j = 0$ implies $x'_j = x_j - y_j \le x_j$. In either case, we see that $x'_j \le x_j$ and $y'_j \le y_j$ for all $j = 0, 1, \ldots, s-2$. Thus, we have

$$\begin{aligned} x' + y' &= \sum_{j=0}^{s-2} x'_j + \sum_{j=0}^{s-2} y'_j \\ &\le \sum_{j=0}^{s-2} x_j + \sum_{j=0}^{s-2} y_j \ . \end{aligned}$$

As we saw in the proof of Lemma 2.6, $\sum_{j=0}^{s-2} x_j \leq s$ and $\sum_{j=0}^{s-2} y_j \leq s - 1$. Therefore, $x' + y' \leq s + (s - 1) = 2s - 1$. ■

Corollary 4.6 *Assuming a 0-1 input but without the divisibility restriction, after step 3.1 of subblock columnsort, the number of 0s in any two columns differs by at most* $3\sqrt{s}$.

Proof: The proof is the same as that of Lemma 4.2, but with the change that because the total length of the rising and falling edges of the dividing line between 0s and 1s is at most $2s - 1$, it passes through at most $3\sqrt{s}$ subblocks. ■

Corollary 4.7 *Assuming a 0-1 input but without the divisibility restriction, after step 4 of subblock columnsort, the mesh consists of some clean columns of 0s on the left, some clean columns of 1s on the right, and a dirty area of size at most* $3s^{3/2}$ *between them.*

Proof: Analogous to the proof of Lemma 4.3. ■

Theorem 4.8 *Assuming that s is a perfect square and that $r \geq 6s^{3/2}$, but without the divisibility restriction, subblock columnsort sorts any input correctly.*

Proof: Analogous to the proof of Theorem 4.4. ■

The subblock permutation

There are several permutations that have the subblock property. The one we use here, which we call the *subblock permutation*, is based on permuting sets of bits within the row and column numbers. For ease of understanding, we assume that P, r and s are powers of 2; Appendix A.2 discusses the general scenario. Therefore,

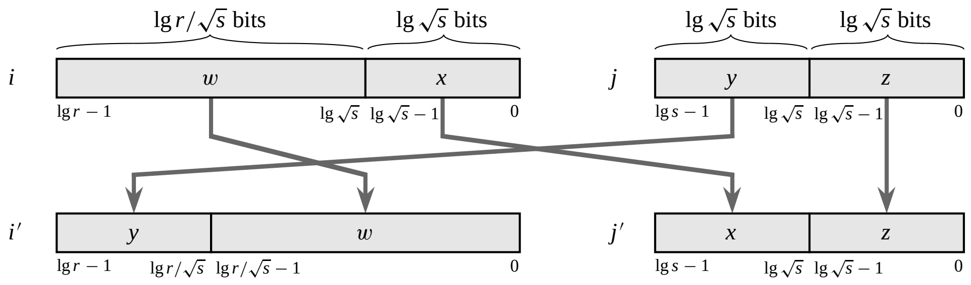

Figure 4.3: The subblock permutation as a bit permutation. Each entry in the $r \times s$ mesh has a row number, expressed in $\lg r$ bits, and a column number, expressed in $\lg s$ bits. The subblock permutation permutes the element in row i and column j to row i' and column j'. We number the bits starting from 0 as the least signifi cant bit, and we use the notation "$..$" to denote ranges of consecutive bits. The permutation maps bits $w = i_{\lg \sqrt{s} .. \lg r - 1}$ to $i'_{0 .. \lg r/\sqrt{s} - 1}$, $x = i_{0 .. \lg \sqrt{s} - 1}$ to $j'_{\lg \sqrt{s} .. \lg s - 1}$, $y = j_{\lg \sqrt{s} .. \lg s - 1}$ to $i'_{\lg r/\sqrt{s} .. \lg r - 1}$, and $z = j_{0 .. \lg \sqrt{s} - 1}$ to $j'_{0 .. \lg \sqrt{s} - 1}$. Because x determines the source row number within an element's subblock and z determines the source column number within the subblock, this mapping ensures that the bits forming an element's target column number come from the bits that determine where in a source subblock the element started.

each row number is a sequence of $\lg r$ bits and each column number is a sequence of $\lg s$ bits. We shall express the subblock permutation in terms of the source row and column numbers and the corresponding target row and column numbers of each mesh element.

To ensure that the subblock property holds, we only need to show that any two distinct elements in the same source subblock map to two different column numbers. We do so by ensuring that the $\lg s$ bits that determine the target column number come from source bits that determine where in a $\sqrt{s} \times \sqrt{s}$ subblock a mesh element resides. Figure 4.3 shows the idea. If we look at the source row and column numbers of a given element, the least significant $\lg \sqrt{s}$ bits of each—denoted by x and z, respectively, in the figure—determine the row and column numbers of that element within its subblock. (The most significant bits—w and y—determine which subblock the element is in, but not where in the subblock it resides.) The subsequences x and z form the bits of the target column number, with z forming the least significant half and x forming the most significant half. Thus, the subblock permutation has the subblock property. As an arithmetic formula, the subblock permutation maps the (i, j) entry to position (i', j'), where

$$\begin{aligned} i' &= \left\lfloor \frac{j}{\sqrt{s}} \right\rfloor \frac{r}{\sqrt{s}} + \left\lfloor \frac{i}{\sqrt{s}} \right\rfloor , \\ j' &= j \bmod \sqrt{s} + (i \bmod \sqrt{s})\sqrt{s} . \end{aligned}$$

It may seem strange that the target row number is formed by using w as the least significant bits, when w started out as the most significant bits of the source row number. The advantage of permuting in this way is that it creates sorted runs of $r/\sqrt{s}$ elements in each column. To see why, first observe that entering the subblock permutation, each column is sorted. Now consider two elements e_1 and e_2 that start in the same column (so that their y and z bits are the same) and are permuted into the same target column (so that their x bits are also the same). There are $r/\sqrt{s}$ elements in a source column that have fixed values of the x, y, and z bits, and they vary in their w bits. Let the w bits of e_1 and e_2 be w_1 and w_2, respectively, and assume that $w_2 = w_1 + 1$ (so that e_2 is $\sqrt{s}$ rows below e_1 in the source column and $e_1 \leq e_2$). Because e_1 and e_2 have the same y bits, which become the most significant bits of the target row, and because the w bits become the least significant bits of the target row, e_2's target row is 1 greater than e_1's target row. Since e_1 and e_2 are any two elements that start out $\sqrt{s}$ rows apart in their source column, we see that all $r/\sqrt{s}$ elements that start in the same source column and are permuted to the same target column appear as one sorted run of size $r/\sqrt{s}$ in the target column.

Implementation notes

Since subblock columnsort differs from columnsort only in the two additional steps, our implementation of subblock columnsort started with 3-pass columnsort and integrated these two steps as one extra pass, which we call the *subblock pass*. The subblock pass performs steps 3 and 3.1 of subblock columnsort. The overall thread structure of subblock columnsort is the same as that of the 3-pass columnsort program.

Like the first two passes in 3-pass columnsort, the subblock pass is decomposed into s/P rounds, where each round processes the next set of P consecutive columns, one column per processor, in a five-phase pipeline. The only phase of the subblock pass that differs substantially from the corresponding phase of passes 1 and 2 of 3-pass columnsort is the communicate phase. In each round's communicate phase within passes 1 and 2 of 3-pass columnsort, each processor sends P messages, where each message consists of r/P records. One of these messages goes back to the sending processor, in which case the message does not need to go over the network. For the subblock pass, however, we shall show three properties:

1. In the communicate phase of each round, each processor sends only $\lceil P/\sqrt{s}\rceil$ messages.

2. When $\sqrt{s} \geq P$, so that $\lceil P/\sqrt{s}\rceil = 1$, the one message is always destined for the sending processor, and therefore no communication over the network occurs.

3. Any permutation that achieves the subblock property must send at least

$\lceil P/\sqrt{s}\rceil$ messages per round, and so the number of messages sent in our subblock pass is optimal.

To see that the first two properties hold, we refer to Figure 4.3. We continue to assume that all parameters are powers of 2.[1] Let us examine the processors to which column j's elements are mapped by the subblock permutation. Column j is owned by processor p, where $p = j \bmod P$. This processor number is the $\lg P$ least significant bits of the column number j.

If $P \leq \sqrt{s}$, then the bits that determine the processor number are entirely within the z field in Figure 4.3. Since these bits are the same in the target column as they are in the source column, the target processor number must be the same as the source processor number. Thus, since $\lceil P/\sqrt{s}\rceil = 1$, we have proved property 2.

If $P > \sqrt{s}$, then the $\lg \sqrt{s}$ least significant bits of the source and target column numbers for a given element are the same. Therefore, only $\lg P - \lg \sqrt{s}$ of the bits determining the processor number can differ. These bits come from the x field, which is part of the row number. All combinations of these bits will occur in a given source column, and so all combinations will occur in the target processor number. There are $2^{\lg P - \lg \sqrt{s}} = P/\sqrt{s}$ such combinations. By the power-of-2 assumption, therefore, we have $\lceil P/\sqrt{s}\rceil = P/\sqrt{s}$, and so we have proved property 1.

To show property 3, we shall show that if the subblock property holds and any source column maps to fewer than $P/\sqrt{s}$ target processors, then a contradiction arises. We start by noting that every processor owns exactly s/P columns. Now let us suppose that some source column, say column j, maps to k target processors, where $k < P/\sqrt{s}$. Therefore, column j maps to fewer than $k(s/P)$ target columns. Since $k < P/\sqrt{s}$, we have that $k(s/P) < \sqrt{s}$, so that column j maps to fewer than

[1]For a general proof, see Appendix A.2.

$\sqrt{s}$ target columns. Because the subblock property holds, any $\sqrt{s}$ entries within a given subblock must map to $\sqrt{s}$ different columns. If we consider the intersection of any subblock with column j, we have $\sqrt{s}$ entries of the subblock, and hence this portion of column j (not even considering the rest of the column) must map to $\sqrt{s}$ different target columns. This fact contradicts our earlier conclusion that column j maps to fewer than $\sqrt{s}$ target columns. Thus, we have proved property 3.

4.2 *M*-columnsort

M-columnsort changes the height interpretation from $r = M/P$ to $r = M$, achieving the problem-size restriction (1.3): $N \leq \sqrt{M^3/2}$. Recall that with this more relaxed restriction, the maximum problem size now scales with the memory in the entire system, so that adding more processors with the same amount of memory per processor increases the maximum problem size. In fact, this increase is superlinear in the total memory size.

Implementation notes

As with subblock columnsort, our implementation of M-columnsort is a modification of 3-pass columnsort. Instead of adding a pass, however, we increase the complexity of the sort phase of each pass. When $r = M/P$, the sort phase is just a local sort on each processor. In M-columnsort, since $r = M$, the sort phase becomes a multiprocessor sort with distributed memory. One benefit of performing a multiprocessor sort is that we can eliminate the communicate phase in the first two passes.

We use in-core columnsort for the distributed-memory multiprocessor sort. We implemented three in-core multiprocessor sorting algorithms: bitonic sort, radix sort, and columnsort. We found that in-core columnsort, with an $(M/P) \times P$ mesh, was consistently faster than bitonic sort on problem sizes representative of those we encounter in the sort phase. Radix sort was competitive with in-core columnsort over a wide range of problem sizes, but we decided to use in-core columnsort because the radix sort code has a high dependence on the key format and because columnsort's communication patterns are independent of the values in the keys.

Our implementation of in-core columnsort is multithreaded. In particular, there are two threads: one for local sorting (steps 1, 3, 5, and 7 of the in-core sort) and one for communication (steps 2, 4, 6, and 8 of the in-core sort). These threads are in addition to the four non-sort threads inherited from 3-pass columnsort. Wherever 3-pass columnsort's pipeline had a single phase for sorting, M-columnsort's pipeline has eight phases. Each phase in a pipeline sends a buffer to its successor, and the additional threads in M-columnsort require the allocation of four additional buffers.

The pipeline for each of the first two passes has 11 phases: read, the eight from in-core columnsort, permute, and write. These 11 phases occupy only four threads: one for both read and write, one for permute, one for the four local sorting steps of in-core columnsort, and one for the four communication steps of in-core columnsort. We are able to eliminate the communicate phase from the out-of-core pipeline because each column is shared among all the processors. After the in-core sort, each processor has its own portion of the sorted data. Depending on the permutation phase of the out-of-core sort, this data is destined for a certain set of target columns. Since each processor owns a portion of each column, we were able to design the in-core sort to ensure that each processor can write out its data into its portion of each of the target columns.

The pipeline for the last pass is rather different. We eliminate one communicate phase, but each of the two sort phases turns into eight in-core sort phases. Although the resulting pipeline has 20 phases, they occupy only seven threads: one for both read and write, one for permute, one for the remaining communicate phase, and two for each of the two in-core sorts.

4.3 Experimental results on Borg

In this section, we discuss the results of our experiments on Borg. We start with a description of Borg. Next, we outline some issues regarding MPI and threads. Finally, we present and analyze our experimental results.

Borg

Borg is a Beowulf cluster that has 16 dual 1.5-GHz P4 Xeon nodes, each with 1 GB of RAM. The nodes run Redhat Linux 7.2 and are connected with a high-speed Myrinet network, which has a peak speed of 250 MB per second. Each node has an Ultra-160 10,000-RPM SCSI-3 hard drive, with about 10 GB of available disk space for user files. For disk I/O, we use the C `stdio` interface. The nodes communicate using standard synchronous MPI [SOHL$^+$98] calls within asynchronous threads. Our threads are implemented using the pthreads package of Linux.

Although each node has two processors, we found no advantage to running more than one process per node.[2] Thus, we treat each node as if it had only one

[2]Since each node has only one disk, running two processes per node would require the two

processor, and we use the terms “node” and “processor” interchangeably.

MPI and threads

Not all MPI implementations work correctly in the presence of multiple threads. As mentioned in [GHLL$^+$98, pp. 25–33], an MPI implementation can have one of the following four levels of thread support:

- *Single*: Each MPI process must be single threaded.
- *Funneled*: An MPI process may be multithreaded, but all MPI calls must be within the same thread.
- *Serialized*: An MPI process may be multithreaded, and multiple threads may make MPI calls, but no two MPI calls should overlap. In other words, at any point in time, at most one thread may be executing an MPI call.
- *Multiple*: Multiple threads may make MPI calls, with no restrictions.

Our programs require the highest level of thread support, i.e., the multiple thread support. We found that MPI/Pro, which allows multiple threads to issue MPI calls, worked well for our experiments.

Experimental setup

Our experimental runs were for combinations of the following:

Algorithm: We ran 3-pass columnsort, subblock columnsort, and M-columnsort.

For a baseline, we also ran just the I/O portions of three and four passes of

processes to share the one disk, slowing down per-process disk I/O. Since our programs were I/O bound, we decided not to use two processes per node.

columnsort. In this way, we can determine how much time of each run was spent waiting for non-I/O activity.

Buffer size: For 3-pass columnsort, subblock columnsort, and M-columnsort, we varied the buffer size (r). We report here results for buffer sizes of 2^{24} and 2^{25} bytes. Note that these buffer sizes, being in bytes, are not in terms of number of records, and so they should not be construed as equaling M/P. Record sizes varied between 64 and 128 bytes, but we found that the buffer size mattered more than the record size.

Number of processors and volume of data: We ran various combinations with 4, 8, and 16 processors and with an amount of data varying from 4 GB up to 32 GB. We did not run any experiments with less than 1 GB of data per processor because file-caching effects masked the out-of-core nature of the problem. We were unable to perform any runs with more than 2 GB of data per processor due to disk-space limitations. We did not overwrite the original data files so that we could verify the correctness of the output files. Moreover, because our implementation requires a temporary file, the input, output, and temporary files together induce a disk-space requirement of three times that of the input file size. The cluster on which we ran our experiments did not permit us to use much more than 6 GB of disk space per node. All runs, therefore, were for either 1 GB or 2 GB of data per processor.

For a given algorithm and buffer size, we found that the amount of data per processor was by far the most important factor in determining run time. Given the large amount of disk I/O that each of the algorithms has to perform, this characteristic did not come as a surprise. We couch our results in terms of GB of data per

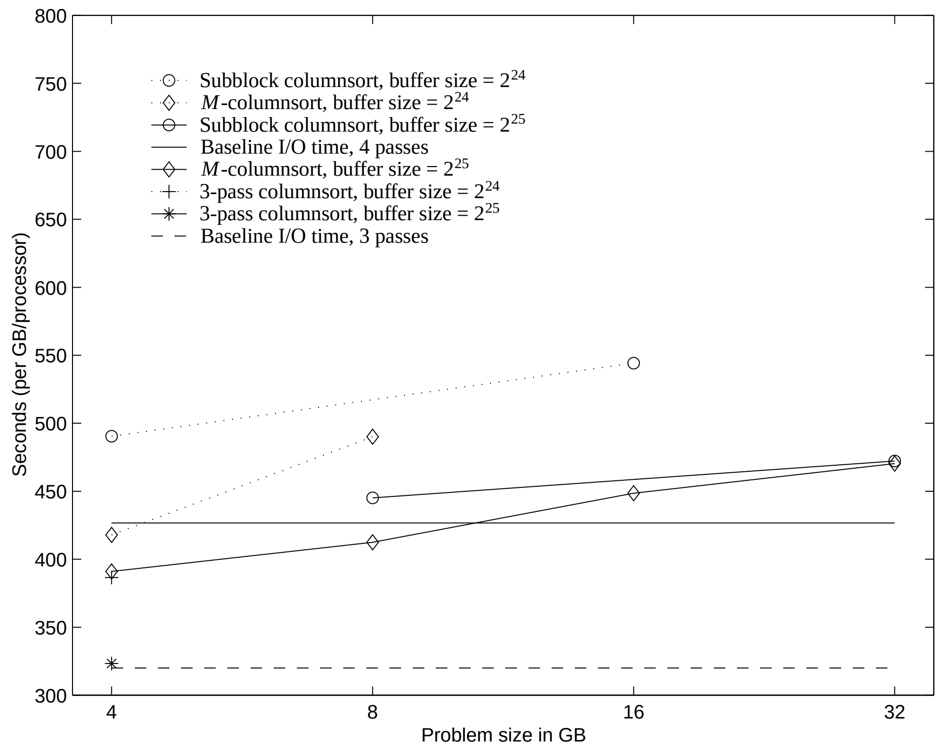

Figure 4.4: Execution times, in seconds, for the three versions of columnsort plus baseline I/O times for three and four passes.

processor.

Analysis of results

Figure 4.4 summarizes our experimental results. Each plotted point in the figure represents the average of multiple runs of an algorithm with a given buffer size, but with the number of processors and the record size varying. Variations in running times were relatively small (within 10%). The horizontal axis is organized by total number of GB of data sorted across all processors.

Due to the problem-size restriction of equation (3.1), 3-pass columnsort could not handle more than 4 GB of data. Results for 3-pass columnsort appear as single points in Figure 4.4. For a buffer size of 2^{25} bytes, the total time is just barely above the baseline 3-pass I/O time; in other words, 3-pass columnsort is almost purely I/O-bound. When we halve the buffer size, to 2^{24} bytes, there are more frequent switches between pipeline phases and so the I/O-boundedness diminishes somewhat. We found that larger buffer sizes resulted in faster execution. We could not make these buffers arbitrarily large because there is a limit on how large these buffers can get until demand paging starts slowing down the execution.

Due to the perfect-square restriction on s in subblock columnsort, not all problem sizes were eligible to be run for a given value of r (i.e., for a given buffer size). That is why the two lines plotted for subblock columnsort in Figure 4.4 cover disjoint problem sizes. With a buffer size of 2^{25} bytes, the running times are only slightly above the baseline 4-pass I/O time. (Recall that subblock columnsort has one pass more than 3-pass columnsort.) Thus, subblock columnsort is substantially I/O-bound. With the smaller buffer size of 2^{24} bytes, execution times increase for the same reason as in 3-pass columnsort. Observe that each of the subblock columnsort lines rises only slightly as the volume of data sorted increases, thus demonstrating our earlier claim that the volume of data per processor is the most salient characteristic in determining execution time.

Figure 4.4 shows one particular advantage of M-columnsort over the other two algorithms: we were able to run it at all four problem sizes. In fact, had sufficient disk space been available, we could have run M-columnsort on up to one terabyte total on 16 processors with 2^{25}-byte buffers and 64-byte records. Execution times are well above the baseline 3-pass I/O time, and so M-columnsort is not nearly as I/O-bound as the other two algorithms. This fact is of course no surprise, since

M-columnsort has a much more complicated in-core sort phase and also uses more memory (due to the extra buffers required by the additional threads).

According to the performance results in Figure 4.4, M-columnsort was always at least as fast as subblock columnsort. The primary reason that M-columnsort was faster is that it makes only three passes over the data rather than four passes. For a given buffer size, due to the perfect-square restriction on s, subblock columnsort can handle fewer problem sizes than M-columnsort. Furthermore, for most realistic configuration sizes, M-columnsort can sort larger problem sizes than subblock columnsort. By restrictions (1.2) and (1.3), M-columnsort can handle a larger number of records than subblock columnsort if $M^{3/2}/\sqrt{2} > (M/P)^{5/3}/4^{2/3}$ or, equivalently, if $M < 32P^{10}$. For example, if $P = 8$ processors, then as long as the total memory in the system holds fewer than 2^{35} records, M-columnsort will sort more records than subblock columnsort.

Chapter 5

Slabpose Columnsort

This chapter presents our new oblivious algorithm: slabpose columnsort. Just like subblock columnsort and M-columnsort, slabpose columnsort relaxes the problem-size restriction. Unlike subblock columnsort and M-columnsort, however, slabpose columnsort does so at no extra I/O or communication cost. We first describe the algorithm and give a proof of its correctness. Next, we explain how a 3-pass implementation of this 10-step algorithm is possible. We conclude with a discussion on the new problem-size restriction and some implementation notes.

5.1 The algorithm and proof of correctness

Slabpose columnsort is based on partitioning the mesh into several vertical "slabs," where we define a *k-slab* as a set of k consecutive columns, with the leftmost column in the slab at an index that is a multiple of k. (The inspiration for using slabs comes from the work of Marberg and Gafni [MG88] for sorting on a square mesh.) When forming slabs, we assume that k is a divisor of s (and hence, by the

$$\begin{array}{|ccc|ccc|}\hline 0 & 9 & 18 & 27 & 36 & 45 \\ 1 & 10 & 19 & 28 & 37 & 46 \\ 2 & 11 & 20 & 29 & 38 & 47 \\ 3 & 12 & 21 & 30 & 39 & 48 \\ 4 & 13 & 22 & 31 & 40 & 49 \\ 5 & 14 & 23 & 32 & 41 & 50 \\ 6 & 15 & 24 & 33 & 42 & 51 \\ 7 & 16 & 25 & 34 & 43 & 52 \\ 8 & 17 & 26 & 35 & 44 & 53 \\ \hline \end{array} \underset{\text{(a)}}{\rightsquigarrow} \begin{array}{|ccc|ccc|}\hline 0 & 1 & 2 & 27 & 28 & 29 \\ 3 & 4 & 5 & 30 & 31 & 32 \\ 6 & 7 & 8 & 33 & 34 & 35 \\ 9 & 10 & 11 & 36 & 37 & 38 \\ 12 & 13 & 14 & 39 & 40 & 41 \\ 15 & 16 & 17 & 42 & 43 & 44 \\ 18 & 19 & 20 & 45 & 46 & 47 \\ 21 & 22 & 23 & 48 & 49 & 50 \\ 24 & 25 & 26 & 51 & 52 & 53 \\ \hline \end{array} \qquad \begin{array}{|ccc|ccc|}\hline 0 & 9 & 18 & 27 & 36 & 45 \\ 1 & 10 & 19 & 28 & 37 & 46 \\ 2 & 11 & 20 & 29 & 38 & 47 \\ 3 & 12 & 21 & 30 & 39 & 48 \\ 4 & 13 & 22 & 31 & 40 & 49 \\ 5 & 14 & 23 & 32 & 41 & 50 \\ 6 & 15 & 24 & 33 & 42 & 51 \\ 7 & 16 & 25 & 34 & 43 & 52 \\ 8 & 17 & 26 & 35 & 44 & 53 \\ \hline \end{array} \underset{\text{(b)}}{\rightsquigarrow} \begin{array}{|cccccc|}\hline 0 & 27 & 9 & 36 & 18 & 45 \\ 1 & 28 & 10 & 37 & 19 & 46 \\ 2 & 29 & 11 & 38 & 20 & 47 \\ 3 & 30 & 12 & 39 & 21 & 48 \\ 4 & 31 & 13 & 40 & 22 & 49 \\ 5 & 32 & 14 & 41 & 23 & 50 \\ 6 & 33 & 15 & 42 & 24 & 51 \\ 7 & 34 & 16 & 43 & 25 & 52 \\ 8 & 35 & 17 & 44 & 26 & 53 \\ \hline \end{array}$$

Figure 5.1: k-slabpose and k-shuffle operations, shown for $k = 3$ on a 9×6 mesh. **(a)** A k-slabpose operation. **(b)** A k-shuffle operation.

divisibility restriction, a divisor of r). For a given value of k, there are s/k k-slabs; since k is a divisor of s, so is s/k.

This algorithm requires two new operations, both of which are oblivious to the data being sorted:

- A *k-slabpose*, shown in Figure 5.1(a), is a transpose and reshape operation, but within each k-slab. Just as step 2 of the original columnsort algorithm transposes the mesh and reshapes it back into an $r \times s$ arrangement, a k-slabpose transposes within each k-slab and reshapes it back into an $r \times k$ configuration.

- A *k-shuffle*, shown in Figure 5.1(b), is a permutation of the s columns of the mesh in which we first take in order all columns whose indices are congruent to 0 modulo k, then take in order all columns whose indices are congruent to 1 modulo k, then 2 modulo k, and so on. More precisely, to determine which index column j maps to, let $j = lk + m$, where $0 \leq l < s/k$ and $0 \leq m < k$; then column j maps to index $ms/k + l$. Since $l = \lfloor j/k \rfloor$ and $m = j \bmod k$, column j maps to index $\mathcal{T}_j$, where

$$\mathcal{T}_j = (j \bmod k)s/k + \lfloor j/k \rfloor . \tag{5.1}$$

With these two new operations, we present the algorithm, which we call *slabpose columnsort*. Start by choosing any divisor k of s. Then perform the following 11 steps:

- Step 1 sorts each column.
- Step 2 performs a k-slabpose.
- Step 3 sorts each column.
- Step 4 is a k-shuffle.
- Step 5 performs an (s/k)-slabpose.
- Steps 6–11 are the same as steps 3–8 of the original columnsort algorithm.

This algorithm is oblivious. Note that because steps 4 and 5 are consecutive and both perform fixed permutations, they can be replaced by a single step that performs the composition of the k-shuffle and (s/k)-slabpose permutations. The resulting algorithm would have 10 steps rather than 11. To ease understanding, however, we shall focus on the 11-step formulation in the remainder of this section.

When $k = 1$, slabpose columnsort reduces to the original columnsort algorithm. Steps 2 and 4 become identity permutations, and so step 3 becomes redundant. Step 5 becomes a transpose-and-reshape operation over the whole mesh, just like step 2 of the original columnsort algorithm.

When $k = \sqrt{s}$, slabpose columnsort reduces to an algorithm similar to subblock columnsort, in terms of the number of passes, the complexity of each pass, and the problem-size restriction; we intend to pursue this case in our future work.

Correctness of slabpose columnsort

Similar to our approach in Chapters 2 and 4, we shall assume a 0-1 input and show that after step 7 of slabpose columnsort (which corresponds to step 4 of original columnsort), the dirty area is at most half a column in size.

Lemma 5.1 *Assuming a 0-1 input, after step 3 of slabpose columnsort, each k-slab consists of some clean rows of 0s at the top, some clean rows of 1s at the bottom, and at most k dirty rows between them.*

Proof: Steps 1–3 of slabpose columnsort are like steps 1–3 of original columnsort, but run on each k-slab independently. The proof then follows from Lemma 2.1. ■

Figure 5.2(a) shows the mesh after step 3. The dirty rows in one k-slab bear no relation to the dirty rows in any other k-slab, and so overall it is possible that up to s rows in the mesh are still dirty. As the following lemma shows, steps 4 and 5 reduce the number of dirty rows in the entire mesh.

Lemma 5.2 *Assuming a 0-1 input, after step 5 of slabpose columnsort, at most $(s/k)(\lceil k^2/s \rceil + 1)$ rows of the mesh are dirty.*

Proof: As Figure 5.2(b) shows, the k-shuffle of step 4 has the effect of creating k (s/k)-slabs. For $j = 0, 1, \ldots, k-1$, the dirty part of the jth column is confined

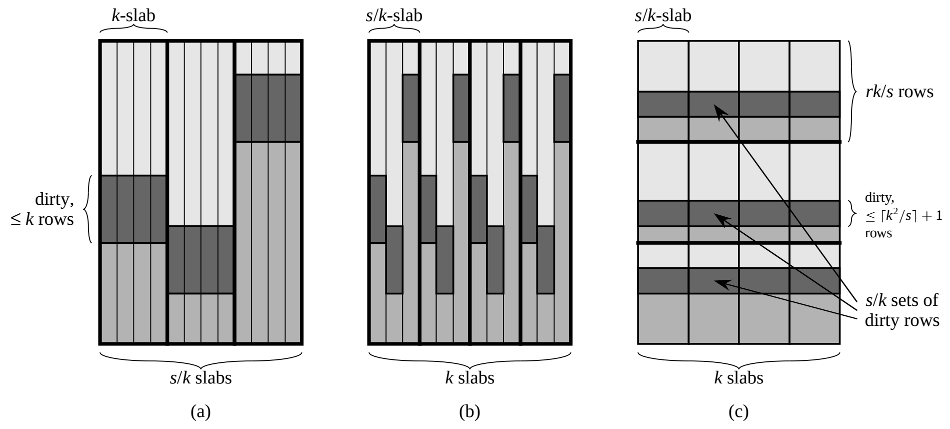

Figure 5.2: The mesh after steps 3, 4, and 5 of slabpose columnsort. **(a)** After step 3, there are s/k k-slabs, and each k-slab has a dirty area at most k rows high. The dirty areas do not necessarily align among k-slabs. **(b)** After step 4, there are k (s/k)-slabs. If we look at the jth column of each slab, the dirty area is confined to the same set of rows. **(c)** After step 5, within each set of rk/s consecutive rows, the dirty rows are confined to the same set of $\lceil k^2/s \rceil + 1$ rows.

to the same set of k rows in each of the (s/k)-slabs. Since s/k is a divisor of r, the (s/k)-slabpose operation of step 5 permutes each column within an (s/k)-slab into a set of $r/(s/k) = rk/s$ consecutive rows within the same (s/k)-slab. Figure 5.2(c) illustrates the result. Because the jth column of each (s/k)-slab forms the jth set of rk/s consecutive rows, and the dirty part of the jth column is confined to the same set of rows in each of the (s/k)-slabs, we see that within each set of rk/s consecutive rows, the dirty rows are confined to the same set of rows.

Next we determine how many rows within each set of rk/s consecutive rows are dirty. Because the dirty rows align among (s/k)-slabs, we need examine just a single (s/k)-slab. Prior to step 5, any given column of an (s/k)-slab has a dirty area at most k rows high. When these k values are moved into rows of width s/k, they fall into at most $\lceil k/(s/k) \rceil + 1 = \lceil k^2/s \rceil + 1$ rows.

Since there are s/k sets of rk/s consecutive rows, the total number of dirty rows after step 5 is at most $(s/k)(\lceil k^2/s \rceil + 1)$. ∎

Theorem 5.3 *As long as k is chosen as a divisor of s, s is a divisor of r, and $r \geq (2s^2/k)(\lceil k^2/s \rceil + 1)$, slabpose columnsort sorts correctly.*

Proof: If we assume a 0-1 input, then by Lemma 5.2, there are at most $(s/k)(\lceil k^2/s \rceil + 1)$ dirty rows after step 5. After step 6 (which sorts each column), there are clean rows of 0s at the top of the mesh, clean rows of 1s at the bottom, and at most $(s/k)(\lceil k^2/s \rceil + 1)$ dirty rows between them. Since the dirty area is at most s columns wide, it is confined to an area of the mesh of size $(s^2/k)(\lceil k^2/s \rceil + 1)$. After step 7, which is a full reshape-and-transpose, when we read the mesh in column-major order, the dirty area is confined to at most $(s^2/k)(\lceil k^2/s \rceil + 1)$ consecutive entries. By Lemma 2.4, as long as this dirty area is at most half a column in size, the final four steps produce a sorted 0-1 output. This bound on the dirty area's size—$(s^2/k)(\lceil k^2/s \rceil + 1) \leq r/2$—is equivalent to the condition $r \geq (2s^2/k)(\lceil k^2/s \rceil + 1)$ in the theorem statement. Noting that slabpose columnsort is oblivious and applying the 0-1 Principle completes the proof. ∎

5.2 A 3-pass implementation

In this section, we describe a 3-pass implementation of one particular case of slabpose columnsort. We set the parameter k of slabpose columnsort to be equal to the number of processors in the out-of-core setting, i.e., $k = P$. Hereafter, we assume that $k = P$ in any implementation of slabpose columnsort.

The out-of-core setting is inherited from our earlier implementations [CCW01, CC02, CHC03, CCH]. The three passes of the out-of-core implementation are structured so that the first pass performs steps 1–5 of slabpose columnsort and passes 2 and 3 together perform steps 6–11 of slabpose columnsort.

First, we elaborate how steps 1–5 of slabpose columnsort can be completed in one single pass over the data. We then explain why passes 2 and 3 of slabpose columnsort are the same as passes 2 and 3, respectively, of 3-pass columnsort.

5.2.1 Pass 1 performs steps 1–5

In all our implementations, we have been assuming that column j maps to processor $\mathcal{P}_{j \bmod P}$. We shall refer to this mapping as the *P-cyclic mapping*. We will show that if we stick to the P-cyclic mapping, step 4 of slabpose columnsort has several rather severe performance drawbacks. Then, we will explain how we can overcome all of these drawbacks by carefully changing the column-to-processor mapping for a few chosen steps.

Step 4 with the *P*-cyclic mapping

In each communication phase of all the passes of all our implementations so far, each processor receives M/P records. First, we will show how it is possible that some processors receive many more than M/P records, if we assume the P-cyclic mapping in step 4 of slabpose columnsort. We will then point out some performance implications of this fact.

To show that a processor can receive more than M/P records, consider step 4 (the P-shuffle) of any round, say round x. According to the P-cyclic mapping, in round x, processor $\mathcal{P}_i$ has the column numbered $j = xP + i$. Let $\mathcal{T}_j$ be the target column of column j, according to the P-shuffle. By formula (5.1), with $k = P$, $j \bmod P = i$, and $\lfloor j/P \rfloor = x$, we have $\mathcal{T}_j = is/P + x$. According to the P-cyclic mapping, $\mathcal{T}_j$ maps to the processor numbered $\mathcal{T}_j \bmod P$. Now, consider the case in which P divides s/P,[1] implying that $\mathcal{T}_j \bmod P = x \bmod P$. Therefore, if P divides s/P, all P columns in the memory of the system map to one single processor, namely $\mathcal{P}_{x \bmod P}$, implying that $\mathcal{P}_{x \bmod P}$ will receive M records. This situation leads to the following performance drawbacks:

- All M records go to processor $\mathcal{P}_{x \bmod P}$, leading to highly unbalanced communication.
- Since each processor can hold only M/P records in its memory, processor $\mathcal{P}_{x \bmod P}$ needs to somehow manage the M records that it receives.
- Processor $\mathcal{P}_{x \bmod P}$ must write these M records to its disks. Because all the other processors receive no records, they must wait for processor $\mathcal{P}_{x \bmod P}$ to finish its write operation, which leads to highly unbalanced I/O as well.
- The first pass must end with step 4, since the M records must be written out to disks. Step 5, therefore, will need another pass over the data.

Steps 1–5 in our implementation

Now, we explain how, by carefully changing the column-to-processor mapping in some steps, we not only overcome the performance drawbacks mentioned above, but we do so while requiring step 4 to perform no communication whatsoever.

We first define a new way to map columns to processors: the (s/P)-*blocked mapping*. According to the (s/P)-blocked mapping, the first s/P columns

[1] This assumption is true in all our experiments.

map to $\mathcal{P}_0$, the next s/P columns map to $\mathcal{P}_1$, and so on. That is, columns with indices $\{0, 1, 2, \ldots, (s/P - 1)\}$ map to processor $\mathcal{P}_0$, columns with indices $\{(s/P), (s/P)+1, \ldots, 2s/P-1\}$ map to processor $\mathcal{P}_1$, and so on. In general, column j maps to processor $\mathcal{P}_{\lfloor j/(s/P) \rfloor}$. We observe that according to the (s/P)-blocked mapping, processor $\mathcal{P}_i$ owns the ith (s/P)-slab.

Similar to most passes in our implementations, pass 1 of slabpose columnsort proceeds in s/P rounds. In any given round x of this pass, where $0 \leq x < s/P$, a column goes through seven phases. The mapping of columns to processors differs among some phases. When we say that a phase assumes a given column-to-processor mapping $\mathcal{M}$, we mean that at the end of the phase, each column should reside with the processor that owns this column according to the mapping $\mathcal{M}$. Below, we describe the seven phases, specifying the column-to-processor mapping assumed in each individual phase:

- *Read phase* (P-cyclic mapping). Each processor reads in a column. In round x, processor $\mathcal{P}_i$ reads the column numbered $j = xP + i$. After the read phase, the xth P-slab is in the internal memory of the system.

- *First sort phase* (P-cyclic mapping). This phase corresponds to step 1 of slabpose columnsort. Each processor sorts the $r = M/P$ records in its local memory.

- *Communicate phase* (P-cyclic mapping). This phase corresponds to step 2, a P-slabpose, of slabpose columnsort. An all-to-all communication performs the P-slabpose on the xth P-slab. At the end of this phase, each processor has P sorted runs, where each run has r/P records.

- *Second sort phase* (P-cyclic mapping). This phase corresponds to step 3 of slabpose columnsort. Each processor sorts the r records in its local memory. Since the r records are composed of P sorted runs, this step is a recursive merge sort with just $\lg P$ levels of recursion.

- *Shuffle phase* ((s/P)-blocked mapping). This phase corresponds to step 4, a P-shuffle, of slabpose columnsort. Surprisingly, this phase is just a no-op, i.e., each processor does nothing.

Let us see why each processor does nothing in the shuffle phase. By formula (5.1), with $k = P$, $j \bmod P = i$, and $\lfloor j/P \rfloor = x$, column j maps to column $\mathcal{T}_j = is/P + x$. We know that column j maps to processor $\mathcal{P}_i$ according to the P-cyclic mapping, the mapping prior to the shuffle phase. For the shuffle phase to be a no-op, it is enough to show that column $\mathcal{T}_j$ maps to processor $\mathcal{P}_i$ as well, albeit according to the (s/P)-blocked mapping, the mapping that the shuffle phase assumes. By the definition of the (s/P)-blocked mapping, column $\mathcal{T}_j$ maps to the processor numbered $\lfloor \mathcal{T}_j/(s/P) \rfloor = i$. Therefore, according to the (s/P)-blocked mapping, column $\mathcal{T}_j$ maps to processor $\mathcal{P}_i$, thus proving that the shuffle phase is a no-op.

- *Permute phase* ((s/P)-blocked mapping). To execute step 5 of slabpose columnsort, each processor performs an (s/P)-slabpose on one slab. According to the (s/P)-blocked mapping, processor $\mathcal{P}_i$ owns columns with indices $\{i(s/P), i(s/P) + 1, \ldots, (i + 1)(s/P) - 1\}$, i.e., $\mathcal{P}_i$ owns the ith (s/P)-slab. Therefore, each processor owns one of the P (s/P)-slabs, implying that the (s/P)-slabpose requires no communication.

- *Write phase* ((s/P)-blocked mapping). Each processor writes out the r records in its local memory to its disks.

5.2.2 Passes 2 and 3 perform steps 6–11

Here, we describe how the last two passes of subblock columnsort are the same as the last two passes of 3-pass columnsort.

Pass 2 performs steps 6 and 7

In pass 2 of slabpose columnsort, we assume the P-cyclic mapping, just as in pass 2 of 3-pass columnsort. Note that pass 2 of slabpose columnsort and pass 2 of 3-pass columnsort execute the same steps: sort each column and then do a reshape-and-transpose operation on the entire mesh. The mapping prior to step 2 of slabpose columnsort is different from the mapping prior to step 2 of 3-pass columnsort. In slabpose columnsort, since the column-to-processor mapping at the end of pass 1 is (s/P)-blocked, the mapping prior to step 2 is (s/P)-blocked as well.

We claim that pass 2 of slabpose columnsort is identical to pass 2 of 3-pass columnsort, despite the difference in the column-to-processor mappings prior to step 2. In other words, slabpose columnsort can revert back to the P-cyclic mapping from pass 2 onward. Each processor can execute pass 2 as if the mapping has always been P-cyclic.

To prove our claim, we argue the following about slabpose columnsort: step 6 is independent of any column-to-processor mapping, and step 7 is independent of the mapping prior to pass 2. Step 6 just sorts each individual column and therefore is independent of the column-to-processor mapping. In other words, each processor can locally sort the column it has in its local memory without violating the

$$\begin{bmatrix} 1 & 7 & 13 \\ 2 & 8 & 14 \\ 3 & 9 & 15 \\ 4 & 10 & 16 \\ 5 & 11 & 17 \\ 6 & 12 & 18 \end{bmatrix} \xrightarrow{\text{reshape-and-transpose}} \begin{bmatrix} 1 & 3 & 5 \\ 7 & 9 & 11 \\ 13 & 15 & 17 \\ 2 & 4 & 6 \\ 8 & 10 & 12 \\ 14 & 16 & 18 \end{bmatrix}$$

Figure 5.3: The operation of the reshape-and-transpose step of columnsort. For simplicity, we choose this small 6×3 mesh to illustrate the step.

correctness of the algorithm.

At the beginning of step 7, it does not matter which column is stored on which processor. Figure 5.3 illustrates the mechanics of step 7. Let us examine any single column, say column j. Step 7 distributes the r elements in column j among all s columns of the mesh, with r/s elements each. An element in row i of column j maps to column $\lfloor i/r \rfloor$; this target column number depends only on the row number i and not on the column number j. Due to the placement observation of Section 2.2, the placement of an element within a column is immaterial. Therefore, step 7 is independent of the mapping prior to pass 2.

Pass 3 performs steps 8–11

Pass 3 of slabpose columnsort is identical to pass 3 of 3-pass columnsort. Given that the column-to-processor mapping of pass 3 of both algorithms is the same (P-cyclic), and given that steps 8–11 of slabpose columnsort are the same as steps 5–8 of 3-pass columnsort, this observation does not come as a surprise.

5.3 Problem-size bound of the 3-pass implementation

As shown in Section 5.1, as long as k is chosen as a divisor of s, s is a divisor of r, and $r \geq (2s^2/k)(\lceil k^2/s \rceil + 1)$, slabpose columnsort sorts correctly. In our case, since $k = P$, the restriction on the height of the column translates to $r \geq (2s^2/P)(\lceil P^2/s \rceil + 1)$.

We consider two cases:

- Case 1: $P^2 \leq s$, implying that $\lceil P^2/s \rceil + 1 = 2$, which in turn implies that the restriction $r \geq 4s^2/P$ suffices. Letting $r = M/P$ and $s = N/r$, we get the problem-size restriction

$$N \leq (M/P)^{3/2} \frac{\sqrt{P}}{2} . \tag{5.2}$$

  As compared with the problem-size restriction (3.1) of 3-pass columnsort, this problem-size restriction is relaxed by a factor of $\sqrt{P}$. Note that slabpose columnsort does the same amount of I/O and communication as 3-pass columnsort. Therefore, this improvement in problem-size restriction comes at almost no performance cost.

- Case 2: $P^2 > s$, implying that $\lceil P^2/s \rceil \geq 1$ and $\lceil P^2/s \rceil + 1 \leq 2P^2/s$. Here, the restriction $r \geq 4sP$ suffices, since

$$\begin{aligned} r &\geq 4sP \\ &= (2s^2/P)(2P^2/s) \\ &\geq (2s^2/P)(\lceil P^2/s \rceil + 1) . \end{aligned}$$

Letting $r = M/P$ and $s = N/r$, we get the problem-size restriction

$$N \leq \frac{(M/P)^2}{4P} \ . \tag{5.3}$$

Although the exponent of M/P in this restriction is much better than that of restriction (5.2), the maximum problem size decreases as we increase the number of processors. As long as $(M/P)^{1/3}/4^{1/3} \geq P$, however, this limit will be better than that of restriction (5.2). Therefore, we have this crossover point in terms of the number of processors; increasing the number of processors beyond this point makes the problem-size restriction worse than that of restriction (5.2). This crossover point, however, improves with increases in the amount of memory per processor. For example, if $M/P = 2^{21}$, this crossover point is 64 processors, whereas if $M/P = 2^{24}$, this crossover point is 128 processors.

5.4 Implementation notes

Slabpose columnsort differs from 3-pass columnsort only in the first pass. Furthermore, this difference is restricted to the permute phase. In the first pass of 3-pass columnsort, the permute phase rearranges records in the correct order of writing. In the first pass of slabpose columnsort, however, since the shuffle phase is a no-op, the second sort phase can be combined into the permute phase, adding to the complexity of the permute phase.

Chapter 6

Performance Modeling and Lower Bounds

This chapter presents a simple technique for computing a lower bound on the execution time of a given run of our program. Our lower bound is based on the simplistic assumption that there is no interdependence among the I/O, communication, and computation portions of our implementations. This assumption implies that the execution time is dominated by the most heavily used of the three resources: disks, network, and the CPUs. Even when there is no interdependence among various operations, the total time that each resource is in use depends heavily on the I/O, communication, and computation patterns of an implementation; our lower bound takes these patterns into account. We shall see how the paradigm of oblivious algorithms plays a role in turning this rather naive lower bound into a realistic one.

Our main goal behind calculating lower bounds is to establish a framework in which to evaluate the observed performance of our implementations. As we shall see in the next chapter, our five threaded implementations perform as expected, for the most part, relative to one another. This chapter devises a method, however, that allows us to assess the performance of each implementation in isolation. By assessing an implementation, we mean determining whether the implementation actually achieves maximum overlap among I/O, communication, and computation. Once we fix the underlying algorithm and our out-of-core setting, the amount of I/O that an implementation needs to perform is fixed; so are the amounts of communication and computation. Hence, we consider efficient overlapping to be the main factor in determining the quality of our implementations.

We start with a description of Jefferson, our experimental testbed. Next, we discuss initial tuning of some input parameters of our program. We then outline the main issues in modeling the performance of an application such as ours, while formally describing the problem of calculating the most realistic lower bound. We then discuss the limitations of an ideal solution to this problem and describe the solution that we have adopted. Finally, we present the lower bounds on execution times for various combinations of problem size and number of processors. In the next chapter, we compare these lower bounds with the the observed performance, demonstrating that the two are a close match.

6.1 Jefferson

Although the technique presented in this chapter should work for any platform, the results of the technique are not platform-independent. In other words, predictions for one platform do not necessarily translate into predictions for another platform.

Since we apply these techniques to experimental runs on Jefferson, a description of Jefferson is necessary.

Jefferson is a Beowulf cluster of 32 dual 2.8-GHz Intel Xeon nodes. Each node has 4 GB of RAM and an Ultra-320 36-GB hard drive. The nodes run Redhat Linux 8.0 and are connected with a high-speed Myrinet network. We use the C `stdio` interface for disk I/O, the `pthreads` package of Linux, and standard synchronous MPI calls within threads. Just as on Borg, we use the MPI/Pro package for MPI calls.

We make certain assumptions about the resources on each node of Jefferson. Although each node has two CPUs, we run only one process on each node. We do assume, however, that two computation threads within the same process can proceed simultaneously, each running on one of the two CPUs. Also, given the full-duplex and dual-port setup of Myrinet on Jefferson, we assume that there are two communication *channels*. In other words, we assume that two communicate threads can be active simultaneously, each running on one of the two channels. When we say that a thread is active, we mean that it is executing its code, instead of waiting.

6.2 Tuning input parameters

In this section, we review the input parameters of our program. We have developed five implementations: 4-pass columnsort, 3-pass columnsort, subblock columnsort, M-columnsort, and slabpose columnsort. All five implementations are a part of one unified program. In a given run of this program, we choose the desired implementation by way of run-time parameters. We use the word “program” to refer to our single unified program and the word “implementation” to refer to one of the five underlying algorithms.

Throughout this chapter, we assume that all programs are threaded. We also assume that each thread uses exactly one of the three resources: disk, network, and CPU.[1]

We start by listing the parameters for our program. Some of these parameters are independent of our experimental testbed, such as N and P (except that P is at most the number of nodes in our cluster). Some parameters, on the other hand, depend heavily on the testbed; these parameters need to be tuned to achieve the best performance. The last two subsections discuss the tuning of two parameters: the number of buffers and the buffer size.

6.2.1 Input parameters

Any run of our program requires the following input parameters:

- Input file: the name of the file where the initial unsorted data resides.
- Output file: the name of the file where the final sorted data must reside.
- N: the number of records to be sorted.
- Record size: the size of each record, in bytes.
- P: the number of processors.
- Passes: the number of passes, either 3 or 4.

[1] In our program, each thread performs exactly one of I/O, communication, and computation. The underlying implementations of disk I/O and communication, however, might (and almost always do) use the CPU for scheduling or copying data into extra buffers.

- r: the number of records in each buffer. We denote the size of each buffer in bytes by R.
- g: the number of buffers in the global pool.
- Algorithm: one of the five threaded implementations.

6.2.2 Tuning the number of buffers

Here, we present a theoretical upper bound on g, and we compare it to what we saw in our experimental runs.

For good performance, it is important to achieve maximum overlap among I/O, communication, and computation. Therefore, we need to accurately determine g^*, the number of buffers required for maximum overlap.

Our programs rely on multiple buffers. Let us recall how each implementation uses buffers. Our program proceeds in a number of passes over the data. Each pass is composed of a certain number of rounds. Each round, depending on the pass, is a pipeline of a certain number of phases. Each phase performs exactly one of I/O, communication, or computation. Using multiple buffers allows distinct rounds to execute simultaneously, thereby allowing I/O, communication, and computation to overlap.

Our theoretical upper bound is based on the maximum number of threads that can be active simultaneously. Let T be the set of threads in an implementation, and let Q be the set of resources. Furthermore, let T_Q be the a subset of T such that no two threads in T_Q use the same resource. Our theoretical upper bound on g^* is $\left|T_Q\right|$. Since each thread works on exactly one buffer at a time, the value $|T|$ induces an upper bound on g^*. This upper bound, however, can be rather conservative. For

	$\lvert T_Q \rvert$	g^*
4-pass columnsort	3	3
3-pass columnsort	4	4
M-columnsort	5	5,6

Table 6.1: The number of buffers required to achieve maximum overlap for 4-pass columnsort, 3-pass columnsort, and M-columnsort. The theoretical upper bound is $\lvert T_Q \rvert$, the maximum number of threads such that no two of them use the same resource. The rightmost column, g^*, shows the value that corresponded to fastest run times in practice.

example, if each processor had a single disk, and if all threads were I/O threads, then only one thread could be running at any given point in time. Therefore, we need to consider T along with Q, the set of resources. Since $\lvert T_Q \rvert$ is the maximum number of threads that can be active simultaneously, it induces a a tighter upper bound on g^*.

To verify our theoretical upper bound, we ran several experiments with different values of g. Since the thread structure of both slabpose columnsort and subblock columnsort is the same as that of 3-pass columnsort, we ran our experiments for just three implementations: 4-pass columnsort, 3-pass columnsort, and M-columnsort. Table 6.1 shows the theoretical upper bound $\lvert T_Q \rvert$, along with the value g^* that yielded the best results in practice. For each implementation, at least one of the observed values of g^* matches $\lvert T_Q \rvert$.

6.2.3 Tuning the buffer size

Not surprisingly, we found the performance of our program to be sensitive to the buffer size. In each pass, the buffer size determines both the number of rounds and the performance of each round. The larger the buffers, the fewer the rounds, meaning that the program performs fewer disk I/Os, but each disk I/O transfers more data. I/O subsystems usually perform better in this scenario, since transferring more data hides the seek time. After a certain point, however, the buffer size becomes so large that even the communication and computation operations on the buffer start paging, thus far outweighing the gains from better I/O performance. We found that when the record size is not fixed, the performance of our program depends more on the buffer size in bytes (i.e., R), rather than the buffer size in records (i.e., r). We ran several experiments, varying the value of R. The two values of R that gave the best results for all the implementations were the same: 64 MB and 128 MB.

6.3 Modeling the system

In this section, we first describe a way to formally represent a threaded application running in an environment with multiple, yet restricted, resources. Next, we discuss an ideal approach toward calculating a lower bound on the performance, along with the limitations of the approach. Finally, we present our technique; along the way we discuss the simplifying assumptions that we make.

6.3.1 Performance model: The problem statement

Our model is based on two assumptions. First, we assume that it suffices to model the performance of a single processor. Since the I/O and communication patterns of our program are oblivious to the data and are completely balanced across all processors, both the sequence and the type of jobs on all processors are almost identical. Thus, we shall formulate the problem as if there was just one processor, say $\mathcal{P}_i$. We model the existence of multiple processors by accounting for them in terms of the running time of the various communicate phases of $\mathcal{P}_i$. We shall refer to this assumption as the *symmetric-processors assumption*. Appendix A.3 shows that, assuming homogeneous processors with identical job sets, there always exists an optimal schedule of jobs that is also symmetric across processors.

Second, we assume that the first round of a pass starts only after the last round of the previous pass completes. With this assumption, we can focus on the lower bound of a single pass at a time. The final lower bound on performance , therefore, will just be the addition of the lower bounds of the individual passes.

We refer to each phase of each round as one *job*. From here on, we use the terms “phase” and “job” interchangeably. Each processor, therefore, has a set of jobs to finish. Given that these jobs can have sequential dependencies and resource requirements, the goal is to figure out the minimum amount of time required by a processor to complete all the jobs.

Here is a formal specification of our problem, which we shall refer to as the *optimum-performance problem*.

Input

- *Jobs*. Each processor has a set $J = \{J_1, J_2, \ldots, J_n\}$ of n jobs. Let T_i be the units of time that it takes to complete job J_i. If J_i starts at time t, it finishes at time $t + T_i$.

- *Interpretation of T_i for a communication job J_i*. Communication jobs require participation of multiple processors. For a communication job J_i, let P' be the set of processors involved. By T_i, we mean the time to complete J_i, given that all the processors in P' start the communication job J_i at the same time.[2]

- *Resources*. Each processor has a set $Q = \{q_1, q_2, \ldots, q_m\}$ of m resources. Each job uses a single resource from the set of m resources. We use the term Q_i to refer to the resource required by J_i.

- *Sequentiality constraints*. We have a list *Seq* of pairs of jobs, such that if $(J_i, J_k) \in Seq$, it means that job J_i must finish before job J_k can be started.

- *Resource constraints*. If $Q_i = Q_k$, then jobs J_i and J_k must not overlap.

Output

Given the set J of jobs, along with their execution times and resource requirements, the solution is the minimum time, say B^*, required to complete all jobs, without violating the two constraints. We assume that the start time of a job can be as early as $t = 0$.

6.3.2 The ideal solution

As it turns out, the optimum-performance problem is NP-hard [GJ79, ABP98]. In fact, it is well known in the scheduling research community as the $P \mid prec \mid C_{\max}$ problem.

Our goal is to compute a lower bound on B^*. Removing both constraints of sequentiality and resource usage results in a trivial but unrealistic lower bound: schedule all jobs at $t = 0$. Instead, we consider the lower bound obtained by

[2]By the symmetric-processors assumption, each processor schedules its jobs similarly, implying that J_i will be started at the same time on all processors in P'.

removing only the sequentiality constraints. The next subsection describes our algorithm.

6.3.3 Our lower-bound algorithm

Our algorithm assumes that there are no sequentiality constraints. This assumption implies that we can schedule the jobs on each resource independently of every other resource. Given this assumption, along with the input parameters to the optimum-performance problem, our algorithm calculates B^*, the minimum time, as follows:

1. For each resource q_i, calculate $B^*_{q_i}$, the total completion time for all the jobs in U_{q_i}, as
$$B^*_{q_i} = \sum_{J_k \in U_{q_i}} T_k \ .$$
2. Calculate B^* as
$$B^* = \max(B^*_{q_1}, B^*_{q_2}, \ldots, B^*_{q_m}) \ .$$
Therefore, whichever resource takes the longest to complete its set of jobs dictates the total running time of the pass.

For an algorithm with multiple passes, we calculate B^* for each pass, and then we add them all up to get the total running time for the program.

6.4 Our lower bound results

This section describes the process of actually computing the lower bound on the running time for a given combination of the problem size, the number of processors, and the algorithm. We first describe how we instantiate the optimum-performance problem with our out-of-core sorting program and the cluster environment. Next, we outline how to determine the total completion time for each individual resource. Finally, we use our algorithm to calculate lower bounds on the running times of several combinations of problem size and number of processors, while analyzing the performance trends that emerge.

6.4.1 Instantiating the problem

The instantiation problem is to create an instance of the optimum-performance problem, given the cluster environment and the input parameters of our out-of-core sorting program. The cluster environment determines the resources and the completion time for any given job. The input parameters, such as N, R, and the implementation (slabpose columnsort, for example), determine the set of jobs. The challenge of the instantiation problem is to come up with the T_i values, i.e., to determine the completion time for each job J_i. As before, each phase of each round is a job, requiring us to know beforehand how long each phase takes. Let us examine the typical size of the set of jobs. Most passes have s/P rounds, and each round has between 5 and 11 phases, leading to at least $5s/P$ jobs per pass. Typical values of s/P range from 256 to 2048, implying that a given pass can have as many as 10240 jobs.

To avoid having to figure out how long each job takes, we make the following assumption: all rounds within a pass are identical. That is, phase 1 of round 1 takes the same amount of time as phase 1 of any other round, and similarly for all phases. This assumption greatly reduces the number of distinct jobs in the set of jobs. For example, for a pass in which each of the s/P rounds has 5 phases, the number of distinct jobs reduces to 5. In our program, this assumption actually holds for all but the in-core sort phases. Since the I/O and communication steps of our algorithm are oblivious to the data, all I/O and the communicate phases do the same amount of work in each round. The in-core sorting phases, however, can differ across rounds. In each round of each pass, both the amount of sortedness in the buffer and the in-core sorting mechanism used to sort the buffer are identical. The actual key values can still potentially affect the running time, however. We shall describe later how we estimate the running times for the various phases.

Given the problem size N, the number of processors P, the algorithm A, and a pass number p, we perform the following steps to construct an instance of the optimum-performance problem:

- Set the values for g and R, using the results of the tuning methods described in Sections 6.2.2 and 6.2.3.

- Calculate r and s. Once we know what R is, we get r, the buffer size in records, by dividing R by the record size in bytes. Consequently, $s = N/r$.

- Determine the resources. There are 5 resources: one disk, two network channels, and two CPUs.

- Determine the jobs, i.e., the set J. Since each phase is one job, we can talk about jobs in terms of phases. Let there be k phases (or jobs) $J_1, J_2, \ldots, J_k$ in each round of pass number p. Since all rounds are identical, there are s/P copies of each of these k jobs.

- Determine the resource usage Q_i for each job J_i. Since each phase uses exactly one resource, there is a mapping of phases to resources. In our program, we map the read and the write phases to the one disk. In all algorithms but M-columnsort, we have at most two communicate phases; we map one to each of the channels. Pass 3 of M-columnsort has four communicate phases; we map two to each channel. The mapping of compute (i.e., sort and permute) phases happens similarly. When there are multiple sort phases, we divide them equally among the two CPUs.[3] Since all rounds are identical, all the copies of a job are mapped to the same resource.

- Estimate T_i for each job J_i. Estimating the T_i values is the least straightforward part of this instantiation process. The next subsection describes how to estimate the T_i values for I/O, communication, and computation jobs.

6.4.2 Estimating the T_i values: Exploratory runs

For a given pass, we estimate the T_i values, the times per phase, by first conducting some exploratory runs of our program. Besides achieving its main goal of estimating the T_i values, a hidden goal of this step is to keep the number of exploratory runs low. We can always perform one exploratory run for every real run of the program, which is clearly redundant. For each type of job—namely I/O, communication, and computation—we have identified the main parameters of our out-of-core sorting program that cause the completion time to vary significantly. We call such parameters *critical*. For example, an obvious critical parameter for a communication job is P, the number of processors, since a communication phase involving 16 processors typically takes more time than a communication phase involving fewer than 16 processors. For any job, the number of critical parameters

[3]Since a typical sort phase takes much longer than a typical permute phase, we make sure that each CPU gets an approximately equal share of both sort and permute phases. We need to make this assumption in order to use the result of this analysis as a lower bound.

dictates the number of exploratory runs required for estimation.

Estimating I/O times

We considered just one critical parameter of I/O times: per-processor problem size in bytes. By per-processor problem size, we mean the total amount of data per processor, in bytes, i.e., $(N/P)x$, where x is the record size in bytes. Due to disk-space limitations, not all per-processor problem sizes were possible. Our experiments cover only two per-processor problem sizes: 4 GB and 8 GB.

To estimate I/O times, we ran separate I/O tests that read and write a file of a given size in almost the same pattern as our program. We measured the time for a few passes, and we extrapolated the per-pass I/O time from the measured time. To calculate the per-phase I/O time, we divide the per-pass I/O time by the number of rounds.

Estimating communication and computation times

For estimating both communication times and computation times, we ran a few experiments, varying the critical parameters of both communication and computation. In each exploratory run, we fix the parameter g to be 1 and augment the program with some profiling code. Notice that with $g = 1$, at most one thread can be active at any time, and it runs without preemption. Hence, having just one buffer sequentializes the threads. The profiling code measures the per-pass time that each thread considers itself active. In each exploratory run, therefore, each thread t_i records a single value per pass: the total time that it considered itself active. To get the per-phase time for each thread, we just divide the total time by the number of rounds. Again, we note that this per-phase time will be parameterized by the critical parameters. For example, one completion time for a communicate phase was the following:

> A pairwise communication involving 8 processors, with a buffer size of 128 MB takes 1.55 seconds.

We considered three parameters to be critical for a communication operation:

- Communication type. Each communication phase of our program is one of the following two types: *pairwise* or *all-to-all*. In a pairwise communication, each processor exchanges records with one other processor. In an all-to-all communication, each processor exchanges records with every other processor. The timings for the two types are not only different, they scale differently with an increase in the number of processors.
- R, the buffer size. Obviously, the size of the buffer will dictate how long each communication phase takes.
- P, the number of processors. Typically, increasing the number of processors increases the communication time, for a given buffer size.

Computation phases include both sorting and permutation phases. Here is a list of the critical parameters for computation:

- A, the algorithm, and p, the pass number. We recall that some of the sorting phases receive data that is completely unsorted, whereas some receive data that is partially sorted; the extent of unsortedness depends on A and p. Not surprisingly, the extent of unsortedness affects the time to sort the data.

- N, the problem size. The problem size does affect the computation time, since, as discussed in Section 3.1, the effect of the partial-sorting optimization diminishes with an increase in problem size.

- R, the buffer size. Not surprisingly, the amount of data being sorted or permuted affects the times it takes to sort or permute it.

We performed a few exploratory runs to estimate the communication and computation times. We did not write any extra code for the exploratory runs, aside from the profiling code. Our exploratory runs were just normal runs of our program, with $g = 1$, and with the profiling code switched on.

6.4.3 Predicted running times of our program

We predict the running times of our program as follows:

- Given the input parameters to our out-of-core sorting program, we estimate I/O, communication, and computation times.

- We then calculate the total execution time per resource, per pass, using our estimated T_i values.

- We compute the total per-pass completion time as the longest time taken by any resource.

- The total execution time for the program is then just the combined total of the times of all the passes.

For each of the five implementations, we predict the running times for various combinations of the number of processors and the problem sizes. Across all our experiments, there are only two values for per-processor problem size: 4 GB and 8 GB. In the rest of the thesis, we shall refer to these two scenarios as the *4-GB case*, and the *8-GB case*, respectively. Since each node has 4 GB of memory, we did not consider per-processor problem sizes less than 4 GB. As mentioned before, we could not run experiments with more than 8 GB of data per processor due to disk-space limitations.

We present our predicted execution times in two sets, one for each of the two per-processor problem sizes. Due to disk-caching effects, we believe that the 8-GB case is more representative of an out-of-core sort than the 4-GB case.

4 GB of data per processor

Figure 6.1 shows the predicted lower bounds for our five threaded implementations, for several problem sizes that all have 4 GB of data per processor. For each problem size in GB, the number of processors is just the problem size in GB, divided by 4. The figure also shows two straight lines, representing baseline I/O times for three and four passes.

The timings for 3-pass columnsort and slabpose columnsort are very close to the 3-pass baseline I/O time, suggesting that these two algorithms are fairly I/O bound. Similarly, the timings for 4-pass columnsort and slabpose columnsort, both with 4 passes, are very close to the 4-pass baseline I/O time.

The only exception, and a major one, to the I/O-boundedness is M-columnsort. Even though M-columnsort has just 3 passes, it performs substantially more communication and computation than 3-pass columnsort. The timings of M-columnsort suggest that 3-pass columnsort, the predecessor of M-columnsort, must

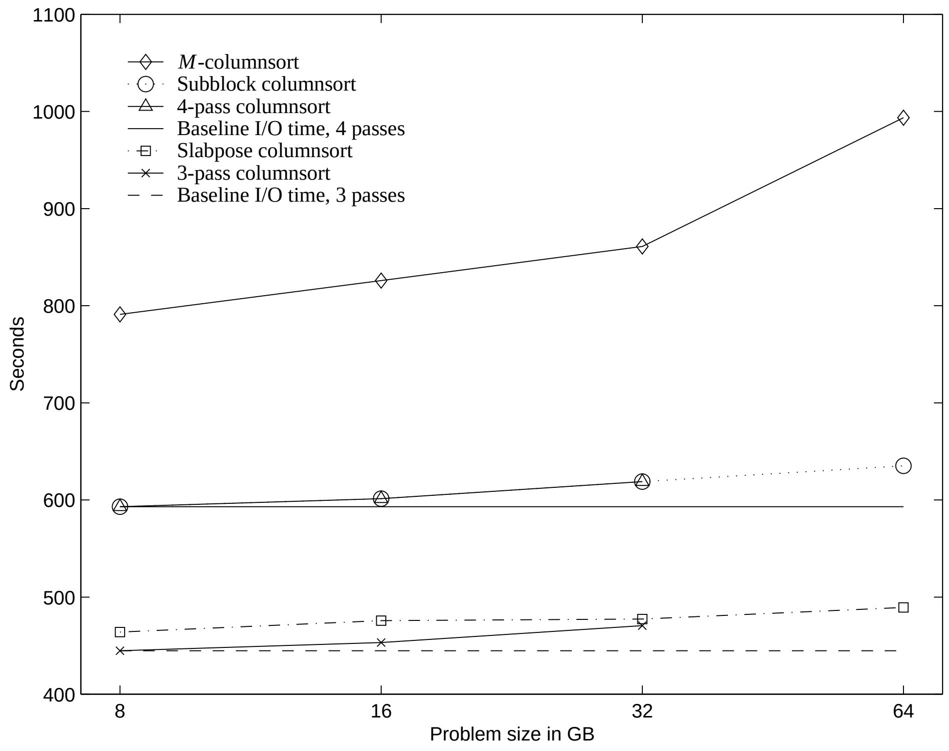

Figure 6.1: Predicted lower bounds on completion times, with 4 GB as the per-processor problem size. The times are on the vertical axis, in seconds. For each of subblock columnsort, slabpose columnsort, and M-columnsort, the figure shows completion times for four problem sizes. For 4-pass and 3-pass columnsort, however, a problem size of more than 32 GB was not possible, due to the problem-size restriction. The two straight lines represent the baseline I/O times for three and four passes.

not be I/O-bounded enough to hide the extra communication and computation.[4]

The figure also shows some scalability issues. As the problem size (and therefore the number of processors) increases, the predicted time goes up. This increase does not come as a surprise since communication times increase with the number

[4]In fact, when we looked at the total disk time for 3-pass columnsort, it was quite close to the total communication time for the passes where the communication type is all-to-all. Furthermore, the total computation time for the first pass (where the data is completely unsorted) was almost as much as the total I/O time for the first pass, suggesting that 3-pass columnsort is not I/O-bound by a large amount.

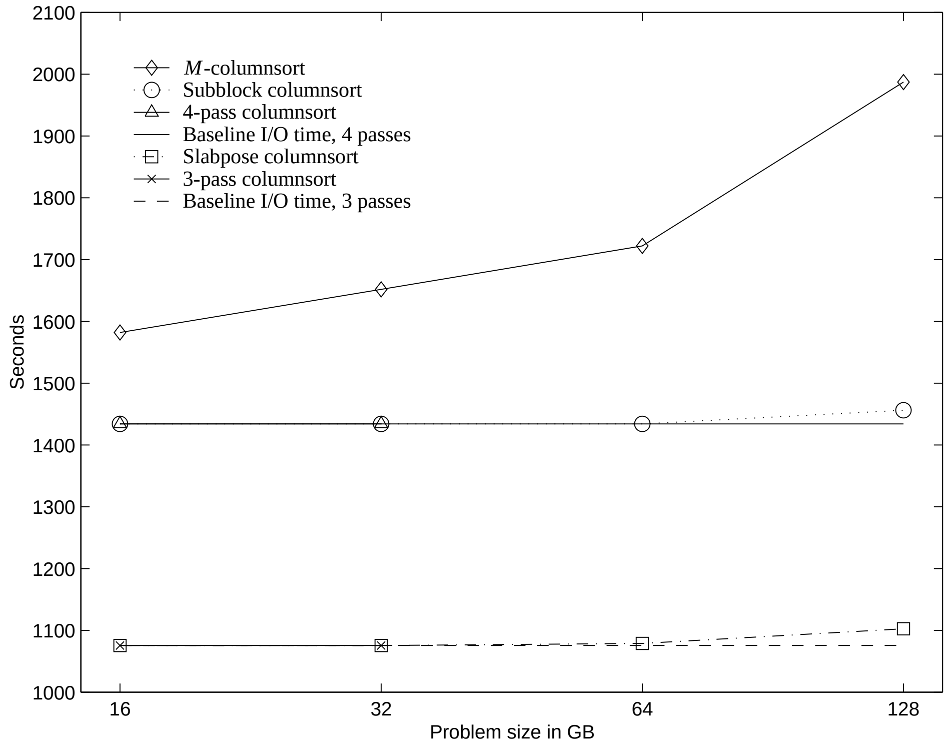

Figure 6.2: Lower bounds on completion times, with 8 GB as the per-processor problem size. The times are on the vertical axis, in seconds. For each of subblock columnsort, slabpose columnsort, and M-columnsort, the fi gure shows completion times for four problem sizes. For 4-pass and 3-pass columnsort, however, only two problem sizes, 16 and 32 GB, were possible. The two straight lines represent the baseline I/O times for three and four passes.

of processors, and the computation times increase with the problem size (due to the partial-sorting optimization).

8 GB data per processor

Figure 6.2 shows the lower bounds for our five threaded implementations, for several problem sizes that all have 8 GB of data per processor. For each problem size in GB, the number of processors is just the problem size in GB, divided by 8. As in the 4-GB case, there are two straight lines, representing baseline I/O times for three and four passes.

Except for the increased extent of I/O boundedness, the trends in the 8-GB case are the same as those in the 4-GB case. Since the 8-GB case is more I/O bound, the increase in execution times, as the problem size increases, is less pronounced.

Chapter 7

Performance Results on Jefferson

This chapter presents the results of our experiments on Jefferson. After outlining our experimental setup, we analyze the performance of our five threaded implementations. We conclude with a comparison of our results with the lower bounds calculated in Chapter 6.

7.1 Experimental setup

Our experimental runs were for combinations of the following:

Algorithm: We ran 4-pass columnsort, 3-pass columnsort, subblock columnsort, slabpose columnsort, and M-columnsort. For a baseline, we also ran just the I/O portions of three and four passes of columnsort. Recall that 4-pass columnsort and subblock columnsort make 4 passes over the data, while subblock columnsort, M-columnsort, and 3-pass columnsort make 3 passes over the data.

Buffer size: For all our implementations except for subblock columnsort, the buffer size used is 64 MB. For subblock columnsort, since s has to be a perfect square, some problem sizes were impossible using 64 MB buffers; we used 128 MB buffers in such cases. Note that these buffer sizes, being in bytes, are not in terms of number of records, and so they should not be construed as equaling M/P. The record size in all experiments is 64 bytes.

Number of processors and volume of data: We ran various combinations with 2, 4, 8, and 16 processors and with an amount of data varying from 8 GB up to 128 GB. We did not run any experiments with less than 4 GB of data per processor because file-caching effects masked the out-of-core nature of the problem. We were unable to perform any runs with more than 8 GB of data per processor due to disk-space limitations.

7.2 Relative performance of the five threaded implementations

In this section, we review various aspects of the observed performance of our five implementations. We shall first examine how I/O bounded each implementation was. Then, we compare the performance of the 4-pass and the 3-pass implementations of columnsort. Finally, we analyze the running times of our three algorithms that relax the problem-size restriction: slabpose columnsort, subblock columnsort, and M-columnsort.

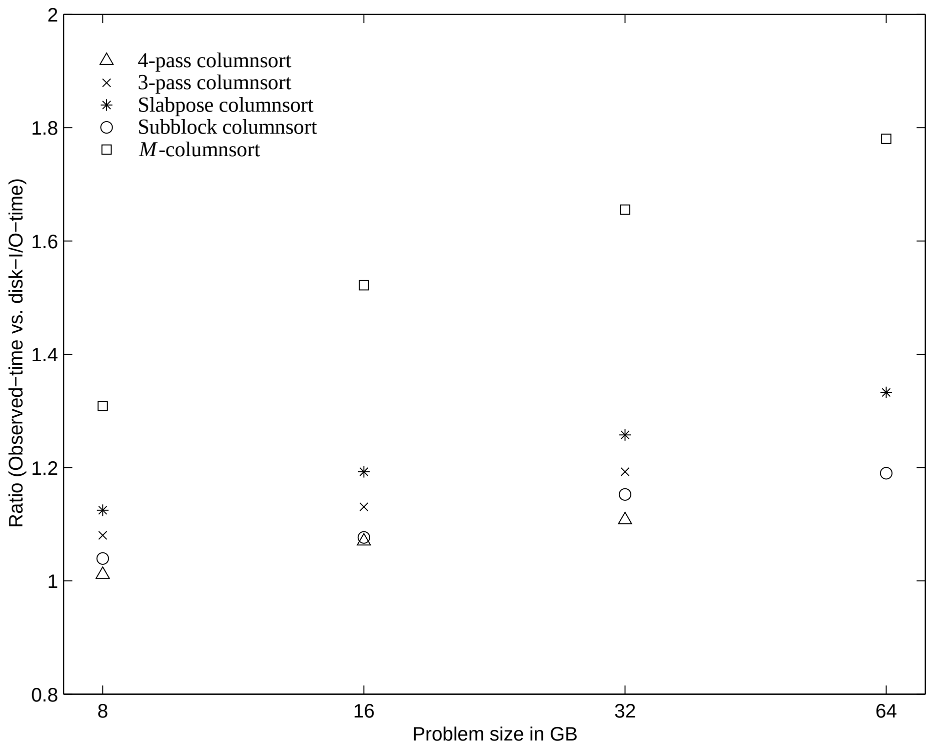

Figure 7.1: Ratio of observed running time vs. the baseline I/O time for the five threaded implementations with 4 GB of data per processor. The ratio (observed-time/I/O-time) is on the vertical axis.

I/O boundedness

We ran our experiments in two sets. In the first set, the per-processor problem size was 4 GB; in the second set, it was 8 GB. For a given algorithm and buffer size, we found that the amount of data per processor was by far the most important factor in determining run time. Given the large amount of disk I/O that each of the algorithms has to perform, this characteristic did not come as a surprise. M-columnsort, however, was an exception to this rule. In M-columnsort, the substantial amounts of communication and computation exceeded the I/O times by far.

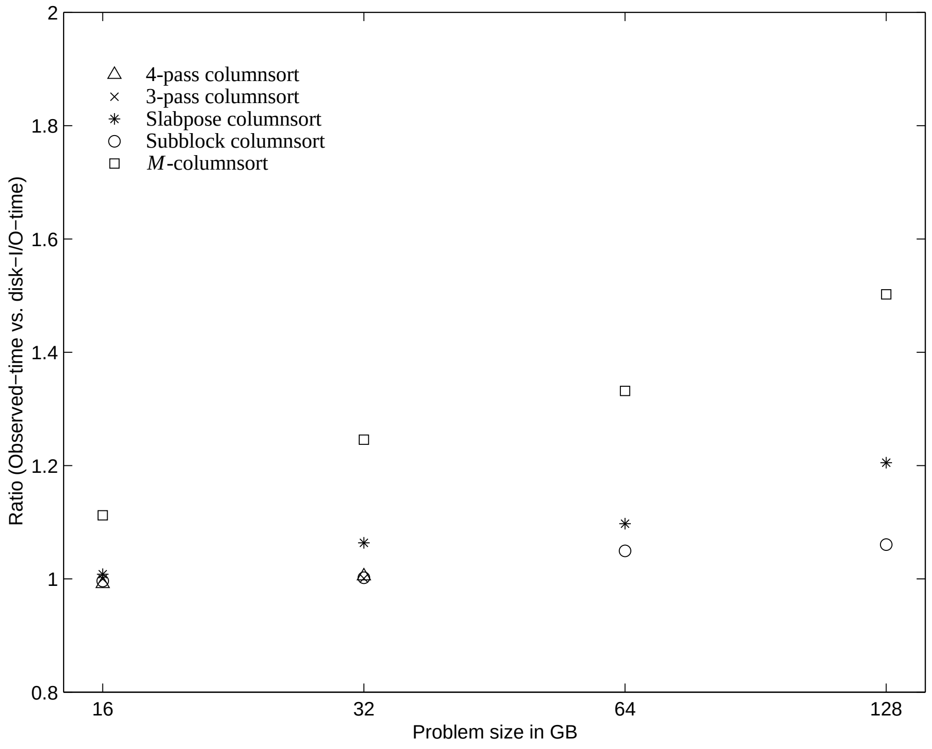

Figure 7.2: Ratio of observed running time vs. the baseline I/O time for the five threaded implementations with 8 GB of data per processor. The ratio (observed-time/I/O-time) is on the vertical axis.

Figures 7.1 and 7.2 show the ratios of total running times and the baseline I/O times for the 4-GB case and the 8-GB case, respectively. For the 4-GB case, the maximum value for this ratio was 1.78, and the average value was 1.16. For the 8-GB case, the maximum value was 1.5, and the average was 1.07. The maximum values, in both cases, are due to M-columnsort, which is clearly not I/O bound.

Excluding M-columnsort, the maximum values of these ratios are 1.33 and 1.20 for the 4-GB and the 8-GB cases, respectively; the average values are 1.09

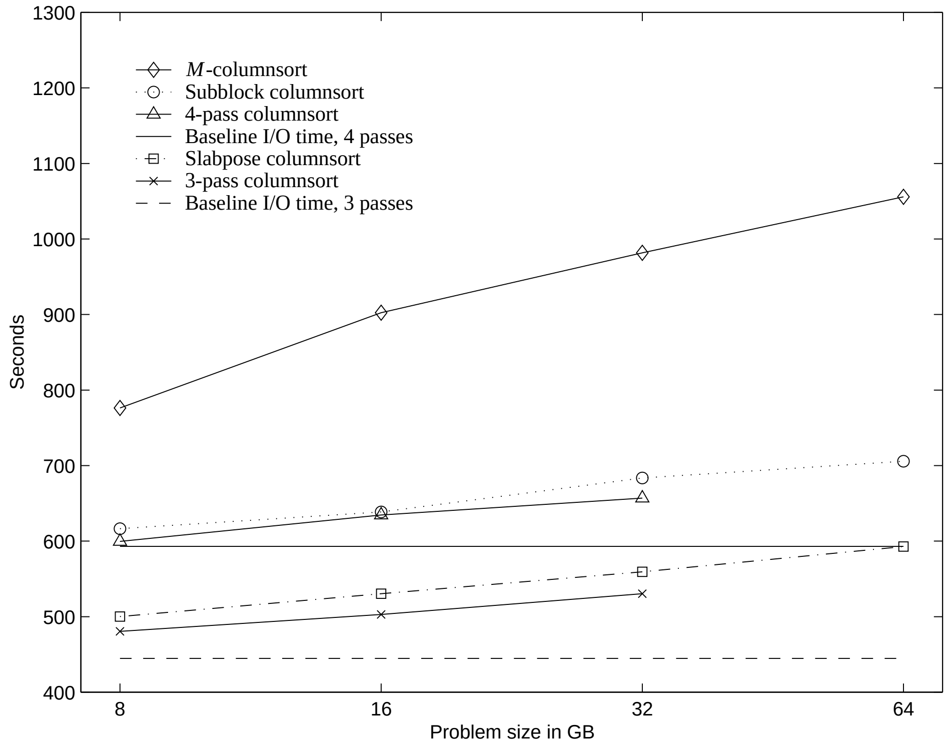

Figure 7.3: Actual completion times, with 4 GB as the per-processor problem size. The times are on the vertical axis, in seconds. For each of subblock columnsort, slabpose columnsort, and M-columnsort, the figure shows completion times for four problem sizes. For 4-pass and 3-pass columnsort, however, only problem sizes up to 32 GB were possible, due to the problem-size restriction. The two straight lines represent the baseline I/O times for three and four passes.

and 1.02. These average values show that, but for M-columnsort, all our implementations are fairly I/O-bound. Furthermore, the average as well as the maximum values are lower for the 8-GB case, thus substantiating our claim that the 8-GB case is more representative of an out-of-core scenario.

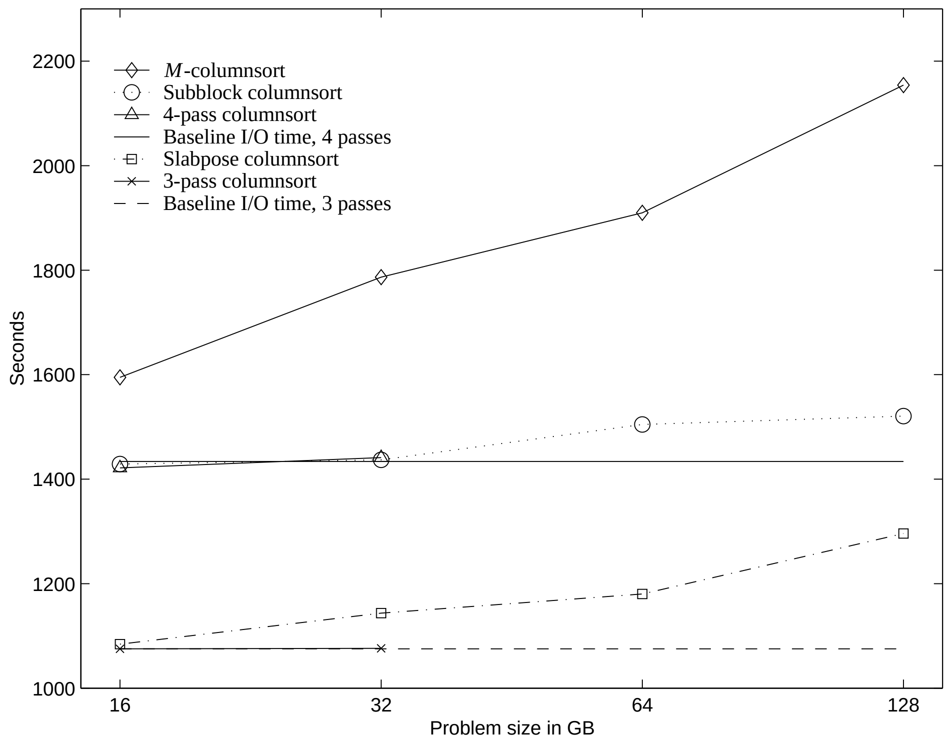

Figure 7.4: Actual completion times, with 8 GB as the per-processor problem size. The times are on the vertical axis, in seconds. For each of subblock columnsort, slabpose columnsort, and M-columnsort, the figure shows completion times for four problem sizes. For 4-pass and 3-pass columnsort, however, only problem sizes up to 32 GB were possible, due to the problem-size restriction. The two straight lines represent the baseline I/O times for three and four passes.

The observed running times

Figure 7.3 shows the running times of our experiments with 4 GB of data per processor; Figure 7.4 covers the experiments with 8 GB of data per processor. Each plotted point in the figure represents the average of multiple runs of an algorithm. Variations in running times were relatively small (within 5%). The horizontal axis is organized by the total amount of data, in GB, sorted across all processors.

Due to the problem-size restriction (3.1), 4-pass columnsort and 3-pass columnsort could not handle more than 32 GB of data. The problem-size restrictions of subblock columnsort, slabpose columnsort, and M-columnsort allowed all the problem sizes that we could sort, given the disk-space limitations.

For the problem sizes of 16 GB and 32 GB, both slabpose columnsort and 4-pass columnsort are almost below the baseline 4-pass I/O time. It might seem like an anomaly that an implementation takes less time than the baseline I/O time. This difference in timings, however, is not much of an anomaly since the presented timings are averages of multiple runs.

4-pass columnsort and 3-pass columnsort

For both the 4-GB and 8-GB cases, 3-pass columnsort runs in approximately 75% of the time taken by 4-pass columnsort. On Sun-A and Sun-B, 3-pass columnsort was only 5–9% faster than 4-pass columnsort, substantiating our claim that 3-pass columnsort would show more of an improvement on recent, more I/O bound systems. Recall that in the last pass of 3-pass columnsort, there are two communicate phases, two sort phases, one permute phase, one read phase, and one write phase. Since 3-pass columnsort is about 25% faster than 4-pass columnsort, we can deduce that up to two communicate phases and up to three compute phases can be hidden behind the disk I/O phases. Given that each node on Jefferson has two CPUs and two communication channels, this deduction does not come as a surprise.

Slabpose columnsort

Slabpose columnsort performs similarly to 3-pass columnsort. The performance of slabpose columnsort degrades as the problem size increases. The good news is that this degradation is less pronounced in the 8-GB case than in the 4-GB case; we can expect that for per-processor problem sizes that are prominently out-of-core, slabpose columnsort would scale well. Since slabpose columnsort relaxes the problem-size restriction with almost no extra cost (it has one extra sort phase), we expect slabpose columnsort to be the algorithm of choice for problem sizes that 3-pass columnsort cannot handle. Our experiments show that, for problem sizes above 32 GB, slabpose columnsort is the clear winner.

Subblock columnsort

Subblock columnsort performs similarly to 4-pass columnsort. In both the 4-GB and 8-GB cases, the performance of subblock columnsort scales well with the problem size. Hence, subblock columnsort scales slightly better than slabpose columnsort. We believe that there are two reasons for this difference in scalability. First, for most cases ($s \geq P^2$), the subblock pass does no communication. Second, the first pass of slabpose columnsort has two sort phases, compared to the single sort phase of the first pass of subblock columnsort. The scenario in which we might consider using subblock columnsort instead of slabpose columnsort would be a system in which I/O is much cheaper than communication and computation.

M-columnsort

For all problem sizes, and all across our experiments, M-columnsort performs the poorest. Recall that passes 1 and 2 of M-columnsort have a total of 11 phases, whereas pass 3 has a total of 20 phases. Each pass, however, has only two I/O phases: the read phase and the write phase. Jefferson is not nearly as I/O bound to hide so many communication and computation times behind disk I/O. The in-

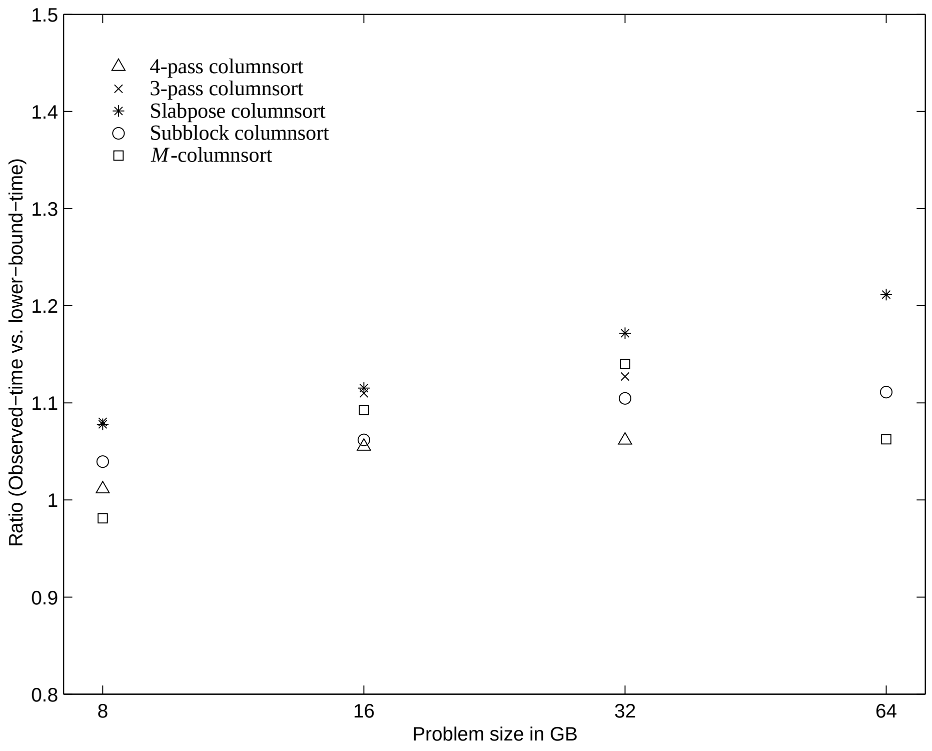

Figure 7.5: Ratio of observed running time vs. the lower bounds on running times for the five threaded implementations with 4 GB of data per processor. The ratio (observed-time/lower-bound) is on the vertical axis.

creased number of phases in M-columnsort is due to the column being M records tall, thus introducing a parallel in-core sort step in each round. For this parallel sorting, we use an in-core parallel version of columnsort. In future, we would like to explore a different sorting algorithm for this purpose, in particular, one that does not have as many computation and communication phases.

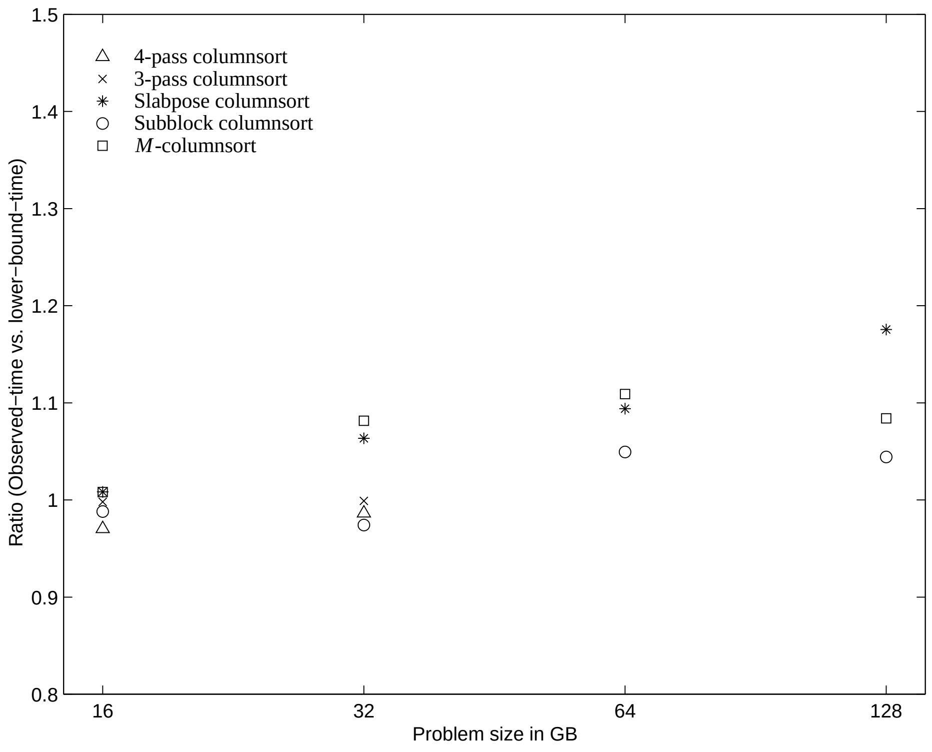

Figure 7.6: Ratio of observed running time vs. the lower bounds on running times for the five threaded implementations with 8 GB of data per processor. The ratio (observed-time/lower-bound) is on the vertical axis.

7.3 Comparison with the lower bounds

In this section, we evaluate how well the performance of our out-of-core sorting program matches the lower bounds on running times from Chapter 6.

For a given problem size N and a given number of processors P, we divide the real observed running time by the lower bound for the same problem size. We examine this ratio to evaluate how far off the real running times are.

Figures 7.5 and 7.6 show the ratios of real running time and the lower bounds on running times for the 4-GB case and the 8-GB case, respectively. For the 4-GB case, the maximum value for this ratio was 1.21, and the average value was 1.08. For the 8-GB case, the maximum value was 1.20, and the average was 1.04. On average, therefore, our programs perform almost as well as the lower bounds.

We believe that the main reason for this encouraging result is that most of the assumptions that we made in the course of our lower-bound calculation process are true in our implementations. Of course, it is not true that there are no sequentiality constraints on jobs. However, if all rounds are identical, if all processors have identical job pools, and if there is at least one buffer per resource, the following might happen: the one resource that takes the longest to complete its jobs is always busy. If the slowest resource is always busy, it is no surprise that the total running time is dictated by the completion time of this slow resource. Let us consider a simple scenario in which each round has k phases, both the first and the last phases are disk-I/O phases, and the disk is the slowest resource (so that each disk I/O phase takes longer to complete than any of the other phases). Let T_i be the time for a single buffer to go through the ith phase. The disk starts by reading data into g buffers, taking a total of gT_1 time. How long does it take for the first buffer to go through the $k-2$ non-I/O phases? It takes $T_2 + T_3 + \cdots + T_{k-1}$ units of time. Since the disk is the slowest resource, we have $T_i \leq T_1$ for all i, implying that $T_2 + T_3 + \cdots + T_{k-1} \leq (k-2)T_1$. If $g = k$, then $T_2 + T_3 + \cdots + T_{k-1} \leq (g-2)T_1$, implying that by the time the disk is done reading all the g buffers, the first buffer is ready to be written out, thereby keeping the disk busy. Note that this scenario will continue throughout the rounds because all rounds are identical.

Chapter 8

Related Work

For a problem as fundamental as sorting, discussing all the work that might qualify as related is beyond the scope of this thesis (Knuth [Knu98, 379–388] gives a thorough summary of the early developments in both in-core and out-of-core sorting). In this chapter, we discuss some work that has been done in the area of out-of-core sorting algorithms

We start by describing the two dominant paradigms of out-of-core sorting: merging-based algorithms and partitioning-based algorithms. For each paradigm, we first outline the major design issues and theoretical results, and then we discuss the algorithms that actually have been implemented. Since we are targeting the cluster architecture, the implementations that interest us the most are the ones that are both multiprocessor and out-of-core for a cluster. We know of only three such implementations, and we discuss them in Section 8.2.

Merging and distribution are not the only paradigms for sorting massive data [PB93]. Radix sort is one example of a sorting algorithm that does not fit either of these paradigms, and it has been implemented for the uniprocessor out-of-core setting [CH97].

Several papers discuss design and experimental results for parallel, but in-core, sorting algorithms. Blelloch et al. [BLM$^+$98] present a novel comparison of a parallel version of radix sort [Knu98, Section 5.2.5], bitonic sort [Bat68], and a sample sort based on flashsort [RV83]; among their conclusions is, that sample sort is the fastest of the three if there are a large number of keys per processor. Helman et al. [HJB98] analyze the performance of a variation of sample sort on various shared-memory machines such as the IBM SP-2-WIN and Thinking Machines CM-5; they show that their algorithm performs well for several input distributions. In [HBJ98], Helman et al. present a deterministic variant of the randomized algorithm of [HJB98]. They show that the deterministic algorithm performs almost as well as the randomized one, while giving deterministic guarantees on performance as well as memory requirements.

8.1 Merging-based algorithms

The basic idea behind the paradigm of merging is to first create sorted runs of the input (the run formation step) and then merge these sorted runs (the merge step). In an out-of-core setting, each sorted run is usually the size of one *local memoryload*: the amount of data that can be held in the memory of one processor.

Design issues

When designing a merging-based, out-of-core, sorting algorithm, the primary challenge is to ensure good data locality in the merge step. Even in the uniprocessor case, it is not possible to fit all the runs being merged in memory. A typical merge step proceeds by reading into memory a few disk blocks worth of data from each of the runs that need to be merged. For I/O efficiency, these blocks must be evenly distributed across the various disks, when both reading and writing. It is tricky to predict which disk blocks are going to be needed next, because that depends on the rate at which the various runs are being consumed to produce the output; this rate, in turn, depends on the keys being merged and cannot be known a priori. Therefore, if implemented naively, algorithms using the merging paradigm can suffer from I/O bottlenecks.

Theoretical work

Several researches have tackled the challenge of assigning blocks to disks. Greed sort by Nodine and Vitter [NV95], (l, m)-merge sort by Rajasekaran [Raj01], Sharesort by Aggrawal and Vitter [AP94], and Simple Randomized Mergesort by Vitter and Barve [BGV96] are a few examples of merging-based, out-of-core, sorting algorithms.

Hutchinson et al. [HSV01a] define a novel duality between scheduling online writes to parallel disks and scheduling offline prefetching read requests to parallel disks. This duality facilitates designing out-of-core algorithms for independent parallel disks. They apply this duality to devise two algorithms: results for online writing translate into RCD+, a partitioning-based sorting algorithm, and results for offline reading translate into RCM, a merging-based sorting algorithm. Vitter and Hutchinson [VH01] use a randomized allocation scheme for parallel independent disks to design a randomized partitioning-based sorting algorithm, RCD, a variant of RCD+. The algorithms and the analyses in both [HSV01a, VH01] assume a uniprocessor, multiple-disk setting.

Zhang and Larson present some strategies for reducing I/O time during the merge phase in external mergesort. They discuss run-time adjustment of work space to changes in available memory, leading to a memory-adaptive merge sort in [ZL97]. They also propose new caching and sequential read-ahead policies for reducing I/O time during the merge phase of external mergesort in [ZL98]. Again, a uniprocessor setting is assumed in these two papers.

Pai et al. [PSV94] explore an interesting systems aspect of external merge sort: the tradeoff between disk parallelism and cache size. In particular, they compare two prefetching strategies, one greedy and the other conservative, by developing a Markov model for each. They derive a closed-form expression for the average number of disk blocks read in one I/O operation as a function of the cache size and the number of runs. They conclude that the conservative strategy attains slightly more parallelism than the greedy one, for what they consider as the most reasonable choices for the number of disks and the cache size. They also present simulation experiments to validate their predictions.

Implementations

In the merging paradigm, the only out-of-core implementations that we know of are the implementations of Rajasekaran's (l, m)-merge sort [Pea99, RJ00] and some recent implementations based on Simple Randomized Mergesort by Vitter and Barve [BGV96, BV02, DS03]. All these implementations, though uniprocessor, are instantiations of algorithms designed for the PDM. Alphasort [NBC$^+$95] is a single-processor, 2-pass algorithm that uses quicksort to generate runs and then uses replacement selection to merge them. The experimental results are for data that almost fits in-core (sorting 1.08 GB with 1.25 GB of main memory).

8.2 Partitioning-based algorithms

The basic idea behind the paradigm of partitioning is to first partition the input into k sets $S_1, S_2, \ldots, S_k$, such that each element in S_i is less than or equal to all the elements in S_{i+1}; this step is known as the distribution phase. The sort phase then produces the sorted output $T_1, T_2, \ldots, T_k$, where T_i is the sorted version of S_i. In an out-of-core setting, k is usually equal to the number of processors in the system. In a typical partitioning-based out-of-core sorting algorithm, after the distribution phase, each processor has to sort its partition, and the problem reduces to external sorting in a uniprocessor system.

Design issues

When designing a partitioning-based sorting algorithm, the principal challenge is to ensure that the amount of data in each partition is approximately equal. If not, the uneven partitioning leads to a poor utilization of both computing power and I/O. A bad key distribution can foil almost any technique of obtaining an even distribution. One obvious way to partition would be to find what are called "exact splitters," i.e., $k - 1$ partition elements such that each partition is of size N/k, where N is the size of the input. It turns out that it is difficult to find the exact splitters in an I/O-efficient way [Vit01]. Instead of finding exact splitters for the entire input, most algorithms find splitters based on a sample of the input file.

Theoretical work

We discuss two generic sampling methods:

- *Deterministic sampling* is the method of taking the samples and finding the splitters in a deterministic manner. Several algorithms use this method [AV88, BHJ96, HJB98, NV93]. These algorithms have provable bounds on how unbalanced the sizes of the partitions can be. These methods, however, are very complicated and quite challenging to parallelize. To date, we do not know of any out-of-core sorting implementation for clusters that uses any of these methods.

- *Probabilistic sampling* is the method of taking the samples and finding the splitters in a randomized manner. Again, several algorithms use a probabilistic sampling method [HBJ98, DNS91, HJ97, VS94]. These algorithms tend to be easier to implement than the deterministic ones, but they have probabilistic bounds on how uneven the sizes of the partitions can be. That is, with high probability, the partitions are approximately load balanced. Since the guarantees for evenly sized partitions are probabilistic, an out-of-core sorting algorithm using these methods must have a provision for recovering from a bad partition. To date, we do not know of any out-of-core sorting implementation for clusters that uses any of these methods.

Implementations

Since there is a wide gap between theory and practical implementations in the field of out-of-core sorting on a cluster, it is appropriate to discuss the implementations that currently exist. Jim Gray maintains the sorting benchmark webpage [Gra] that keeps track of current sorting records of various kinds. Some of these sorting programs, like HPVM Minutesort, are in-core: the size N of the input file is no more than the size M of memory. In some case, they are almost in-core; $N \leq 2M$.

Others target either a shared-memory system (Ordinal Nsort) or a uniprocessor system (Alphasort [NBC$^+$95]). Some of the sorting implementations discussed in [Gra] (SPSort, for example) are custom-engineered for the benchmarks.

Randomized, partitioning-based, out-of-core sorting on a cluster

One might think that a randomized partitioning-based algorithm like the one in [DNS91] is ideal for sorting out-of-core data on distributed-memory clusters. Below, we discuss the stages of a typical partitioning-based algorithm, along with the main issues in adapting each stage to an out-of-core setting of distributed-memory clusters:

- **Find the partitioning elements**. In most cases, each processor selects a random sample of its data, and the partitioning elements are decided by sorting the set of the samples of all the processors [DNS91, HJ97]. The sorting of this set is most often serialized and takes place on one processor, which then broadcasts the $P - 1$ partitioning elements $l_1, l_2, \ldots, l_{P-1}$ to all processors. This step, therefore, consists of two rounds of communication (sending samples to one processor and the broadcast of the partitioning elements) and requires less than a full pass, since no records are written to the disks and perhaps not even all records need to be read.

- **Distribute the data**. All N records are distributed among the P processors as follows. Letting $l_0 = -\infty$ and $l_P = \infty$, then for $i = 0, 1, \ldots, P - 1$, processor i gets all elements with keys m such that $l_i < m \le l_{i+1}$. This step proceeds on each processor in several phases, where each phase has two steps. First, each processor reads in a memoryload of data, divides it into

P groups, one per processor, and then sends each group to the appropriate processor. Next, each processor writes out the data that it received. None of the phases are guaranteed to be load balanced. In other words, almost all the records in a phase can belong to one single processor, leading to unbalanced communication and unbalanced I/O, the latter being more costly. Note that this imbalance within individual phases can exist even if the partitioning elements divide the entire data into approximately equal-sized partitions.

- **Locally sort each partition**. After the distribution stage, there is no guarantee that the partitions are equal in size. Since any load balancing requires each partition to be sorted, the total time for this step is dictated by the time to sort the largest partition. Each processor then locally sorts its partition using an out-of-core sorting method of choice, which takes at least two passes due to the data being out-of-core. Depending on the sorting method in use, however, one may combine the step of distributing the data and the first pass of sorting each partition into a single pass. The succeeding out-of-core local sorting passes remain.

- **Produce output that is sorted, load-balanced, and striped across all the disks**. Even if all the partitions are equal in size, the output so far is sorted in processor-major order. Since our goal is to produce output that is striped across all the disks, load balanced and with the striping unit being any desired size, an additional pass is needed.

We claim that any partitioning-based algorithm that produces striped output on a distributed-memory cluster needs to make at least three passes over the data: one for distribution, at least one for local sorting, and one for load balancing and striping. Our programs, though restricted in the maximum problem size, require at most four passes over the data. Our programs have predictable worst-case performance, however, and they make no assumptions about the input keys. Curiously, our upper bound of four passes is better than the $\omega(1)$-pass lower bound for the multiheaded, single-disk model proven by Aggarwal and Vitter [AV88], which is at least as strong a model as the PDM. Our upper bound is achieved at the cost of the problem-size restriction.

8.3 Out-of-core sorting on a cluster: Existing implementations

We know of three implementations of out-of-core sorting on a cluster. They are all distribution-based. Neither deals with the main challenge of a distribution-based sort: obtaining perfectly load-balanced partitions. Two of the three implementations, NOW-Sort [ADADC$^+$97] and [CIR98], assume that keys are uniformly distributed, and the other [Gra90] assumes that the exact splitters are known a priori. Neither of these implementations produces output in PDM order.

Chapter 9

Conclusion and Future Work

In this chapter, we summarize the key contributions of this thesis and discuss future work.

We have explored the paradigm of oblivious algorithms for sorting out-of-core data on distributed-memory clusters. In marked contrast to the two dominant paradigms of merging and partitioning, both of which have unpredictable I/O and communication patterns, we offer this third paradigm. With the paradigm of oblivious algorithms, we have achieved our goal: robust out-of-core sorting on a distributed-memory cluster using off-the-shelf software. An oblivious algorithm in an out-of-core setting leads to predetermined I/O and communication patterns, allowing excellent opportunity for both planning and achieving good overlap among I/O, communication, and computation. Using this paradigm, we have designed and implemented several algorithms that sort out-of-core data on clusters.

9.1 Key contributions

Here, we summarize the main contributions of this thesis:

- As a first milestone, we present three out-of-core sorting programs [CC02, CCW01] for distributed-memory clusters: *non-threaded 4-pass columnsort*, *4-pass columnsort*, and *3-pass columnsort*. Each of these three programs is based on Leighton's columnsort [Lei85], an oblivious sorting algorithm. In non-threaded 4-pass columnsort, we find that the oblivious nature of columnsort does indeed allow us to adapt it to a robust out-of-core implementation, generating I/O and communication patterns that are predetermined, regular, and balanced across all processors and disks. As an encouraging first result, on Sun-A and Sun-B, we found non-threaded 4-pass columnsort to be competitive with NOW-Sort [CCW01]. In 4-pass columnsort, we redesign our implementation to use threads; this modification reduces the running time to approximately half of the non-threaded version, apart from making the code much cleaner. The 3-pass columnsort implementation is a result of algorithmic design and engineering toward combining the last two passes into a single pass, an improvement that decreases the running time by as much as 25%.

Our preliminary implementations suffer from a restriction on the maximum problem size that they can sort. Columnsort is an 8-step algorithm that sorts N records arranged as an $r \times s$ mesh, where $N = rs$, subject to the divisibility restriction and the height restriction. The problem size restriction is a combined artifact of the height restriction ($r \geq 2s^2$) and the height interpretation ($r = M/P$).

The height restriction is due the underlying algorithm. The height interpretation, on the other hand, is a result of our adaptation of columnsort to an out-of-core setting. The next items in the list of contributions summarize the results of our efforts toward relaxing this restriction by employing theoretical as well as engineering techniques.

- Subblock columnsort and slabpose columnsort are results of new theoretical ideas. Subblock columnsort adds two steps to the original columnsort algorithm, whereas slabpose columnsort is an entirely new oblivious algorithm that we designed specifically for the out-of-core setting. Both subblock columnsort and slabpose columnsort relax the problem-size restriction. In the course of our theoretical work, we also show that the divisibility restriction in the original columnsort algorithm is unnecessary.

- The implementations of M-columnsort (where we set $r = M$), subblock columnsort, and slabpose columnsort result from carefully engineering these algorithms into the already robust and efficient implementation of 3-pass columnsort. The main engineering challenge of subblock columnsort was to find a permutation that has the desired properties mentioned in Section 4.1. For slabpose columnsort, the tricky engineering part was to figure out a 3-pass implementation of the 10-step algorithm.

- We present experimental results for M-columnsort, subblock columnsort and slabpose columnsort. Subblock columnsort, with its 4 passes, runs as fast as 4-pass columnsort. On Jefferson, M-columnsort is much slower than all the other implementations, since it incurs substantial amounts of communication and additional computation when compared to 3-pass columnsort. Experimental evidence on Jefferson shows that, as expected, the relaxed problem-size restriction of slabpose columnsort comes at almost no loss in performance: slabpose columnsort runs nearly as fast as 3-pass columnsort.

- Our implementations have several robustness properties. They use only standard, off-the-shelf software: MPI [SOHL$^+$98, GHLL$^+$98] for communication and UNIX file system calls for I/O. There are no assumptions required about the keys. To the best of our knowledge, our implementations are the first to sort out-of-core data on a distributed-memory cluster without making any assumptions about the input keys. We parallelize all disk I/O operations across all the disks in the cluster. The output appears in the standard striped ordering used by the Parallel Disk Model (PDM). To the best of our knowledge, our programs are the first multiprocessor sorting algorithms whose output is in striped PDM order.

9.2 Future work

We envision the following directions for extending the work of this thesis:

- We intend to implement an adaptive version of slabpose columnsort. Our current implementation sets the parameter k to be P. As mentioned in Section 5.1, columnsort is a special case of slabpose columnsort; setting $k = 1$ in slabpose columnsort leads to columnsort. Moreover, setting $k = \sqrt{s}$ in slabpose columnsort leads to an algorithm that has a problem-size restriction identical to that of subblock columnsort. Our hope is to have one single implementation of slabpose columnsort that sets the value of k depending on the input parameters, thereby sparing the user the need to choose between 3-pass columnsort, slabpose columnsort, and subblock columnsort. The main challenge of this approach is to engineer slabpose columnsort with $k = \sqrt{s}$ so that the resulting implementation performs no worse than subblock columnsort.

- All our implementations make at least three passes over the data. We plan to investigate the possibility of a two-pass algorithm. Ideally, we would like to implement a two-pass out-of-core algorithm that sorts data in an oblivious manner, or we would like to determine an upper bound on how much data such an algorithm can sort.

- In this thesis, we present no comparisons of our later implementations with sorting implementations from the two dominant paradigms of merging and distribution. In the case of the merging paradigm, no multiprocessor implementation exists. For the distribution paradigm, the existing implementations assume uniformly distributed data. Nevertheless, it would be interesting to compare the performance of columnsort with that of a distribution-based sorting algorithm, on a given experimental testbed. We intend to implement a distribution-based algorithm that sorts out-of-core data. This implementation will not make any assumptions about the keys for correctness, though it may rely on certain such assumptions for performance. Some challenges of building such an implementation would be to handle the unpredictable I/O and communication patterns, produce load-balanced as well as striped output, and select an appropriate sampling method.

- We would like to investigate extensions of the paradigm of oblivious algorithms to solve problems related to sorting. Such problems include incorporating prior information about the data, producing the first few elements of the output quickly, and approximately sorting the data according to some measure of approximation.

Bibliography

[ABP98] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129. ACM Press, 1998.

[ACPtNt95] Thomas E. Anderson, David E. Culler, David E. Patterson, and the NOW team. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[ADADC+97] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. In *SIGMOD*, 1997.

[AP94] Alok Aggarwal and C. Greg Plaxton. Optimal parallel sorting in multi-level storage. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 659–668, January 1994.

[Arg97] Lars Arge. External-memory algorithms with applications in geographic information systems. In M. van Kreveld, J. Nievergelt,

T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*, volume 1340 of *Lecture Notes in Computer Science*, pages 213–254. Springer-Verlag, 1997.

[ATV01] Lars Arge, Laura Toma, and Jeffrey Scott Vitter. I/O-efficient algorithms for problems on grid-based terrains. *Journal of Experimental Algorithmics*, 6(1), 2001.

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

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computing Conference*, volume 32, pages 307–314, 1968.

[BBH99] Mark Baker, Rajkumar Buyya, and Dan Hyde. Cluster computing: A high-performance contender. *IEEE Computer*, 32(7):79–80, July 1999.

[BG02] Gordon Bell and Jim Gray. What's next in high-performance computing? *Communications of the ACM*, 45(2):91–95, 2002.

[BGV96] Rakesh D. Barve, Edward F. Grove, and Jeffrey Scott Vitter. Simple randomized mergesort for parallel disks. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 109–118, June 1996.

[BHJ96] David A. Bader, David R. Helman, and Joseph Jájá. Practical parallel algorithms for personalized communication and integer sorting. *Journal of Experimental Algorithmics*, 1(3):1–42, 1996.

[BLM$^+$98] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems*, 31:135–167, 1998.

[BV02] Rakesh D. Barve and Jeffrey Scott Vitter. A simple and efficient parallel disk mergesort. *Theory of Computing Systems*, 35(2):189–215, April 2002.

[CC02] Geeta Chaudhry and Thomas H. Cormen. Getting more from out-of-core columnsort. In *4th Workshop on Algorithm Engineering and Experiments (ALENEX 02)*, pages 143–154, January 2002.

[CC03] Geeta Chaudhry and Thomas H. Cormen. Stupid columnsort tricks. Technical Report TR2003-444, Dartmouth College Department of Computer Science, April 2003. Submitted to *Journal of Algorithms*.

[CCH] Geeta Chaudhry, Thomas H. Cormen, and Elizabeth A. Hamon. Parallel out-of-core sorting: The third way. *Cluster Computing*. To appear.

[CCW01] Geeta Chaudhry, Thomas H. Cormen, and Leonard F. Wisniewski. Columnsort lives! An efficient out-of-core sorting program. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, July 2001.

[CFM$^+$98] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for some geometric problems. In *Proceedings of the Fourteenth Annual Symposium on Computational Geometry*, pages 259–268. ACM Press, 1998.

[CGG$^+$95] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, January 1995.

[CH97] Thomas H. Cormen and Melissa Hirschl. Early experiences in evaluating the Parallel Disk Model with the ViC* implementation. *Parallel Computing*, 23(4–5):571–600, June 1997.

[CHC03] Geeta Chaudhry, Elizabeth A. Hamon, and Thomas H. Cormen. Relaxing the problem-size bound for out-of-core columnsort. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, June 2003. SPAA Revue paper.

[CIR98] Matthew M. Cettei, Walter B. Ligon III, and Robert B. Ross. Out of core sorting on Beowulf class computers. Technical Report 1998-01, Parallel Architecture Research Laboratory, Clemson University, October 1998.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.

[DNS91] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 280–291. IEEE Computer Society Press, 1991.

[DS03] Roman Dementiev and Peter Sanders. Asynchronous parallel disk sorting. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 138–148, June 2003.

[Fri56] Edward H. Friend. Sorting on electronic computer systems. *Journal of the ACM*, 3(3):134–168, July 1956.

[GHLL$^+$98] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI—The Complete Reference, Volume 2, The MPI Extensions*. The MIT Press, 1998.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[Gra] Jim Gray. http://research.microsoft.com/barc/sortbenchmark/. Sort Benchmark Home Page.

[Gra90] Goetz Graefe. Parallel external sorting in Volcano. Technical Report CU-CS-459-90, University of Colorado at Boulder, Department of Computer Science, March 1990.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[GTVV93] Michael T. Goodrich, Jyh-Jong Tsay, Darren E. Vengroff, and Jeffrey Scott Vitter. External-memory computational geometry. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pages 714–723, November 1993.

[Ham03] Elizabeth A. Hamon. Enhancing asynchronous parallel computing. Technical Report TR2003-460, Dartmouth College Department of Computer Science, 2003.

[HBJ98] David R. Helman, David A. Bader, and Joseph Jájá. A randomized parallel sorting algorithm with an experimental study. *Journal of Parallel and Distributed Computing*, 52(1):1–23, July 1998.

[HJ97] David R. Helman and Joseph Jájá. Sorting on clusters of SMPs. Technical Report CS-TR-3833, University of Maryland, College Park, Department of Computer Science, 1997.

[HJB98] David R. Helman, Joseph Jájá, and David A. Bader. A new deterministic parallel sorting algorithm with an experimental evaluation. *Journal of Experimental Algorithmics*, 3(4):1–24, 1998.

[HSV01a] David A. Hutchinson, Peter Sanders, and Jeffrey Scott Vitter. Duality between prefetching and queued writing with applications to external sorting. In *European Symposium on Algorithms*, volume 2161 of *Lecture Notes in Computer Science*, pages 62–73. Springer-Verlag, August 2001.

[HSV01b] David A. Hutchinson, Peter Sanders, and Jeffrey Scott Vitter. The power of duality for prefetching and sorting with parallel disks. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 334–335, July 2001.

[ID90] Balakrishna R. Iyer and Daniel M. Dias. System issues in parallel sorting for database systems. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 246–255. IEEE Computer Society, February 1990.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison Wesley, 1998.

[Lei85] Tom Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34(4):344–354, April 1985.

[Lei92] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.

[MG88] John M. Marberg and Eli Gafni. Sorting in constant number of row and column phases on a mesh. *Algorithmica*, 3:561–572, 1988.

[NBC$^+$95] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and David B. Lomet. Alphasort: A cache-sensitive parallel external sort. *VLDB Journal*, 4(4):603–627, 1995.

[NV91] Mark H. Nodine and Jeffrey Scott Vitter. Large-scale sorting in parallel memories. In *Proceedings of the 3rd Annual ACM Sympo-*

sium on Parallel Algorithms and Architectures, pages 29–39, July 1991.

[NV93] Mark H. Nodine and Jeffrey Scott Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 120–129, June 1993.

[NV95] Mark H. Nodine and Jeffrey Scott Vitter. Greed sort: Optimal deterministic sorting on parallel disks. *Journal of the ACM*, 42(4):919–933, July 1995.

[PB93] Markus Pawlowski and Rudolf Bayer. Parallel sorting of large data volumes on distributed memory multiprocessors. In Arndt Bode and Mario Dal Cin, editors, *Parallel Computer Architectures: Theory, Hardware, Software, Applications*, pages 246–264. Springer-Verlag, 1993.

[Pea99] Matthew D. Pearson. Fast out-of-core sorting on parallel disk systems. Technical Report PCS-TR99-351, Dartmouth College Department of Computer Science, June 1999.

[PSV94] Vinay Sadananda Pai, Alejandro A Schäffer, and Peter J. Varman. Markov analysis of multiple-disk prefetching strategies for external merging. *Theoretical Computer Science*, 128:211–239, 1994.

[Raj01] Sanguthevar Rajasekaran. A framework for simple sorting algorithms on parallel disk systems. *Theory of Computing Systems*, 34(2):101–114, 2001.

[RJ00] Sanguthevar Rajasekaran and Xiaoming Jin. A practical realization of parallel disks. In *International Workshop on Parallel Processing*, pages 337–344, August 2000.

[RV83] John H. Reif and Leslie G. Valiant. A logarithmic time sort for linear size networks. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 10–16. ACM Press, 1983.

[SGV03] Alexander S. Szalay, Jim Gray, and Jan Vandenberg. Petabyte scale data mining: Dream or reality? In *SPIE Conference on Astronomical Telescopes and Instrumentation 4836-47*, August 2003.

[SKT$^+$00] Alexander S. Szalay, Peter Z. Kunszt, Ani Thakar, Jim Gray, Don Slutz, and Robert J. Brunner. Designing and mining multi-terabyte astronomy archives: the sloan digital sky survey. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 451–462. ACM Press, 2000.

[SOHL$^+$98] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI—The Complete Reference, Volume 1, The MPI Core*. The MIT Press, 1998.

[SS86] C. P. Schnorr and A. Shamir. An optimal sorting algorithm for mesh connected computers. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 255–263, May 1986.

[Tys02] J. Anthony Tyson. Large synoptic survey telescope: Overview. In *Proceedings of SPIE 4836*, pages 10–20, August 2002.

[Ull99a] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I: Classical Database Systems*. Computer Science Press, 1999.

[Ull99b] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II: The New Technologies*. Computer Science Press, 1999.

[VG84] Patrick Valduriez and Georges Gardarin. Join and semijoin algorithms for a multiprocessor database machine. *ACM Transactions on Database Systems (TODS)*, 9(1):133–161, 1984.

[VH01] Jeffrey Scott Vitter and David A. Hutchinson. Distribution sort with randomized cycling. In *12th Annual SIAM/ACM Symposium on Discrete Algorithms*, pages 77–86, January 2001.

[Vit91] Jeffrey Scott Vitter. Efficient memory access in large-scale computation. In *Proceedings of the 1991 Symposium on Theoretical Aspects of Computer Science (STACS '91)*, pages 26–41, Berlin, 1991. Springer-Verlag. Published as *Lecture Notes in Computer Science* volume 480.

[Vit01] Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with MASSIVE DATA. *ACM Computing Surveys*, 33(2):209–271, June 2001.

[VS94] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2/3):110–147, August and September 1994.

[ZL97] Weiye Zhang and Per-Åke Larson. Dynamic memory adjustment for external mergesort. In *Proceedings of 23rd International Conference on Very Large Data Bases*, pages 376–385, August 1997.

[ZL98] Weiye Zhang and Per-Åke Larson. Buffering and read-ahead strategies for external mergesort. In *Proceedings of 24rd International Conference on Very Large Data Bases*, pages 523–533, August 1998.

Appendix A

Appendix

A.1 Parallel Disk Model

This section describes a model for parallel I/O, which is based on the Parallel Disk Model [VS94]. This model lays out data on multiple disks in a specific manner, which is the output order produced by our implementations.

In the *Parallel Disk Model*, or *PDM*, N records are stored on D *disks* $\mathcal{D}_0, \mathcal{D}_1, \ldots \mathcal{D}_{D-1}$, with N/D *records* stored on each disk. A record is a contiguous sequence of bytes. The records on each disk are partitioned into *blocks* of B records each. Any disk access (read or write) transfers an entire block of records between the disks and an M-record *memory*. The PDM lays out data on a parallel disk system as shown in Figure A.1. A *stripe* consists of the D blocks at the same location on all D disks.

We assess the I/O efficiency of a PDM algorithm by the number of parallel I/O operations it requires. Each *parallel I/O operation* transfers up to D blocks between the disks and memory, with at most one block transferred per disk, for a total of up to BD records transferred. When records are stored in PDM order, we can access them, *in order*, BD records at a time. With other orderings, we cannot access the records both in order and BD at a time.

Since each parallel I/O operation accesses at most BD records, any algorithm that must access all N records requires $\Omega(N/BD)$ parallel I/Os, and so $O(N/BD)$ parallel I/Os is the analogue of linear time in sequential computing. A *pass* consists of reading each record once, doing some computation, and writing it back to disk, with a cost of $2N/BD$ parallel I/O operations. Vitter and Shriver showed an asymptotically tight bound of $\Theta\left(\frac{N}{BD}\frac{\lg(N/B)}{\lg(M/B)}\right)$ parallel I/Os for sorting. Several asymptotically optimal sorting algorithms for the PDM have appeared in the literature; see Vitter's survey of external-memory algorithms [Vit01] for references. Few of these algorithms have been implemented, however, for one of two reasons. Some of these algorithms, though asymptotically optimal, have relatively large constant factors. Others are so intricate that they are too difficult to implement.

We make some additional assumptions for our implementation of out-of-core sorting on a cluster:

- Each record contains a *key* embedded at a fixed offset within the record.
- There are P processors $\mathcal{P}_0, \mathcal{P}_1, \ldots \mathcal{P}_{P-1}$ that communicate via a network.
- The D disks and the M-record memory are evenly partitioned among the P processors. Each processor can hold M/P records in its memory, and each processor accesses D/P disks.
- We require that $BD \leq M/P$, so that each processor's memory can hold the contents of one block from each disk, and we also require that $M < N$ so

	$\mathcal{P}_0$				$\mathcal{P}_1$				$\mathcal{P}_2$				$\mathcal{P}_3$			
	$\mathcal{D}_0$		$\mathcal{D}_1$		$\mathcal{D}_2$		$\mathcal{D}_3$		$\mathcal{D}_4$		$\mathcal{D}_5$		$\mathcal{D}_6$		$\mathcal{D}_7$	
stripe 0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
stripe 1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
stripe 2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
stripe 3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Figure A.1: The layout of $N = 64$ records in a parallel disk system with $P = 4$, $B = 2$, and $D = 8$. Each box represents one block. The number of stripes is $N/BD = 4$. Numbers indicate record indices.

that the problem is out-of-core.

- In the case of a cluster architecture, it is difficult to characterize optimum I/O complexity in terms of number of parallel I/O operations on D disks. In our implementations, we assume the following analog of parallel I/O. First, the I/O operations must be evenly distributed across the P processors. Second, the reads and writes on each processor's disks must be parallel, and hence evenly distributed across the D/P disks of that processor.

Our implementations are not designed to be PDM algorithms, although they produce output that is laid out in the PDM order of Figure A.1. Producing output in PDM order enables our implementations to be used as subroutines in out-of-core applications that rely on PDM ordering.

A.2 Subblock permutation and the subblock property

This section presents the general version of the proofs presented in Section 4.1, where we assume that all parameters are powers of 2. In our implementation, we do not assume all parameters to be powers of 2. We do assume, however, that the greater of the two values P and $\sqrt{s}$ is an integer multiple of the other. We prove that the subblock permutation has the following four properties:

1. It is a permutation. In other words, no two distinct elements map to the same position in the target mesh.
2. It has the subblock property, i.e., any two distinct elements in the same source subblock map to two different column numbers.
3. In the communicate phase of each round, each processor sends only $\lceil P/\sqrt{s} \rceil$ messages.
4. When $\sqrt{s} \geq P$, so that $\lceil P/\sqrt{s} \rceil = 1$, the one message is always destined for the sending processor, and therefore no communication over the network occurs.

To prove these four properties, we will refer to the algebraic version of subblock permutation, which maps the (i, j) entry to position (i', j'), where

$$i' = \left\lfloor \frac{j}{\sqrt{s}} \right\rfloor \frac{r}{\sqrt{s}} + \left\lfloor \frac{i}{\sqrt{s}} \right\rfloor , \tag{A.1}$$

$$j' = j \bmod \sqrt{s} + (i \bmod \sqrt{s})\sqrt{s} . \tag{A.2}$$

For the rest of the discussion, let (i'_1, j'_1) be the target entry of (i_1, j_1), and let (i'_2, j'_2) be the target entry of (i_2, j_2). Therefore,

$$j'_1 = j_1 \bmod \sqrt{s} + (i_1 \bmod \sqrt{s})\sqrt{s} ,$$

$$j'_2 = j_2 \bmod \sqrt{s} + (i_2 \bmod \sqrt{s})\sqrt{s} ,$$

$$i'_1 = \left\lfloor \frac{j_1}{\sqrt{s}} \right\rfloor \frac{r}{\sqrt{s}} + \left\lfloor \frac{i_1}{\sqrt{s}} \right\rfloor ,$$
$$i'_2 = \left\lfloor \frac{j_2}{\sqrt{s}} \right\rfloor \frac{r}{\sqrt{s}} + \left\lfloor \frac{i_2}{\sqrt{s}} \right\rfloor .$$

Proof of property 1

We first show that the subblock permutation maps each mesh element to a valid entry of the mesh. Then, we show that if two elements map to the same position in the target mesh, then the two source elements are not distinct.

Given that $0 \leq i \leq r-1$, and $0 \leq j \leq s-1$, it follows from equation (A.1) that

$$\begin{aligned} i' &\leq (\sqrt{s}-1)r/\sqrt{s} + (r/\sqrt{s}-1) \\ &= r/\sqrt{s}(\sqrt{s}-2) \\ &< r , \end{aligned}$$

so that $0 \leq i' < r$. Also, it follows from equation (A.2) that

$$\begin{aligned} j' &\leq (s-1) \bmod \sqrt{s} + ((r-1) \bmod \sqrt{s})\sqrt{s} \\ &\leq (\sqrt{s}-1) + (\sqrt{s}-1)\sqrt{s} \\ &= (\sqrt{s}-1)(1+\sqrt{s}) \\ &= s-1 \leq s , \end{aligned}$$

and so $0 \leq j' < s$. Therefore, for all (i, j), the target entry (i', j') is a valid entry of the mesh.

Now, we show that if $j'_1 = j'_2$ and $i'_1 = i'_2$, then $j_1 = j_2$ and $i_1 = i_2$. Because

$j'_1 = j'_2$, we have

$$j_1 \bmod \sqrt{s} + (i_1 \bmod \sqrt{s})\sqrt{s} = j_2 \bmod \sqrt{s} + (i_2 \bmod \sqrt{s})\sqrt{s}\ ,$$

which in turn implies

$$(j_1 - j_2) \bmod \sqrt{s} = (i_2 \bmod \sqrt{s} - i_1 \bmod \sqrt{s})\sqrt{s}\ .$$

The right-hand side of the above equation is a multiple of $\sqrt{s}$, whereas the left-hand side is at most $\sqrt{s}$, implying that both sides are equal to 0, which in turn implies that $j_1 = j_2$ and $i_2 \bmod \sqrt{s} = i_1 \bmod \sqrt{s}$. Also, since $i'_1 = i'_2$, we have

$$\left\lfloor \frac{j_1}{\sqrt{s}} \right\rfloor \frac{r}{\sqrt{s}} + \left\lfloor \frac{i_1}{\sqrt{s}} \right\rfloor = \left\lfloor \frac{j_2}{\sqrt{s}} \right\rfloor \frac{r}{\sqrt{s}} + \left\lfloor \frac{i_2}{\sqrt{s}} \right\rfloor .$$

Since $j_1 = j_2$, the equation reduces to

$$\left\lfloor \frac{i_1}{\sqrt{s}} \right\rfloor = \left\lfloor \frac{i_2}{\sqrt{s}} \right\rfloor ,$$

which in turn implies that $i_1 = i_2$, since $i_2 \bmod \sqrt{s} = i_1 \bmod \sqrt{s}$.

Proof of property 2

We prove property 2 by showing its contrapositive: if $j'_1 = j'_2$, then the two distinct entries (i_1, j_1) and (i_2, j_2) do not belong to the same source subblock. In other words, any two entries that map to the same destination column come from two distinct source subblocks.

While proving property 1, we saw that $j'_1 = j'_2$ implies $j_1 = j_2$ and

$i_2 \bmod \sqrt{s} = i_1 \bmod \sqrt{s}$. Since $j_1 = j_2$, the two entries (i_1, j_1) and (i_2, j_2) belong to the same source column. Also, since $i_2 \bmod \sqrt{s} = i_1 \bmod \sqrt{s}$, either $i_1 = i_2$ (in which case the two entries are not distinct) or the row indices i_1 and i_2 are at least $\sqrt{s}$ rows apart, implying that they belong to two different source subblocks. (Recall that each subblock is a contiguous set of $\sqrt{s}$ rows and $\sqrt{s}$ columns).

Proof of property 3 and property 4

To prove property 3, we will first argue that for any column, say column j, there are $\sqrt{s}$ unique destination columns, each pair of which differs by a multiple of $\sqrt{s}$. Then, we will show that there can be at most $\lceil P/\sqrt{s} \rceil$ unique processors that collectively own these $\sqrt{s}$ columns, proving property 4 along the way.

Given that, within a given column j, two row indices that are $\sqrt{s}$ rows apart map to the same destination column, there are exactly $\sqrt{s}$ unique destination columns for any given source column. Furthermore, these $\sqrt{s}$ unique columns are characterized by the destination columns of row indices $0, 1, \ldots, \sqrt{s} - 1$. From equation (A.2), the $\sqrt{s}$ unique destination columns for source column j are $k + 0, k + \sqrt{s}, k + 2\sqrt{s}, \ldots, k + (\sqrt{s} - 1)\sqrt{s}$, where $k = j \bmod \sqrt{s}$.

Now, let us examine the sequence of processors that own these destination columns. We will refer to these processors as the destination processors. Recall that processor $j \bmod P$ owns column j. Therefore, the destination processors are $k \bmod P, (k + \sqrt{s}) \bmod P, \ldots, (k + (\sqrt{s} - 1)\sqrt{s}) \bmod P$. We claim that the number of unique values in this sequence of destination processors is $\lceil P/\sqrt{s} \rceil$. We consider two cases:

- If $\sqrt{s} \geq P$, we can say that $\sqrt{s} = aP$, for some $a \geq 0$. The destination processors, then, are $k \bmod P$, $(k+aP) \bmod P, \ldots, (k+(\sqrt{s}-1)aP) \bmod P$, each of which is the same as $k \bmod P$. That is, all destination columns map to a single processor. Note that this single destination processor is $k \bmod P = (j \bmod aP) \bmod P$, which is the same as $j \bmod P$, the sending processor, thus proving property 4.

- If $P > \sqrt{s}$, we can say that $P = b\sqrt{s}$, for some $b > 1$. In the sequence, $k \bmod P, (k + \sqrt{s}) \bmod P, \ldots, (k + (\sqrt{s} - 1)\sqrt{s}) \bmod P$, any two values that are b entries apart are identical: $l \bmod P = (l + b\sqrt{s}) \bmod P$, since $(l + b\sqrt{s}) \bmod P = (l + P) \bmod P = l \bmod P$. Consequently, there are at most $b = \lceil P/\sqrt{s} \rceil$ unique entries.

Therefore, the maximum number of processors that own these destination columns is $\lceil P/\sqrt{s} \rceil$. In other words, each processor needs to send at most $\lceil P/\sqrt{s} \rceil$ messages, one to each destination processor, in any given communication round.

A.3 Symmetry in the optimal schedule

This section justifies the symmetric-processors assumption of Section 6.3.1. We argue that, assuming homogeneous processors with identical job sets, there always exists an optimal schedule of jobs that is also symmetric across processors. We will proceed by showing that any asymmetric schedule can be transformed into a symmetric schedule, where the new schedule takes no longer to complete than the old one.

Let $\mathcal{S}$ be an asymmetric schedule that takes B amount of time to complete. We define $\mathcal{S}_i$ to be the schedule of $\mathcal{P}_i$, and B_i to be the time taken by $\mathcal{P}_i$ to complete its set of jobs, implying that $B = \max_i\{B_i\}$ over all i. Since $\mathcal{S}$ is asymmetric, there exists at least one processor whose schedule differs from $\mathcal{S}_1$, the schedule of $\mathcal{P}_1$. Without loss of generality, let this processor be $\mathcal{P}_2$. Recall that the schedule of each processor is a sequence of all of the jobs in the job set of that processor, where this sequence obeys the given sequentiality constraints and the resource constraints.

First, we note that for any i, we have $B_i \leq B$, implying that $B_1 \leq B$ and $B_2 \leq B$. Since the job sets of both $\mathcal{P}_1$ and $\mathcal{P}_2$ are the same, the sequentiality constraints and the resource constraints will still be satisfied if we modify the schedule of $\mathcal{P}_2$ to be $\mathcal{S}_1$, thus implying that B_2 now becomes equal to B_1. This modification can not increase the value of B because $B_1 \leq B$. We keep repeating this step of changing the schedule of a processor to $\mathcal{S}_1$ for as long as the resulting schedule is still asymmetric. After at most $P - 1$ such modification steps, we will have a symmetric schedule that completes in $B_1 \leq B$ units of time.