Virtual Memory for Data-Parallel Computing

by

Thomas H. Cormen

B.S.E., Princeton University (1978)
S.M., Massachusetts Institute of Technology (1986)

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1993

Signature of Author . . . .
Department of Electrical Engineering and Computer Science
September 11, 1992

Certified by . . . .
Charles E. Leiserson
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . .
Campbell L. Searle
Chairman, Departmental Committee on Graduate Students

Virtual Memory for Data-Parallel Computing

by

Thomas H. Cormen

Submitted to the Department of Electrical Engineering and Computer Science
on September 11, 1992, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This thesis explores several issues that arise in the design and implementation of virtual-memory systems for data-parallel computing.

Chapter 1 presents an overview of virtual memory for data-parallel computing. The chapter lists some applications that may benefit from large address spaces in a data-parallel model. It also describes the view of virtual memory for data-parallel computing used in this thesis, along with a machine model. Chapter 1 also summarizes VM-DP, a system we designed and implemented as a first cut at a system for data-parallel computing with virtual memory.

Chapter 2 covers bit-defined permutations on parallel disk systems. It gives an algorithm for performing bit-permute/complement, or BPC, permutations, a class of permutations in which each target address is formed by permuting the bits of its corresponding source address according to a fixed bit permutation and then complementing a fixed subset of the resulting bits. This class includes common permutations such as matrix transpose (with dimensions integer powers of 2), bit-reversal permutations, hypercube permutations, and vector-reversal permutations.

Chapter 2 also shows how to perform bit-matrix-multiply/complement, or BMMC, permutations efficiently. In a BMMC permutation, each target address is formed by multiplying its corresponding source address, treated as a vector, by a nonsingular matrix over $GF(2)$ and then complementing a fixed subset of the resulting bits. BMMC permutations include all BPC permutations and classes such as Gray code and inverse Gray code permutations.

Not only are the algorithms for BPC and BMMC permutations efficient, but they are also deterministic, easily programmed, and can be performed "on-line" in the sense that they take little time and space.

Chapter 2 also proves lower bounds for BMMC and BPC permutations, showing that the BPC algorithm is asymptotically optimal.

Chapter 3 examines the differences between performing general and special permutations. It presents a simple, though asymptotically suboptimal, algorithm to perform general permutations by sorting records according to their target addresses. Chapter 3 also explores several classes of special permutations that we can perform much faster than general permutations: monotonic and k-monotonic routes, mesh and torus permutations, BMMC and BPC permutations, and general matrix transpose. These classes arise frequently. Chapter 3 shows, for each of these classes, how to perform them quickly and how to detect them at run-time given a vector of target addresses.

Chapter 3 also focuses on the question of how to invoke special permutations. It argues that although we can detect many special permutations quickly at run time, it is better to invoke them by specifying them in the source code. Chapter 3 supports this argument with empirical data for BPC permutations.

In a data-parallel computer with virtual memory, the way in which vectors are laid out on the disk system affects the performance of data-parallel operations. Chapter 4 presents a general method of vector layout called banded layout, in which we divide a vector into bands of a number of consecutive vector elements laid out in column-major order. Chapter 4 analyzes the effect of band size on the major classes of data-parallel operations, deriving the following results:

- For permuting operations, the best band sizes are a track or smaller. Moreover, regardless of the number of grid dimensions, we can perform mesh and torus permutations efficiently.
- For scan operations, the best band size equals the size of the I/O buffer, and these sizes depend on several machine parameters.
- The band size has no effect on the performance of reduce operations.
- When there are several different record sizes, band sizes for elementwise operations should be based on the largest record size, which has the smallest band size.

Chapter 5 presents the design of the demand-paging portion of the VM-DP system. We implemented three different paging schemes and collected empirical data on their I/O performance. One of the schemes was straight LRU (least-recently used) paging, in which all tracks are treated equally. The other two schemes treat tracks differently according to the sizes of vectors they contain. The empirical tests yielded two somewhat surprising results. First, the observed I/O counts for the test suite were roughly equal under all three schemes. Moreover, as problem sizes increased, performance differences among the three schemes decreased. Second, the best scheme overall of the three was the simplest one: straight LRU paging. The other two schemes, which attempt to account for vast differences in vector sizes, were not quite as good overall. Chapter 5 also discusses some of the addressing and implementation issues that arose in the VM-DP system.

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Computer Science and Engineering

Contents

Acknowledgments

I'll try to keep this short, but it won't be easy. Over the course of eight years, so many people have helped me, at MIT and elsewhere.

I had the good fortune to meet Charles Leiserson as soon as I arrived at MIT. I cannot imagine a better advisor.[1] Charles didn't have all the answers, but he had something even more valuable: the right questions. He also knew when to scratch me behind the ears and when to let me sink or swim. And for five of my eight years at MIT, Charles arranged my support.[2] Charles, I am proud to call you my advisor, coauthor, and friend.

I also consider myself fortunate to know Ron Rivest and Tom Leighton, the other members of my thesis committee. I worked with Ron for over three years on *Introduction to Algorithms.* Thank you, Ron, for being so easy to work with.[3] I have known Tom since we were classmates at Princeton. Tom, I've finally realized why it's 1992 and you're a full professor and I'm still a graduate student: you're a lot smarter than me.

Lars Bader implemented the VM-DP system with me. As hard as I worked finishing up this thesis, Lars worked harder. Not only was he reworking our paging system on his own, he was also trying to finish up his own thesis. Thank you, Lars.

I also received help from many other people in the course of researching and writing this thesis. I hope I've got them all: Tim Bartel, Bonnie Berger, Bobby Blumofe, Guy Blelloch, Ted Charrette, Marc D'Alarcao, Dan Davenport, Michael Ernst, Michelangelo Grigni, Steve Heller, Peter Highnam, Marshall Isman, C. Esther Jesurum, Michael Klugerman, Steven Kratzer, Martin Lewitt, Bruce Maggs, Nashat Mansour, Mark Nodine, James Park, Cindy Phillips, C. Greg Plaxton, John Prentice, Jim Salem, Eric Schwabe, Sandy Seale, Mark Sears, Jay

[1]Except maybe for one who gets off the pitcher's mound and covers first base quickly on grounders to the right side.

[2]I was supported for three years by a National Science Foundation fellowship and for five years by the Defense Advanced Research Projects Agency under Grant N00014-91-J-1698.

[3]One should not infer, because I thank Ron for being easy to work with but I don't thank Charles for same, that Charles is not easy to work with. He's just, well, different.

Sipelstein, Burton Smith, Stephen Smith, Cliff Stein, Roberto Stolow, Lisa Tucker, and Marco Zagha.

I have had several fine officemates at MIT, total shredders all. In chronological order: Val Breazu-Tannen, Bruce Maggs, Alex Ishii, Sally Goldman, Cliff Stein, Jon Riecke, Cynthia Dwork, and Igal Galperin.

Thanks also to the support staff in the Theory of Computation Group at MIT. William Ang, David Jones, and Tim Wright answered my picky LaTeX questions. And I'm glad to have Denise Sergent-Leventhal, Be Hubbard, Cheryl Patton, and Arline Benford as my friends. Thanks also to all the other students, faculty, and staff in the Theory Group over the last eight years for making my MIT experience way more fun than I had imagined when I moved to Cambridge from Santa Cruz. I'll miss you all.

The MIT crowd is only part of the story. Equally essential has been the support of my family and friends. Among my friends, I want to specially thank Paul Daro and Stanley Geberer. I guess I should thank Craig and Mary Ward, too, for letting me eat off Craig's back. At least I no longer have to hear Craig ask, "Wait a minute. Are you still working on the *same* book?"

The staff of Sports Etc. in Arlington thanks Bruce Maggs and Cindy Phillips for getting me started in hockey. And I thank Rejean Lemelin, for showing me that some old goalies can still play, and all the Sloths who actually showed up.

I am blessed to have had the love and support of my parents, Renee and Perry Cormen, and my wife's parents, Colette and Paul Sage. With all my love, I thank you.

Finally, there's one person whose interest in virtual-memory for data-parallel computing increased markedly over the last three months. I guess that's what the lust to leave the Geek Tower will do to you. Anyway, Nicole, it's now done and you'll never have to worry about virtual memory again. How you've put up with me for the last eight years, and especially the last three months, I'll never know. I give you big thanks and baisers. You're the one who wears my ring, and I love you.

THC

September, 1992

Chapter 1

Virtual Memory for Data-Parallel Computing

It was 1974. I had been programming for about a year, and I was trying to bring up my most ambitious program yet. It was a baseball simulation, and it was to run on the IBM 1130 in my high school. The 1130 had 8K of core memory, and my program wouldn't fit. The operating system valiantly tried to overlay the program, but to no avail. I related my core-memory woes to a friend. My recollection is a tad fuzzy, but I recall him telling me about a computer in Israel that had 150K of core.

I suspected there might be an easier solution than going all the way to Israel.

It was 1980. I was working with a small team of programmers at a startup company developing the first microprocessor-based workstation for VLSI design. Despite an elaborate overlay system for our code, we were having a lot of trouble getting the code and data to fit in the 256K of RAM we had to work with. We lobbied the company management to put another 128K into the machine. RAM wasn't all that cheap in those days. Management acceded to our request, but only after commenting, "If we give you more RAM, you're just going to use it."

We programmers thought that was the idea.

It was 1990. I was finishing up my summer job at Thinking Machines. I had spent the summer designing and implementing a segment library for the C* language. I had asked Jim Salem, who develops visualization code on the Connection Machine, what sort of features he wanted in a segment library, expecting him to list a few more that I might easily add in to what I had done. Instead, Jim told me about visualization code that ran on a dataset so large that the data had to be swapped into RAM from the attached disk array. That didn't seem like it had much to do with the segment library, but I thought about how to implement segment routines that could deal with such huge datasets.

Charles Leiserson pointed out to me that the scope of what Jim wanted was far wider than the segment library.

Jim wanted virtual memory for the Connection Machine.

I could relate to that.

The problem, as we all know, is that faster memory is more expensive. The cost per byte of semiconductor RAM is dropping, but it is still not competitive with the cost per byte of disk storage. As a quick example, an advertisement in a current magazine [Mac92, pp. 390–391] is selling 16 megabytes of Macintosh RAM for $499, or $31.19 per megabyte, and Seagate disks, 1900 megabytes formatted, for $2949, or $1.55 per megabyte. The RAM costs 20 times more per byte than the disk, and this RAM price is very good. RAM for workstations costs even more; a good price is $50–$100 per megabyte.

There may yet come a day when the price of RAM is low enough, and its power and packaging requirements are not too stringent, and reliability is a moot issue, that disk storage is priced out of the picture. But that day is not at hand. Gibson [Gib92, Section 2.4] presents several scenarios extrapolating when the price of DRAM will become competitive with disk prices. His scenarios place the crossover point somewhere between the years 2001 and 2027, with later dates more likely. In the meantime—for the next nine to 36 years—it appears that if your data doesn't fit in RAM, your next best choice is a disk.

OK. We're stuck with RAM as a limited resource. Almost thirty years ago, computer architects devised virtual memory for sequential machines. But parallel machines that support data-parallel programming don't have it yet.

What can we do about that?

1.1 Who needs virtual memory for data-parallel computing?

Throughout the short history of electronic computing, no matter how big and fast the top machines have been, there have always been applications that needed them to be bigger and faster. The largest Thinking Machines Corporation CM-2 has 2^{34} bytes of RAM. That's 16 gigabytes, and it isn't enough for Jim Salem.

If we give Jim a 64-gigabyte machine, will that satisfy him? Maybe for a while, but eventually he'll need more RAM. Hey, Jim, if we give you more RAM, you're just going to use it.

Jim Salem is not alone.

There are many applications that run on parallel machines and use huge datasets. I have compiled a list of some applications that a quick scan of the literature and a well-placed request over a computer network[1] turned up. I don't claim to be knowledgeable about these fields, and it may be the case that some of them don't need virtual memory today. But these applications all have large memory requirements. Moreover, because the huge address space provided by virtual memory is not yet available, people might not yet be thinking about running larger problems than they are running today.

Here's the list, in alphabetic order, including quantitative information that was readily available:

- A* search [EHMN90], a heuristic search algorithm.
- Blackboard systems [Cor91, Dav92].
- Computational biology [Can90, NGHG$^+$91, Wat90]. The human genome project is attempting to determine the human DNA sequence of 3 billion base pairs.
- Computational fluid dynamics [Has87, MK91, SBGM91].
- Conjugate gradient methods [Chr91].

[1]Thanks to Cindy Phillips.

- Genetic algorithms [Man92, MF92].
- Geophysics [MS91, Wil91, YZ91]. Seismic problems are highly parallel and have huge datasets. Highnam [Hig92] and Seale [Sea92] report datasets well into the gigabytes, some over 30 gigabytes.
- Intelligence. According to Myers and Williams [MW91], “Mass storage currently implies the storage of data in excess of 10^{12} bits and it is not uncommon in this community to see requirements of over 10^{15} bits.”
- Iterative parallel region growing [Til90].
- Light propagation simulation [Koc90].
- Logic simulation [CC90].
- Low-energy scattering of neutral atoms from crystal surfaces. Prentice [Pre92a, Pre92b] reports that dense, complex-valued, linear systems with over 10,000 rows and columns require over 1.6 gigabytes just to store the matrix, to say nothing of the memory needed to solve it.
- Meteorology [Che91, Kau88, MMF$^+$91].
- Molecular dynamics [BME90, SBGM91, TMB91].
- Ocean modeling [SDM91].
- Partial differential equations [PF91].
- Synthetic aperture radar image processing [PMG91].
- Visualization and graphics [Dem88, Sal92].

Apparently, a wide range of applications, especially scientific ones, would benefit from virtual memory for data-parallel computing.

1.2 Virtual memory and data-parallel computing

Virtual memory for data-parallel computing has some significant differences from virtual memory for sequential computing.

We adopt Denning's [Den70] definition of *virtual memory*: giving "the programmer the *illusion* that he has a very large main memory at his disposal, even though the computer actually has a relatively small main memory." According to Denning, virtual memory (which we'll sometimes refer to as VM from here on) was first proposed by the research group at Manchester, England in 1961. It was in use by the mid-1960s. For example, VM was an integral part of the Multics system [BCD72, DD68].

Many computer scientists confuse virtual memory, as defined above, with implementations of it. The mention of "virtual memory" usually evokes thoughts of "demand paging" rather than "illusion of a large address space." This distinction is important in VM for data-parallel computing because many issues arise other than demand paging.

Virtual memory for sequential machines

In sequential computing, virtual memory—the illusion of a large address space—is generally accomplished by a combination of hardware and software. Only a portion of the large logical address space appears in RAM at any one time. The remainder resides on a disk or drum. The hardware maps logical addresses appearing in registers to physical addresses in RAM, detecting *faults*—references by the program to logical addresses that are not currently in RAM. The hardware usually maintains rudimentary usage statistics for the physical addresses in RAM, typically information about how recently a range of addresses has been referenced. Software takes over during a page fault, using these statistics to determine which data to eject from RAM and performing disk I/O as necessary to write changed information out to the disk or drum and read the requested data into RAM.

The logical address space is partitioned into pages or segments of contiguous addresses. *Segments* usually are of varying lengths but with a common logical function. Examples would be segments of code or a segment representing a task's run-time stack. *Pages* have fixed lengths and usually no logical grouping of function within each page. A typical page might be 4K bytes, and it could contain both code and data. Segments or pages are usually the smallest unit moved in and out of RAM by the VM system. When a reference to a logical address causes a fault, the entire segment or page containing the address is brought into RAM, and an entire segment or page may be ejected.

To limit the cost of disk accesses, VM systems for sequential machines rely on locality: references to logical addresses tend to cluster. That is, a page or segment that is referenced once is likely to be referenced again soon. Some referencing paradigms exhibit locality, e.g., straight-line code, sequential array accesses, and stack operations. Other paradigms can cause non-local references, e.g., branches in code, subroutine calls and returns, and following pointers.

Virtual memory for data-parallel computing

As used in this thesis, virtual memory for data-parallel computing differs from virtual memory for sequential computing in two fundamental ways: it is for data only and not code, and it exploits predictable patterns of data access.

Data-parallel virtual memory is for data only. Data-parallel computing supports operations on *vectors.* A given computation may use many vectors, and some of them may be rather large. In a VM setting, the amount of data contained in the vectors is far greater than the size of the code. Consequently, the illusion of a large address space is more important for the data than for the code. Moreover, many data-parallel machines use SIMD control. There is a separate, sequential *front-end machine* that broadcasts an instruction at a time to the processors of the parallel machine. The entire program resides in the memory of the front-end machine and not in the memories of the individual parallel processors. If the code goes through any VM mechanism, it is that of the sequential front-end machine, not that of the parallel machine.

Data-parallel virtual memory exploits predictable data-access patterns. The vector operations supported by data-parallel computing have access patterns that tend to be either highly regular or can be choreographed using algorithmic techniques. In sequential VM systems, locality is more of a hope than a guarantee. In data-parallel VM systems, locality is often guaranteed. Consider, for example, elementwise operations. An *elementwise operation* applies a function to one or more *source vectors*, or *operands*, to compute a *target vector*, or *result.* All vectors involved are of equal length, and the value of each element of the result depends only on the corresponding elements in the operands. In an elementwise operation, a reference to *any* element of a given vector implies references to *all* elements of the vector.

What do we mean by access patterns that can be choreographed using algorithmic techniques? Consider a permuting operation. We are given a source vector, a target vector, and a vector of *target addresses* that map elements of the source vector to positions in the target vector. One way to perform a permutation is to sort by target addresses. In general, there is plenty of algorithmic technique that can be brought to bear when sorting. Data-access patterns for some sorting algorithms are predictable to some degree. In fact, some sorting algorithms (such as Leighton's Columnsort [Lei85]) are *oblivious* and access sections of the vector independently of the values being sorted. For some permutations, sorting is not necessary. There are algorithms to perform them that are more efficient than sorting, and their data-access patterns are even more predictable. We shall see examples of such permutations in Chapters 2 and 3.

1.3 The machine model

In this section, we present the model of a parallel machine with an attached parallel disk system that we use throughout this thesis. It extends the parallel-disk-system model proposed by Vitter and Shriver [VS90a, VS90b] to include processors and a mapping from disk locations to local processor memories.

Figure 1.1 shows the model. There are P processors $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{P-1}$, and they can hold a total of M records in RAM. Each processor has its own local RAM and can hold up to M/P records. In this thesis, records are usually of a fixed size, say 4 or 8 bytes. Only occasionally will we concern ourselves with the exact nature of a record.

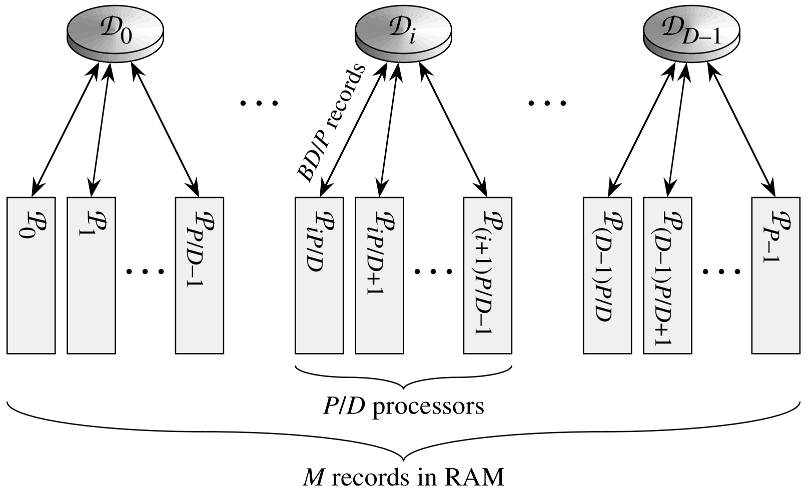


Figure 1.1: The VM data-parallel machine model. The P processors $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{P-1}$ can hold a total of M records in RAM, and an array of D disks $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{D-1}$ is connected to the processors. Records are transferred between disk $\mathcal{D}_i$ and the local RAM of processors $P_{iP/D}, P_{iP/D+1}, \ldots, P_{(i+1)P/D-1}$, with disk I/O occurring in blocks of B records per disk.

An array of D disks $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{D-1}$ is connected to the processors. We assume that $D \leq P$, so that each disk corresponds to one or more processors.[2] In particular, for $i = 0, 1, \ldots, D-1$, we can transfer records between disk $\mathcal{D}_i$ and the local RAM of the P/D processors $P_{iP/D}$, $P_{iP/D+1}, \ldots, P_{(i+1)P/D-1}$.

Disk I/O occurs in *blocks* of B records. That is, when disk $\mathcal{D}_i$ is read or written, B records are transferred to or from the disk. Each block of B records is partitioned into sections of BD/P records, and each section is transferred to or from the RAM of one of the P/D processors connected to disk $\mathcal{D}_i$. All blocks start on multiples of B records on each disk.

Parallel I/O operations

We will be interested primarily in counting the number of *parallel I/O operations*. In a *parallel read operation*, we read a block from any subset of the D disks into the processors' local RAMs. In a *parallel write operation*, we write blocks from the local RAMs to any subset of the disks. Each parallel I/O operation used in this thesis will use all D disks.

[2]If $D > P$, we gang together groups of D/P disks to act in concert as one disk of D/P times the size.

Parallel I/O operations can be either independent or striped. In a *striped* I/O operation, all blocks read or written must reside at the same location on each disk. We call such a set of blocks a *track*.[3] In an *independent* I/O operation, disk blocks read or written may reside at any location as long as only one block per disk is accessed. Striped I/O has the constraining effect of transforming the block size B and number of disks D to the values $B' = BD$ and $D' = 1$. Some of the algorithms and data-access methods in this thesis will use striped I/O and others will use independent I/O. Note that up to BD records may be transferred in one parallel I/O operation, whether independent or striped.

Let us look briefly at how striped and independent I/O operations influence the maintaining of checksums for data reliability. With striped I/O, we can maintain a separate checksum disk. Checksum maintenance is then fast because all the data needed to compute a checksum is present in the write buffer; no additional disk read is required. If we try to maintain checksum information at all times with independent I/O, however, we generally need an extra disk read for each write [PGK88]. With independent I/O, therefore, it may be faster overall to recompute checksums only at occasional checkpoint times over the course of a computation.

There are two minor restrictions placed on the parameters P, B, M, and D. First, for ease of exposition and analysis, we shall assume throughout this thesis that each is an integer power of 2. Second, in order for the RAM to accommodate the records transferred in a parallel I/O operation to all D disks, we require that $BD \leq M$.

Data organization

Data on a parallel I/O system is organized into vectors. A *vector* is an array of records. Although a given application may view a vector as multidimensional array, the VM system views each vector as one-dimensional. We often call an individual record within a vector an *element*, and we shall usually use N to denote the number of records, or *length*, or the vector. We shall often index the elements of vector X as $X_0, X_1, \ldots, X_{N-1}$. We require that for each vector X, element X_0 maps to the RAM of processor $\mathcal{P}_0$.

[3]We use the term "track" to maintain consistency with the Vitter-Shriver terminology; the term "stripe" is sometimes used elsewhere.

Data-parallel instructions operate on one or more vectors at a time. We will generally focus on vectors that occupy several tracks. Vectors are laid out a track at a time. Indexing the tracks from zero,[4] the first BD elements are in track 0, the next BD are in track 1, and so on. In general, element X_k is in track number $\lfloor k/BD \rfloor$. In some chapters, we will pay attention to how vectors are laid out within tracks, but in other chapters it will not matter.

1.4 Previous results and related work

In this section, we summarize previous results for the I/O complexity of problems and other related work.

Early work with no parallelism

Early work on external problems generally assumed just one disk with one read/write head.

External sorting has been studied more with tapes than with disks as the external storage device. With either tapes or disks, external sorting is generally performed as a variation on merge sort [CLR90, Section 1.3]. The hard part is determining the best sequence of merges; Knuth [Knu73, Section 5.4.9] contains a study for disks.

The Fast Fourier Transform, or FFT, was studied as early as 1967 for cases in which the data did not fit in fast memory and had to be stored on a single disk. Using eight disk files, Singleton [Sin67] showed how to compute the FFT of an N-element vector in $\lg N$ I/O passes over the data. Brenner [Bre69] presented two external FFT methods. One, to be used when the data is slightly larger than RAM, makes $\Theta(N/M)$ I/O passes. The other, for when $N^2 \gg M$, makes $\Theta(\lg M)$ I/O passes. Fraser [Fra76] showed a bound of $\Theta\left(\frac{\lg(N/B)}{\lg(M/B)}\right)$ passes, matching the number of passes in the later bounds of Aggarwal and Vitter and of Vitter and Shriver. Kim et al. [KNPF87] explored the use of disk interleaving to speed up data transfers and reduce I/O times.

[4]Just about all indexing in this thesis is from zero.

Floyd [Flo72] showed that general permutations on external storage require $\Omega((N \lg B)/M)$ I/Os. Fraser [Fra76] also studied some special cases of external permutations, in particular bit-reversal permutations and matrix transpose.

McKellar and Coffman [MC69] studied how to organize matrices on disk in order to minimize the number of disk accesses for matrix addition, multiplication, and inversion.

Parallel I/O models

Aggarwal and Vitter [AV88] introduced a model that seems to be more powerful than the Vitter-Shriver model. The Aggarwal-Vitter model has just one disk but with D independent read/write heads. Each head can read or write a B-record block in one parallel I/O. Unlike the Vitter-Shriver model, which requires each of the D blocks accessed simultaneously to reside on different disks, this D-headed model requires no separation of the blocks. They can be any D at all. Curiously, no known problem separates these two models. Obviously, the D-headed model can simulate the D-disk model directly. Conversely, there is no problem known that requires asymptotically fewer I/O operations on the D-headed model than on the D-disk model. It remains an open question whether these two models are equivalent.

Aggarwal and Vitter proved asymptotically tight bounds on the number of parallel I/Os for several problems on their D-headed model, where each parallel I/O operation reads or writes with multiple heads. Vitter and Shriver subsequently proved the same asymptotic bounds on parallel I/Os for the same problems on the D-disk model. In the following list of problems, each bound has a $\Theta(N/BD)$ factor. Since each parallel I/O accesses at most BD records, it takes N/BD parallel I/Os to access each record once. The $\Theta(N/BD)$ factor is thus the parallel I/O analogue of linear time in sequential computing.

- Sorting and FFT (pebbling an FFT network):

  $$\Theta\left(\frac{N}{BD}\,\frac{\lg(N/B)}{\lg(M/B)}\right)$$

  parallel I/Os.

- Performing a general permutation:

  $$\Theta\left(\min\left(\frac{N}{D}, \frac{N}{BD}\frac{\lg(N/B)}{\lg(M/B)}\right)\right)$$

  parallel I/Os. The first term comes into play when the block size is small, and the second term comes from the sorting bound.

- Transpose of an $R \times S$ matrix:

  $$\Theta\left(\frac{N}{BD}\left(1 + \frac{\lg\min(R, S, B, N/B)}{\lg(M/B)}\right)\right)$$

  parallel I/Os.

- Pebbling permutation networks:[5]

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

  parallel I/Os are required for any permutation network, and

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

  parallel I/Os are achieved by a permutation network formed by concatenating three FFT networks together.

- The standard matrix-multiplication algorithm on two $\sqrt{N} \times \sqrt{N}$ matrices:

  $$\Theta\left(\frac{N}{BD}\frac{\sqrt{N}}{\min(\sqrt{N}, \sqrt{M})}\right)$$

  parallel I/Os. (This result was shown only by Vitter and Shriver, for their D-disk model.)

A few comments about these theoretical results are in order.

[5]Here, we are not given a permutation network. Rather, we are interested in finding permutation networks that can be pebbled using the fewest possible parallel I/Os.

- The upper bounds come from algorithms that carefully choreograph their disk I/O and use independent I/O. These characteristics are essential in order to make use of blocked I/O.

- Sorting is important, because the upper bound for general permutations relies on it. Practical sorting algorithms yield practical permuting algorithms as well.

- The Vitter-Shriver upper bound for sorting uses a randomized algorithm and holds with high probability. Nodine and Vitter [NV90, NV91, NV92] subsequently developed deterministic sorting algorithms for the D-disk model.

- Some permutations are harder than others. In particular, matrix transpose requires fewer I/Os than do general permutations when either of the matrix dimensions R and S or the block size B is smaller than the number of blocks N/B. This question of which classes of permutations can be performed faster than general permutations led to the work in Chapters 2 and 3 of this thesis.

Other work in parallel I/O

The above results for the D-headed and D-disk models are of a theoretical bent, although some of the algorithms are practical. There is also a body of more systems-oriented work on parallel disk systems, most notably the RAID (Redundant Arrays of Inexpensive Disks) project at Berkeley [CGK$^+$88, CP90, GHK$^+$89, Gib92, LK91, PGK88].

A RAID is given a sequence of I/O operations to perform; it does not determine them, unlike the algorithms above. RAIDs perform disk I/O quickly and with reliability as an important criterion. That is, they can tolerate the catastrophic failure of a number of disks with no loss of data integrity and a minimal impact on performance.

1.5 The VM-DP system

As a testbed for some of the concepts in this thesis, Lars Bader and I developed the VM-DP system. VM-DP is by no means a production-quality system, but it has been of great help in understanding the costs of VM for data-parallel computing. Once we know where the costs accrue, we know where to focus our efforts.

The current implementation of VM-DP is as the back end of the NESL/VCODE/CVL system developed by Guy Blelloch and his research group at Carnegie Mellon University. Source programs are written in NESL [Ble92], a strongly-typed, applicative, data-parallel language. In NESL, all data is organized into vectors; what most languages view as scalars are 1-element vectors in NESL. The NESL compiler produces VCODE [BC90, BCK$^+$92], a stack-based, intermediate, data-parallel language. VCODE is interpreted, and the interpreter's interface to the underlying machine is CVL [BCSZ91], the C Vector Library. VM-DP is a complete implementation of CVL plus some slight modifications to the VCODE interpreter.

Currently, VM-DP is a simulator of a data-parallel machine with VM. The code comprises about 7500 lines of C and runs under Unix on Sun Sparcstations. Operations on the parallel disk system are simulated by operations on a very large Unix file.

Accessing vectors

The implementations of CVL functions that make up the VM-DP system access vectors in two basic ways: through a demand paging system and by algorithms that choreograph their disk I/O.

We use the demand paging system for vector operations that make sequential passes through vectors. Such operations include elementwise operations, scans, and reduces. We examined elementwise operations briefly in Section 1.2, and Chapter 4 will focus on scans. In addition, one type of permuting operation, monotonic routes (see Section 3.3), goes through the demand paging system.

Chapter 5 presents the VM-DP paging system in detail, but a few key characteristics of it are worth noting here. We were willing to have all vector accesses go through a software mechanism to check whether the requested vector already resides in RAM and to read it in if necessary. This approach is unlike conventional serial machines with VM, which typically have hardware support. We were willing to go through software for three reasons. First, our code is only a simulation and so we really had no choice. Second, implementing the entire system in software permits rapid prototyping of paging policies. Third, accesses are of vectors, which often contain many elements. An individual access, therefore, may very well be for many values, not just one. The software overhead is then spread over the many elements accessed.

Although VM-DP is designed for amounts of data in excess of the RAM size, we want performance to be good when all the data fits in RAM. In other words, computations that do not really need VM should not pay a high price for its presence in the system. Although we incur a performance penalty for accessing all vectors through software, the VM-DP paging system is designed to avoid disk I/O whenever possible.

The CVL functions that perform permutations in VM-DP bypass the demand paging system to choreograph the disk I/O on their own. As we shall see in Section 3.1, VM-DP performs general permutations by sorting according to target addresses using an external radix sort algorithm. Moreover, VM-DP can also detect and perform certain special permutations. It is worthwhile to do so because, as Chapters 2 and 3 will show, we can perform some special permutations that arise frequently much faster than we can perform general permutations. The algorithms for these special permutations require complete control over the disk accesses. More important, they require as much RAM as possible for their own use. We shall see in Chapters 2 and 3 that the number of disk accesses required by permutation algorithms decreases as the RAM size M increases. These algorithms, therefore, disable the demand paging system and clear out RAM so they can use all of it.

1.6 Contributions of this thesis

This thesis studies issues that arise in developing VM systems for data-parallel computing. The principal contributions of this thesis are the following:

- A system, VM-DP, for data-parallel computing with virtual memory.
- Theoretical results for classes of bit-defined permutations on parallel disk systems. Chapter 2 presents efficient, practical algorithms for performing bit-permute/complement (BPC) and bit-matrix-multiply/complement (BMMC) permutations. These permutation classes are generalizations of the matrix-transpose permutation, and they also include such common permutations as bit-reversal, hypercube, vector-reversal, Gray code, and inverse Gray code. Chapter 2 defines additional classes of permutations that can be performed in only one pass over records. A lower-bound argument in Chapter 2 proves that the BPC algorithm is asymptotically optimal and that the BMMC algorithm is optimal in some cases.
- Practical methods for performing permutations. Chapter 3 presents how VM-DP performs general permutations and lists several special permutations that can be performed faster: monotonic routes, mesh and torus permutations, BPC and BMMC permutations, and general matrix transpose. Moreover, each of these special permutations can be detected efficiently at run time.

  Nevertheless, Chapter 3 argues that the best way to handle special permutations is to invoke them in the source code. Perhaps the best reason is that source-level specification obviates the potentially high cost of generating target addresses.
- A study of how the layout of vectors on disk affects data-parallel operations. Chapter 4 presents a parameterized layout method. In this method, called banded layout, we divide a vector into bands of a number of consecutive vector elements laid out in column-major

order. Chapter 4 shows how the band size affects the performance of data-parallel operations.

- A comparison of alternative demand paging strategies in VM-DP. Chapter 5 contains an empirical study of three demand-paging strategies that were implemented as part of VM-DP and concludes that straight LRU paging, the simplest of the three schemes, is best overall.

It is important to understand what this thesis is not. It is not the last word in VM for data-parallel computing. It leaves many open questions. Moreover, this thesis does not address issues in languages and compilers, which may ultimately decide the success or failure of virtual memory for data-parallel computing. Chapter 6 discusses the future of VM for data-parallel computing.

Chapter 2

Performing Bit-Defined Permutations on Parallel Disk Systems

Data-parallel computations must often permute the elements of a vector. This type of operation is expensive enough when the vector fits in RAM, but the cost increases even more when the vector must reside on disk. This chapter presents efficient, practical algorithms to perform many common classes of permutations on vectors larger than the RAM size.

This work was motivated by the observation that there is a gap in the Vitter-Shriver results for permuting (see [VS90a, VS90b] or Section 1.4). Vitter and Shriver showed that although general permutations on N elements require $\Theta\left(\min\left(\frac{N}{D}, \frac{N}{BD}\frac{\lg(N/B)}{\lg(M/B)}\right)\right)$ parallel I/Os, a specific class of permutations, transposing an $R \times S$ matrix, can be performed using only $\Theta\left(\frac{N}{BD}\left(1 + \frac{\lg\min(R,S,B,N/B)}{\lg(M/B)}\right)\right)$ parallel I/Os. This gap raised the question of which related permutation classes can also be performed faster than general permutations.

This chapter presents an asymptotically optimal algorithm for the class of bit-permute/complement (BPC) permutations. In a BPC permutation, each target address is formed by permuting the bits of its corresponding source address according to a fixed bit permutation and then complementing a fixed subset of the resulting bits. The BPC algorithm uses at most $\frac{2N}{BD}\left(2\left\lceil\frac{\rho(A)}{\lg(M/B)}\right\rceil + 1\right)$ parallel I/Os, where $\rho(A)$, which we refer to as the "cross-rank" of the permutation, is a measure of the bit movement of the bit permutation. The class of BPC permutations includes matrix transpose as a subclass, and it includes other common permutations, such as bit-reversal, hypercube, and vector-reversal. The algorithm herein is not only asymptotically optimal, but its small constant factors render it very practical.

Just as BPC permutations generalize matrix transpose, the class of bit-matrix-multiply/

Most of the material in this chapter appears in [Cor92].

complement (BMMC) permutations generalizes BPC permutations. In a BMMC permutation, each target address is formed by multiplying its corresponding source address, treated as a vector, by a nonsingular "characteristic" matrix over $GF(2)$ and then complementing a fixed subset of the resulting bits. This chapter presents an algorithm that performs BMMC permutations using at most

$$\frac{2N}{BD}\left(2\left\lceil\frac{\lg M - \operatorname{rank}(A_{0..\lg M-1,0..\lg M-1})}{\lg(M/B)}\right\rceil + H(N,M,B)\right)$$

parallel I/Os, where

$$H(N,M,B) = \begin{cases} 4\left\lceil\dfrac{\lg B}{\lg(M/B)}\right\rceil + 9 & \text{if } M \le \sqrt{N}\ , \\ 4\left\lceil\dfrac{\lg(N/B)}{\lg(M/B)}\right\rceil + 1 & \text{if } \sqrt{N} < M < \sqrt{NB}\ , \\ 5 & \text{if } \sqrt{NB} \le M\ , \end{cases}$$

and where $A_{0..\lg M-1,0..\lg M-1}$ is the leading $(\lg M) \times (\lg M)$ submatrix of the characteristic matrix A. BMMC permutations include all BPC permutations and classes such as Gray code and inverse Gray code permutations. As we shall see, the BMMC algorithm is asymptotically optimal in some cases but not in others.

Not only are the algorithms for BPC and BMMC permutations efficient, but they are also deterministic, easily programmed, and can be performed "on-line" in the sense that they take little time and space. The data structures are vectors of length $\lg N$ or $\lg N \times \lg N$ matrices, and serial algorithms for the harder computations take time polynomial in $\lg N$.

This chapter focuses on minimizing the number of parallel I/O operations rather than on processing time. We will not examine the in-RAM permutations performed by the algorithms, primarily because permutation algorithms for parallel machines are highly dependent on aspects of the machine architecture beyond the scope of this thesis. Examples of the types of in-RAM permutation algorithms one might use appear in [EHJ92, JH91, NS81, NS82].

To perform BPC and BMMC permutations, we define some restricted classes of permutations that we can perform in one pass each. We perform a BPC permutation by factoring it into the composition of these restricted permutations. We do the same for BMMC permutations, but with BPC permutations among the factors.

This chapter includes the BPC and BMMC algorithms and much more. Section 2.1 formally defines the classes of bit-defined permutations that we work with in this chapter. It presents examples of these classes and discusses some of their general properties. Section 2.2 contains the BPC algorithm, proves it correct, and analyzes its I/O requirements. Section 2.3 describes a technique that Sections 2.4 and 2.5 use to perform the restricted classes of permutations in one pass each. Using results from the previous sections, we see how to perform BMMC permutations quickly in Section 2.6. Section 2.7 presents an off-line algorithm for arbitrary block permutations. In Section 2.8, we prove that our BPC algorithm is asymptotically optimal and prove a lower bound for BMMC permutations. The algorithms in this chapter assume a particular style of data layout on the parallel disk system. Section 2.9 shows that when other data-layout organizations are used, the BPC and BMMC algorithms in this chapter need not be changed; we need only adjust the input permutations slightly. Finally, Section 2.10 contains some concluding remarks about performing bit-defined permutations.

2.1 Classes of permutations

This section defines the classes of permutations that appear in the remainder of this chapter.[1] We start by defining some notation and the style of data layout assumed by the permutation algorithms.

Notation and data layout

For convenience, this chapter uses the following notation extensively:

$$\begin{aligned} b &= \lg B \,, \\ d &= \lg D \,, \end{aligned}$$

[1] We'll also see BPC and BMMC permutations in Chapter 3.

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

Figure 2.1: The layout of $N = 64$ records in a parallel disk system with $B = 2$ and $D = 8$. Each box is one block. The number of tracks needed is $N/BD = 4$. Numbers indicate record indices.

$$\begin{aligned} m &= \lg M \ , \\ t &= \lg(N/BD) \ , \\ n &= \lg N \ . \end{aligned}$$

Because B, D, and M are assumed to be exact powers of 2 thoughout this thesis, b, d, and m are nonnegative integers. As usual, N is the number of elements of the vector that we wish to permute. In this chapter, we assume that N is an exact power of 2, and so n is a nonnegative integer. We also assume that $N > M$, for otherwise we can perform any permutation easily by reading the entire vector into RAM, permuting it, and writing it back out to the parallel disk system. Note that $n = b + d + t$ and that, since $BD \leq M < N$, we have $b + d \leq m < n$.

This chapter follows the Vitter-Shriver scheme for vector layout on the parallel disk system, as shown in Figure 2.1. Record indices vary most rapidly within a block, then among disks, and finally among tracks. Thus, we can parse the address of a record as shown in Figure 2.2: the least significant b bits give the offset of the record within its block, the next most significant d bits give the number of the disk that the record resides on, and the most significant t bits give the number of the track containing the record. We indicate the address of a record as an n-bit vector x with the least significant bit first: $x = (x_0, x_1, \ldots, x_{n-1})$. The offset is given by the b bits $x_0, x_1, \ldots, x_{b-1}$, the disk number by the d bits $x_b, x_{b+1}, \ldots, x_{b+d-1}$, and the track number by the t bits $x_{b+d}, x_{b+d+1}, \ldots, x_{n-1}$. Figure 2.2 also shows the relationship of m to b, d, and t, and the bit positions $x_b, x_{b+1}, \ldots, x_{m-1}$, which will be important later on.

A permutation of records is a one-to-one mapping of input addresses from the set $\{0, 1, \ldots, N-1\}$ onto itself. We specify a permutation by considering each record's *source address*

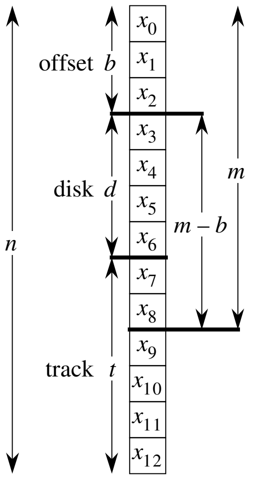


Figure 2.2: Parsing the address $x = (x_0, x_1, \ldots, x_{n-1})$ of a record. Here, $n = 13$, $b = 3$, $d = 4$, $t = 6$, and $m = 9$. The least significant b bits contain the offset of a record within its block, the next d bits contain the disk number, and the most significant $t = n - (b + d)$ bits contain the track number. The least significant m bits and bits $x_b, x_{b+1}, \ldots, x_{m-1}$ are also indicated.

$x = (x_0, x_1, \ldots, x_{n-1})$ and its corresponding *target address* $y = (y_0, y_1, \ldots, y_{n-1})$. Table 2.1 shows the classes of permutations we study in this chapter and summarizes the upper-bound results herein.

BPC permutations

In the class of *bit-permute/complement*, or *BPC*, permutations[2] we form each record's target address y from its source address x by applying a fixed permutation π to the address bits and then complementing a fixed subset of bits of the result. The complementing is equivalent to exclusive-oring by an n-bit *complement vector* $c = (c_0, c_1, \ldots, c_{n-1})$. A source address x maps to a target address y by the equation

$$y_{\pi(j)} = x_j \oplus c_{\pi(j)} \tag{2.1}$$

for $j = 0, 1, \ldots, n-1$, where $\oplus$ denotes the exclusive-or (XOR) operation.

[2] Johnsson and Ho [JH91] call BPC permutations *dimension permutations*, and Aggarwal, Chandra, and Snir [ACS87] call BPC permutations without complementing *rational permutations.*

Permutation	Characteristic matrix	Number of passes
BPC (bit-permute/ complement)	permutation matrix A	$2\left\lceil \frac{\rho(A)}{\lg(M/B)} \right\rceil + 1$
BMMC (bit-matrix-multiply/ complement)	nonsingular matrix A	$2\left\lceil \frac{\lg M - r}{\lg(M/B)} \right\rceil + H(N,M,B)$
Block BMMC	$\begin{array}{ccc} b & n-b & \\ \text{nonsingular} & 0 & b \\ 0 & \text{nonsingular} & n-b \end{array}$	1
Block BPC	$\begin{array}{ccc} b & n-b & \\ \text{nonsingular} & 0 & b \\ 0 & \text{permutation} & n-b \end{array}$	1
MRC (memory-rearrangement/ complement)	$\begin{array}{ccc} m & n-m & \\ \text{nonsingular} & \text{arbitrary} & m \\ 0 & \text{nonsingular} & n-m \end{array}$	1
Arbitrary block		1 (off-line scheduling)

Table 2.1: Classes of permutations, their characteristic matrices, and upper bounds shown in this chapter on the number of passes needed to perform them. A pass uses exactly $2N/BD$ parallel I/Os. All characteristic matrices are $n \times n$. For block BMMC, block BPC, and MRC permutations, submatrix dimensions are shown on matrix borders. For BPC permutations, the function $\rho(A)$ is defined in equation (2.6). For BMMC permutations, the term r is equal to $\text{rank}(A_{0..\lg M-1,0..\lg M-1})$, and the function $H(N,M,B)$ is given by equation (2.18). Each of these classes, with the exception of arbitrary block permutations, can be performed on-line.

Many familiar permutations fit into the class of BPC permutations. For example, matrix transposition is a type of BPC permutation. Suppose that the N records are the entries of an $R \times S$ matrix stored in row-major order. The n-bit address of each record is comprised of a $(\lg R)$-bit row number followed by a $(\lg S)$-bit column number. To transpose this matrix, we map the (i,j) entry to the (j,i) position. We can do so by performing the permutation in which the upper $\lg R$ bits and the lower $\lg S$ bits of the source address are swapped to form the target address. Here, the fixed permutation of address bits is a cyclic rotation by either $\lg R$ positions in one direction or $\lg S$ positions in the other, and no bits are complemented. (Cyclic rotation by one bit position is equivalent to the perfect shuffle and inverse perfect shuffle permutations.) In the context of equation (2.1), we have

$$\pi(j) = (j + \lg R) \bmod n = (j - \lg S) \bmod n \tag{2.2}$$

and $c_j = 0$ for $j = 0, 1, \ldots, n-1$.

Bit-reversal permutations, which are often used in performing FFTs, are another example of BPC permutations. Here, we reverse the bits of each record's source address to form its target address. Thus, if a record's n-bit source address is $(x_0, x_1, \ldots, x_{n-1})$, its target address is $(x_{n-1}, x_{n-2}, \ldots, x_0)$. In the context of equation (2.1), we have $\pi(j) = (n-1) - j$ and $c_j = 0$ for $j = 0, 1, \ldots, n-1$.

Vector-reversal permutations are yet another type of BPC permutation. In a vector-reversal permutation, the record with source address i has target address $(N-1)-i$ for $i = 0, 1, \ldots, N-1$. This permutation of records is accomplished by complementing each address bit. Here, $\pi(j) = j$ and $c_j = 1$ for $j = 0, 1, \ldots, n-1$. Complementing just one address bit yields a hypercube permutation, which corresponds to swapping records across one dimension of a hypercube.

We can also formulate a BPC permutation as a matrix-vector product followed by an XOR operation. We define an $n \times n$ permutation matrix[3] $A = (a_{ij})$, which is related to the address-bit permutation π of equation (2.1) by the relationship

$$a_{ij} = \begin{cases} 1 & \text{if } i = \pi(j) \ , \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

for $i, j = 0, 1, \ldots, n-1$. We treat a source address x as an n-vector (equivalent to an $n \times 1$ matrix, as in Figure 2.2). Using the same complement vector c as in equation (2.1), the same

[3]A permutation matrix has exactly one 1 in each row and exactly one 1 in each column.

target address y from equation (2.1) is given by $y = Ax \oplus c$, or

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1,n-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & a_{n-1,2} & \cdots & a_{n-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{bmatrix} \oplus \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{bmatrix} . \qquad (2.4)$$

We shall characterize BPC permutations by either a matrix or an address-bit permutation according to which is more convenient at the time.

BMMC permutations

The class of *bit-matrix-multiply/complement*, or *BMMC*, permutations[4] is a generalization of the class of BPC permutations. If the entries the matrix A in equation (2.4) are drawn from $\{0, 1\}$ and A is nonsingular (i.e., invertible) over $GF(2)$,[5] then equation (2.4) defines a permutation. (We shall prove this property later in this section.) Each target-address bit y_i is given by

$$y_i = \left(\bigoplus_{0 \le j \le n-1} a_{ij} x_j \right) \oplus c_i \ ,$$

where the product $a_{ij} x_j$ is simply the logical-and of the two bits. We call the matrix A the *characteristic matrix* of the BMMC permutation.

BMMC permutations include all BPC permutations, since any permutation matrix is nonsingular. The class of BMMC permutations also includes permutations such as the Gray code and inverse Gray code permutations. If $y = \text{Gray}(x)$, then

$$y_i = \begin{cases} x_i \oplus x_{i+1} & \text{if } 0 \le i < n-1 \ , \\ x_i & \text{if } i = n-1 \ . \end{cases}$$

[4]Edelman, Heller, and Johnsson [EHJ92] call BMMC permutations *affine transformations* or, if there is no complementing, *linear transformations*.

[5]Matrix multiplication over $GF(2)$ is like standard matrix multiplication over the reals but with all arithmetic performed modulo 2. Equivalently, multiplication is replaced by logical-and, and addition is replaced by XOR.

For example, if $N = 2^6$, the corresponding BMMC permutation is given by

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} .$$

If $y = \text{Gray}^{-1}(x)$, then

$$y_i = \bigoplus_{i \le j \le n-1} x_j \ ,$$

and as a BMMC permutation for $N = 2^6$, we have

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} .$$

BMMC permutations are also useful for reducing data-access conflicts in parallel systems. When data accesses are not to consecutive records, but rather occur with a stride s that is not relatively prime to the number of storage devices, many parallel references will access the same device. As a result, one device is a "hot spot"—overloaded with I/O requests—while the rest are idle. This situation can occur, for example, when accessing entire columns of matrices stored in row-major order. Norton and Melton [NM87] show how by rearranging the records according to BMMC permutations, the conflicts can be dramatically reduced. When the devices in question form a parallel disk system, rather than endure the data-access conflicts induced by the stride-s accesses, it can be faster to first permute the records according to some BMMC permutation with matrix A, then perform the stride-s accesses, and finally restore the original record ordering by performing the BMMC permutation with matrix A^{-1}. We refer the interested reader to [NM87] for details on computing the matrix A.

The next three permutation classes to be defined are useful as subroutines in performing

BMMC and BPC permutations. Each is a restricted BMMC permutation and is described by a characteristic matrix and a complement vector.

Block BMMC permutations

Block BMMC permutations are BMMC permutations with the restrictions shown in Table 2.1. We require that both the leading $b \times b$ submatrix and the trailing $(n-b) \times (n-b)$ submatrix be nonsingular and that the rest of the characteristic matrix be 0.[6] A block BMMC permutation can be thought of as a BMMC permutation in which the records are entire blocks; the trailing $(n-b) \times (n-b)$ submatrix gives this BMMC permutation. Moreover, information within the records can be reordered according to the leading $b \times b$ submatrix.

Block BPC permutations

Block BPC permutations are block BMMC permutations with the additional restriction shown in Table 2.1: the trailing $(n-b) \times (n-b)$ submatrix must be a permutation matrix. A block BPC permutation can be thought of as a BPC permutation in which the records are entire blocks and information within the records can be reordered according to the leading $b \times b$ submatrix. Any hypercube or vector-reversal permutation is trivially a block BPC permutation, since its characteristic matrix is an identity matrix.

MRC permutations

Table 2.1 shows the form of a *memory-rearrangement/complement*, or *MRC*, characteristic matrix. Both the leading $m \times m$ and trailing $(n-m) \times (n-m)$ submatrices are nonsingular, the upper right $m \times (n-m)$ submatrix can contain any 0-1 values at all, and the lower left $(n-m) \times m$ submatrix is all 0. As we shall see more formally in Section 2.4, an MRC permutation can be performed by reading in a *memoryload* (i.e., M records) of M/BD tracks, permuting the records read within memory, and writing them out to M/BD tracks. Gray code and inverse Gray code permutations are MRC permutations.

[6] A *leading* $R \times S$ submatrix of a matrix A consists of the intersection of the first R rows and S columns of A, and a *trailing* $R \times S$ submatrix consists of the intersection of the last R rows and S columns.

Note that BMMC and BPC permutations are independent of the machine parameters B and M, unlike block BMMC, block BPC, and MRC permutations. Given an $n \times n$ matrix, one can easily check whether it characterizes a BMMC or BPC permutation, but unless we know the block size B, we cannot check whether it characterizes a block BMMC or block BPC permutation. Similarly, we need to know the RAM size M to determine whether a matrix characterizes an MRC permutation.

Arbitrary block permutations

There is one more class of permutation that we shall study in this chapter, although we won't use it to perform other classes of permutations. In an *arbitrary block permutation*, all records within a source block remain together in a target block, and the blocks may be permuted arbitrarily. That is, the permutation on block addresses may be any mapping from $\{0, 1, \ldots, N/B - 1\}$ onto itself. This class includes block BMMC and block BPC permutations. Section 2.7 will show how to perform arbitrary block permutations off-line.

Further notational conventions

The remainder of this chapter will rely on a few more notational conventions. Matrix row and column numbers are indexed from 0 starting from the upper left. We index rows and columns by sets to indicate submatrices, using ". ." notation to indicate sets of contiguous numbers. For example, if $U = \{0, 1, 2\}$ and $V = \{1, 3\}$, then

$$A_{U,V} = A_{0..2,V} = \begin{bmatrix} a_{01} & a_{03} \\ a_{11} & a_{13} \\ a_{21} & a_{23} \end{bmatrix} .$$

When a submatrix index is a singleton set, we shall often omit the enclosing braces: $A_{i,j} = A_{\{i\},\{j\}}$. We denote an identity matrix by I and a matrix whose entries are all 0s by 0; the dimensions of such matrices will be clear from their contexts. We will frequently switch between bit vectors and their interpretations as base-2 integers; we interpret the vector $x = (x_0, x_1, \ldots, x_{n-1})$ as the integer $\sum_{i=0}^{n-1} x_i 2^i$. Vectors are treated as 1-column matrices in context.

Properties of the permutations

The reader might not find it obvious that if A is nonsingular, then matrix multiplication by A over $GF(2)$ yields a permutation of the source addresses. The following lemma and its corollary prove this property. For a $p \times q$ matrix A with 0-1 entries, we define

$$\mathcal{R}(A) = \{y : y = Ax \text{ for some } x \in \{0, 1, \ldots, 2^q - 1\}\} ,$$

that is, $\mathcal{R}(A)$ is the set of values that can be produced by multiplying $0, 1, \ldots, 2^q - 1$ by A over $GF(2)$. The *rank* of a matrix A, denoted $\operatorname{rank}(A)$, is the cardinality of the largest set of linearly independent rows or columns of A.

Lemma 2.1 *Let A be a $p \times q$ matrix whose entries are drawn from $\{0,1\}$, and let $r = \operatorname{rank}(A)$. Then $|\mathcal{R}(A)| = 2^r$.*

Proof: Let S index a maximal set of linearly independent columns of A, so that $S \subseteq \{0, 1, \ldots, q-1\}$, $|S| = r$, the columns of the submatrix $A_{0..p-1,S}$ are linearly independent, and for any column number $j \notin S$, the column $A_{0..p-1,j}$ is linearly dependent on the columns of $A_{0..p-1,S}$. By the definition of linear dependence, we have that $\mathcal{R}(A) = \mathcal{R}(A_{0..p-1,S})$. But $|\mathcal{R}(A_{0..p-1,S})| = 2^{|S|} = 2^r$, since each column index in S may or may not be included in a sum of the columns. Thus, $|\mathcal{R}(A)| = 2^r$. ■

Corollary 2.2 *Let A be a $q \times q$ matrix whose entries are drawn from $\{0,1\}$. Then A is nonsingular over $GF(2)$ if and only if $\mathcal{R}(A)$ is a permutation of $\{0, 1, \ldots, 2^q - 1\}$.*

Proof: If A is nonsingular, then it has rank q. Thus, $|\mathcal{R}(A)| = 2^q$, and each element of $\{0, 1, \ldots, 2^q - 1\}$ is represented in $\mathcal{R}(A)$, making it a permutation of $\{0, 1, \ldots, 2^q - 1\}$.

Conversely, if A is singular, then its rank is strictly less than q, which implies that $|\mathcal{R}(A)| < 2^q$. Thus, there is some number in $\{0, 1, \ldots, 2^q - 1\}$ that is not in $\mathcal{R}(A)$, so $\mathcal{R}(A)$ is not a permutation of $\{0, 1, \ldots, 2^q - 1\}$. ■

To show that equation (2.4) defines a permutation when the characteristic matrix is nonsingular, we also need to show that complementing by a fixed q-vector defines a permutation.[7]

Theorem 2.3 *Let c be a q-vector, and let $x, z \in \{0, 1, \ldots, q-1\}$. Then $x = z$ if and only if $x \oplus c = z \oplus c$.*

Proof: That $x = z$ implies $x \oplus c = z \oplus c$ is obvious. Now suppose that $x \neq z$. Let x and z differ in position i, for some $i \in \{0, 1, \ldots, q-1\}$. Then $x_i \oplus c_i \neq z_i \oplus c_i$, and so $x \oplus c \neq z \oplus c$. ■

Next, we show that block BMMC, block BPC, and MRC permutations are indeed permutations. To do so, we need only show that their characteristic matrices are nonsingular over $GF(2)$.

Theorem 2.4 *The characteristic matrices for block BMMC, block BPC, and MRC permutations are nonsingular.*

Proof: Let $A = \left[\begin{array}{c|c} \alpha & 0 \\ \hline 0 & \delta \end{array}\right]$ characterize a block BMMC permutation, where α, which is $b \times b$, and δ, which is $(n-b) \times (n-b)$, are both nonsingular submatrices. Then, $A^{-1} = \left[\begin{array}{c|c} \alpha^{-1} & 0 \\ \hline 0 & \delta^{-1} \end{array}\right]$, and so A is nonsingular.

The proof for block BPC permutations is similar, except that δ is a permutation matrix.

Now let $A = \left[\begin{array}{c|c} \alpha & \beta \\ \hline 0 & \delta \end{array}\right]$ characterize an MRC permutation, where α is $m \times m$ and nonsingular, β is any 0-1 submatrix that is $m \times (n-m)$, and δ is $(n-m) \times (n-m)$ and nonsingular. Then, $A^{-1} = \left[\begin{array}{c|c} \alpha^{-1} & \alpha^{-1}\beta\delta^{-1} \\ \hline 0 & \delta^{-1} \end{array}\right]$, and so A is nonsingular. ■

Cross-ranks

We will characterize the I/O complexity of BPC permutations by their *cross-ranks*, which we can define on the characteristic matrix or the equivalent address-bit permutation. Intuitively, for any bit position k, the *k-cross-rank*, denoted ρ_k, is the number of bits that cross a dividing

[7]Theorem 2.3 is essentially a special case of a cancellation lemma for groups such as Lemma 2.3.2 in [Her75, p. 34].

line between bit positions $k-1$ and k one way. More formally, for a bit permutation π or the equivalent characteristic matrix A, and for $k = 0, 1, \ldots, n-1$, we define

$$\begin{aligned}
\rho_k(\pi) &= |\{j : 0 \le j \le k-1 \text{ and } k \le \pi(j) \le n-1\}| \\
&= |\{j : k \le j \le n-1 \text{ and } 0 \le \pi(j) \le k-1\}| \ , \\
\rho_k(A) &= \operatorname{rank}(A_{0..k-1,k..n-1}) \\
&= \operatorname{rank}(A_{k..n-1,0..k-1}) \ , \qquad (2.5)
\end{aligned}$$

and we define the *cross-rank* ρ by

$$\begin{aligned}
\rho(\pi) &= \max(\rho_m(\pi), \rho_b(\pi)) \ , \\
\rho(A) &= \max(\rho_m(A), \rho_b(A)) \ . \qquad (2.6)
\end{aligned}$$

The cross-rank, therefore, is the maximum of the m- and b-cross-ranks and is thus dependent on the RAM size and the block size. Note that $\rho_k(\pi) = \rho_k(A)$ for all k, and thus $\rho(\pi) = \rho(A)$, when π and A are related by equation (2.3).

2.2 BPC permutations

In this section, we examine how to perform BPC permutations, which, as we have seen, include many commonly used permutations. We are given an $n \times n$ characteristic matrix A that is a permutation matrix and a complement vector c of length n. Our algorithm performs at most $\frac{2N}{BD}\left(2\left\lceil\frac{\rho(A)}{\lg(M/B)}\right\rceil + 1\right)$ parallel I/Os.

It is more convenient to think in terms of address-bit permutations than in terms of permutation matrices. Given a characteristic matrix A for a BPC permutation, we shall work with the equivalent address-bit permutation π that satisfies equation (2.3). For now, we ignore the complement part of the BPC permutation.

We shall perform BPC permutations by alternately performing MRC and block BPC permutations. As we shall see in Sections 2.4 and 2.5, we can perform any MRC or any block BPC permutation using exactly $2N/BD$ parallel I/Os. The characteristic matrices of these permutations have particular forms. For the MRC permutations, the leading $m \times m$ submatrix and the trailing $(n-m) \times (n-m)$ submatrix are permutation matrices. For the block BPC permutations, the leading $b \times b$ submatrix and the trailing $(n-b) \times (n-b)$ submatrix are permutation matrices.

The idea in performing BPC permutations is to sort the bits according to the permutation π. We maintain a running "remaining permutation" π_{rem} and a running "permutation performed so far" π_{perf}, updating them each time we perform an MRC or block BPC permutation. Initially, π_{perf} is the identity permutation and $\pi_{\text{rem}} = \pi$; the algorithm terminates when π_{rem} is the identity permutation, at which time $\pi_{\text{perf}} = \pi$.

We sort the π_{rem} permutation by alternately performing MRC and block BPC permutations. Intuitively, each MRC permutation sorts the first m items of π_{rem}, and each block BPC permutation sorts the last $n-b$ items. There is an $(m-b)$-sized area of overlap through which address bits can pass between the first b and last $n-m$ positions.

The pseudocode for the BPC algorithm uses the following conventions. The symbol $\circ$ indicates function composition—$(f \circ g)(j) = f(g(j))$—which is associative. The input to the procedure PERFORM-BPC is the permutation π to be performed, and the procedure's side effect is to perform π. Parameters are passed by reference, so that changes to the parameters π_{perf} and π_{rem} in the subroutines PERFORM-MRC and PERFORM-BLOCK-BPC are seen by PERFORM-BPC. Inspection of the code reveals that the permutation π_{perf} is not really needed; we include it to facilitate the proof of correctness below. Section 2.4 shows how to perform line 5 of PERFORM-MRC, and Section 2.5 shows how to perform line 5 of PERFORM-BLOCK-BPC.

```
PERFORM-BPC($\pi$)
1  $\pi_{\text{perf}} \leftarrow$ identity permutation
2  $\pi_{\text{rem}} \leftarrow \pi$
3  PERFORM-MRC($\pi_{\text{perf}}, \pi_{\text{rem}}$)
4  while $\pi_{\text{rem}}$ is not the identity permutation
5      do PERFORM-BLOCK-BPC($\pi_{\text{perf}}, \pi_{\text{rem}}$)
6         PERFORM-MRC($\pi_{\text{perf}}, \pi_{\text{rem}}$)
```

```
PERFORM-MRC($\pi_{\text{perf}}, \pi_{\text{rem}}$)
1  for $j \leftarrow 0$ to $m-1$
2      do $\pi_{\text{MRC}}(j) \leftarrow i$, where $\pi_{\text{rem}}(j)$ is the $i$th smallest member (starting from $i = 0$)
              of $\{\pi_{\text{rem}}(0), \pi_{\text{rem}}(1), \ldots, \pi_{\text{rem}}(m-1)\}$
3  for $j \leftarrow m$ to $n-1$
4      do $\pi_{\text{MRC}}(j) \leftarrow m + i$, where $\pi_{\text{rem}}(j)$ is the $i$th smallest member (starting from
              $i = 0$) of $\{\pi_{\text{rem}}(m), \pi_{\text{rem}}(m+1), \ldots, \pi_{\text{rem}}(n-1)\}$
5  perform the permutation $\pi_{\text{MRC}}$
6  $\pi_{\text{perf}} \leftarrow \pi_{\text{MRC}} \circ \pi_{\text{perf}}$
7  $\pi_{\text{rem}} \leftarrow \pi_{\text{rem}} \circ \pi_{\text{MRC}}^{-1}$
```

```
PERFORM-BLOCK-BPC($\pi_{\text{perf}}, \pi_{\text{rem}}$)
1  for $j \leftarrow 0$ to $b-1$
2      do $\pi_{\text{BlockBPC}}(j) \leftarrow i$, where $\pi_{\text{rem}}(j)$ is the $i$th smallest member (starting from
              $i = 0$) of $\{\pi_{\text{rem}}(0), \pi_{\text{rem}}(1), \ldots, \pi_{\text{rem}}(b-1)\}$
3  for $j \leftarrow b$ to $n-1$
4      do $\pi_{\text{BlockBPC}}(j) \leftarrow b + i$, where $\pi_{\text{rem}}(j)$ is the $i$th smallest member (starting from
              $i = 0$) of $\{\pi_{\text{rem}}(b), \pi_{\text{rem}}(b+1), \ldots, \pi_{\text{rem}}(n-1)\}$
5  perform the permutation $\pi_{\text{BlockBPC}}$
6  $\pi_{\text{perf}} \leftarrow \pi_{\text{BlockBPC}} \circ \pi_{\text{perf}}$
7  $\pi_{\text{rem}} \leftarrow \pi_{\text{rem}} \circ \pi_{\text{BlockBPC}}^{-1}$
```

At this point, an example is in order. Let $b = 6$, $m = 9$, and $n = 16$, and suppose that we have the following permutation π:

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi(j)$	10	7	14	8	2	13	11	15	9	3	12	0	5	4	1	6

(Vertical lines separate positions $\{0, 1, \ldots, b-1\}$, $\{b, b+1, \ldots, m-1\}$, and $\{m, m+1, \ldots, n-1\}$.) For the first call of Perform-MRC, we have

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi_{\text{perf}}(j)$ before	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi_{\text{rem}}(j)$ before	10	7	14	8	2	13	11	15	9	3	12	0	5	4	1	6
$\pi_{\text{MRC}}(j)$	4	1	7	2	0	6	5	8	3	11	15	9	13	12	10	14
$\pi_{\text{perf}}(j)$ after	4	1	7	2	0	6	5	8	3	11	15	9	13	12	10	14
$\pi_{\text{rem}}(j)$ after	2	7	8	9	10	11	13	14	15	0	1	3	4	5	6	12

and we perform the MRC permutation π_{MRC}. In the first iteration of the **while** loop, we call Perform-Block-BPC and Perform-MRC, with the following effects:

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi_{\text{perf}}(j)$ before	4	1	7	2	0	6	5	8	3	11	15	9	13	12	10	14
$\pi_{\text{rem}}(j)$ before	2	7	8	9	10	11	13	14	15	0	1	3	4	5	6	12
$\pi_{\text{BlockBPC}}(j)$	0	1	2	3	4	5	13	14	15	6	7	8	9	10	11	12
$\pi_{\text{perf}}(j)$ in between	4	1	14	2	0	13	5	15	3	8	12	6	10	9	7	11
$\pi_{\text{rem}}(j)$ in between	2	7	8	9	10	11	0	1	3	4	5	6	12	13	14	15
$\pi_{\text{MRC}}(j)$	2	4	5	6	7	8	0	1	3	9	10	11	12	13	14	15
$\pi_{\text{perf}}(j)$ after	7	4	14	5	2	13	8	15	6	3	12	0	10	9	1	11
$\pi_{\text{rem}}(j)$ after	0	1	2	3	7	8	9	10	11	4	5	6	12	13	14	15

and we perform the permutations π_{BlockBPC} and π_{MRC}. The second iteration yields

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi_{\text{perf}}(j)$ before	7	4	14	5	2	13	8	15	6	3	12	0	10	9	1	11
$\pi_{\text{rem}}(j)$ before	0	1	2	3	7	8	9	10	11	4	5	6	12	13	14	15
$\pi_{\text{BlockBPC}}(j)$	0	1	2	3	4	5	9	10	11	6	7	8	12	13	14	15
$\pi_{\text{perf}}(j)$ in between	10	4	14	5	2	13	11	15	9	3	12	0	7	6	1	8
$\pi_{\text{rem}}(j)$ in between	0	1	2	3	7	8	4	5	6	9	10	11	12	13	14	15
$\pi_{\text{MRC}}(j)$	0	1	2	3	7	8	4	5	6	9	10	11	12	13	14	15
$\pi_{\text{perf}}(j)$ after	10	7	14	8	2	13	11	15	9	3	12	0	5	4	1	6
$\pi_{\text{rem}}(j)$ after	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

and we again perform π_{BlockBPC} and π_{MRC}. The algorithm then terminates, since the permutation π_{rem} is now the identity permutation.

Having presented the algorithm, we need to prove it correct, establish an upper bound on the number of parallel I/Os, and show how to extend it to the case in which the complement vector is nonzero.

Proof of correctness

Lemma 2.5 *When* PERFORM-BPC *terminates, it has performed the permutation π.*

Proof: We start by showing that between subroutine calls, PERFORM-BPC preserves the following invariant:

$$\pi_{\text{rem}} \circ \pi_{\text{perf}} = \pi \ .$$

Initially, π_{perf} is the identity permutation and $\pi_{\text{rem}} = \pi$, so that $\pi_{\text{rem}} \circ \pi_{\text{perf}} = \pi$. Now we consider the effects of calling the subroutines. Let π_{perf} and π_{rem} be the permutations before a subroutine call, and $\widehat{\pi}_{\text{perf}}$ and $\widehat{\pi}_{\text{rem}}$ be the permutations after the call. We have $\pi_{\text{rem}} \circ \pi_{\text{perf}} = \pi$ before the call. After calling PERFORM-MRC, we have

$$\widehat{\pi}_{\text{rem}} \circ \widehat{\pi}_{\text{perf}} \quad = \quad (\pi_{\text{rem}} \circ \pi_{\text{MRC}}^{-1}) \circ (\pi_{\text{MRC}} \circ \pi_{\text{perf}})$$

$$\begin{aligned}
&= \pi_{\text{rem}} \circ (\pi_{\text{MRC}}^{-1} \circ \pi_{\text{MRC}}) \circ \pi_{\text{perf}} \quad \text{(associativity of } \circ) \\
&= \pi_{\text{rem}} \circ \pi_{\text{perf}} \\
&= \pi \ .
\end{aligned}$$

After calling Perform-Block-BPC, we have

$$\begin{aligned}
\widehat{\pi}_{\text{rem}} \circ \widehat{\pi}_{\text{perf}} &= (\pi_{\text{rem}} \circ \pi_{\text{BlockBPC}}^{-1}) \circ (\pi_{\text{BlockBPC}} \circ \pi_{\text{perf}}) \\
&= \pi_{\text{rem}} \circ (\pi_{\text{BlockBPC}}^{-1} \circ \pi_{\text{BlockBPC}}) \circ \pi_{\text{perf}} \quad \text{(associativity of } \circ) \\
&= \pi_{\text{rem}} \circ \pi_{\text{perf}} \\
&= \pi \ .
\end{aligned}$$

Having proven that the invariant holds, we conclude that when π_{rem} is the identity permutation, $\pi_{\text{perf}} = \pi$. Since π_{perf} is the permutation performed so far, we have performed the desired bit permutation π. ■

Bounding the number of parallel I/Os

Lemma 2.6 Perform-BPC *achieves the termination condition that π_{rem} is the identity permutation after at most $\lceil \rho(\pi)/(m-b) \rceil$ iterations of the* **while** *loop.*

Proof: We start by observing that, as we noted above, Permute-BPC works by sorting the π_{rem} permutation. It calls Perform-MRC to sort the first m items and Perform-Block-BPC to sort the last $n-b$ items. There is an $(m-b)$-sized area of overlap. Thus, we can view Permute-BPC as sorting the numbers $\{0, 1, \ldots, n-1\}$ in an array of size n in which we alternately sort the first m numbers and the last $n-b$ numbers until the entire array is sorted. Once a number in $\{0, 1, \ldots, b-1\}$ is placed into one of the first b positions, it remains in the first b positions, since there aren't enough smaller numbers to push it out. Similarly, once a number in $\{m, m+1, \ldots, n-1\}$ is placed into one of the last $n-m$ positions, it remains in the last $n-m$ positions.

Let κ_m be the set of numbers that start in the first m positions but belong in the last $n-m$

positions, so that $\rho_m(\pi) = |\kappa_m|$. Consider the combined effect of calling PERFORM-MRC followed by calling PERFORM-BLOCK-BPC. First, PERFORM-MRC moves the $m - b$ largest numbers in the first m positions into the overlap area, and then PERFORM-BLOCK-BPC moves the members of κ_m that are in the overlap area into the last $n - m$ positions, where they will remain. Each time we call PERFORM-MRC and PERFORM-BLOCK-BPC, we move another $m - b$ members of κ_m into the last $n - m$ positions. The numbers so moved are drawn from the set κ_m as long as any remain in the first m positions. After $\lceil |\kappa_m| / (m - b) \rceil$ pairs of calls, therefore, all the numbers of κ_m have been moved into the last $n - m$ positions.

A similar argument shows that if κ_b is the set of numbers that start in the last $n-b$ positions but belong in the first b positions (so that $\rho_b(\pi) = |\kappa_b|$), after $\lceil |\kappa_b| / (m - b) \rceil$ pairs of calls of PERFORM-BLOCK-BPC and PERFORM-MRC, all the numbers in κ_b have been moved into the first b positions.

Therefore, one call of PERFORM-MRC followed by the calls of PERFORM-BLOCK-BPC and PERFORM-MRC made during $\lceil \max(|\kappa_m|, |\kappa_b|)/(m - b) \rceil$ iterations of the **while** loop moves all the numbers in κ_m and κ_b into their correct ranges. The sorts performed in the last call place all the numbers into their actual correct positions. Since $\rho(\pi) = \max(\rho_m(\pi), \rho_b(\pi)) = \max(|\kappa_m|, |\kappa_b|)$, we have proven the lemma. ■

Corollary 2.7 PERFORM-BPC *performs at most* $\frac{2N}{BD}\left(2\left\lceil \frac{\rho(\pi)}{\lg(M/B)} \right\rceil + 1\right)$ *parallel I/Os.*

Proof: Since $m - b = \lg(M/B)$, PERFORM-BPC makes at most $2\left\lceil \frac{\rho(\pi)}{\lg(M/B)} \right\rceil + 1$ calls of PERFORM-MRC and PERFORM-BLOCK-BPC. Each call performs $2N/BD$ parallel I/Os. ■

Theorem 2.8 *A BPC permutation with characteristic matrix A and complement vector c can be performed using at most* $\frac{2N}{BD}\left(2\left\lceil \frac{\rho(A)}{\lg(M/B)} \right\rceil + 1\right)$ *parallel I/Os.*

Proof: Letting π be the permutation for which A is the permutation matrix, Lemma 2.5 and Corollary 2.7 prove the theorem for cases in which the complement vector c is all 0. To handle cases in which $c \neq 0$, we simply modify PERFORM-BPC so that the last call of PERFORM-MRC also complements according to c. ■

2.3 Permutable sets of blocks

In this section, we describe the technique of decomposing a permutation into permutable sets of blocks. We shall use this technique in later sections to perform MRC and block BMMC permutations.

We assume that the records to be permuted initially reside in areas on their disks that we call the *source portion* of the parallel disk system. Each disk holds N/D records. We also assume that each disk has enough spare storage to hold another copy of the data it holds, that is, that in addition to the N/D records initially on each disk, there is room for another N/D records as well; we call this spare storage area the *target portion*. The track number in a record address is relative to the beginning of the disk portion in which the record resides. A *pass* consists of repeatedly reading *source blocks* from the source portion of each disk into RAM, permuting the records of these blocks in RAM, and then writing the records out in *target blocks* to the target portion of each disk. The roles of the source and target portions can then be reversed for the next pass.

Each record, and thus each block, is read exactly once and written exactly once during a pass. One pass, therefore, performs exactly $2N/BD$ parallel I/Os.

To perform a pass, we need to decompose the permutation into disjoint sets of blocks that we can read in k blocks per disk at a time, permute their records within RAM, and then write out k blocks per disk. For a permutation of source addresses to target addresses and a positive integer k, we define a *k-permutable* set of blocks as a set of kD source blocks such that

1. each disk contains exactly k of these source blocks,
2. the kBD records in the source blocks are mapped to exactly kD full target blocks, and
3. each disk has exactly k of these target blocks mapped to it.

Any permutation that can be decomposed into disjoint k-permutable sets of blocks can be performed in one pass as long as $k \leq M/BD$, so that each set fits in RAM.

We call such a sequence of N/kBD reads and writes of k-permutable sets of blocks a *schedule*. In Sections 2.4 and 2.5, we shall see how to devise schedules for MRC and block BMMC permutations.

2.4 MRC permutations

In this section, we show how to perform any MRC permutation in just one pass, thus using exactly $2N/BD$ parallel I/Os. We shall do so by decomposing the permutation into M/BD-permutable sets of blocks to form a schedule.

The schedule is simply comprised of the following N/M sets:

$$\begin{gathered} \{0,1,\ldots,M-1\}\ , \\ \{M,M+1,\ldots,2M-1\}\ , \\ \vdots \\ \{N-M,N-M+1,\ldots,N-1\}\ . \end{gathered} \tag{2.7}$$

Theorem 2.9 *The sets (2.7) form a schedule for any MRC permutation. Therefore, any MRC permutation can be performed in one pass.*

Proof: The sets (2.7) obviously partition all N source addresses. Moreover, each set consists of M/BD source tracks, and hence each set contains exactly M/BD source blocks per disk. To show that the sets (2.7) form a schedule, we need to show that for each set, the M source records map to exactly M/B target blocks and that each disk has exactly M/BD of these target blocks mapped to it.

It suffices to show that any pair of source addresses in the same memoryload are mapped to the same memoryload by any MRC permutation. That is, we wish to show that if x and z are two distinct source addresses in the kth set of (2.7) and their corresponding target addresses are y and w, then $y_{m..n-1} = w_{m..n-1}$. (Since an MRC mapping is a permutation, it must also follow that $y_{0..m-1} \neq w_{0..m-1}$.)

Consider two distinct source addresses x and z in the kth set. That is, $x = kM + i$ and $z = kM + j$, where $0 \le k \le N/M - 1$, $0 \le i, j \le M - 1$, and $i \ne j$. Then

$$x_{0..m-1} = i \ne j = z_{0..m-1} \tag{2.8}$$

and

$$x_{m..n-1} = z_{m..n-1} = k \ . \tag{2.9}$$

Let $y = Ax \oplus c$ and $w = Az \oplus c$ be the corresponding target addresses for an MRC permutation with characteristic matrix A and complement vector c. An MRC permutation maps a source address x' to a target address y' by the equation

$$\left[\begin{array}{c} y'_{0..m-1} \\ \hline y'_{m..n-1} \end{array}\right] = \left[\begin{array}{c|c} \alpha & \beta \\ \hline 0 & \delta \end{array}\right] \left[\begin{array}{c} x'_{0..m-1} \\ \hline x'_{m..n-1} \end{array}\right] \oplus \left[\begin{array}{c} c_{0..m-1} \\ \hline c_{m..n-1} \end{array}\right] ,$$

where α is $m \times m$ and nonsingular and δ is $(n-m) \times (n-m)$ and nonsingular. Thus,

$$y'_{0..m-1} = \alpha\, x'_{0..m-1} \oplus \beta\, x'_{m..n-1} \oplus c_{0..m-1} \ , \tag{2.10}$$

$$y'_{m..n-1} = \delta\, x'_{m..n-1} \oplus c_{m..n-1} \ . \tag{2.11}$$

By equations (2.9) and (2.11), therefore, we have $y_{m..n-1} = w_{m..n-1}$, which completes the proof. Note also that because the submatrix α in equation (2.10) is nonsingular, equation (2.8) implies that $y_{0..m-1} \ne w_{0..m-1}$. ■

In one pass, therefore, we can step through the data one memoryload at a time by reading in a memoryload from the source portion, permuting it in RAM, and writing it back out, although to possibly a different set of track numbers in the target portion.

For some MRC permutations, it may be possible to form k-permutable sets of blocks for some value of k that is strictly less than M/BD. But $k = M/BD$ always works for MRC permutations. Moreover, in practice it is often more efficient to read and write contiguous sets of tracks at once than to break the I/O into several pieces.

2.5 Block BMMC and block BPC permutations

In this section, we show how to perform any block BMMC permutation in just one pass, using exactly $2N/BD$ parallel I/Os. We also show how some of the mathematics of the block-BMMC algorithm can be avoided for the more restricted class of block BPC permutations.

The algorithm for block BMMC permutations is easier to understand if we assume for now that $B = 1$, and thus $b = 0$ and $n = d + t$. That is, we assume that blocks contain just one record each and that we perform a BMMC permutation on them. We are given an $n \times n$ characteristic matrix A and a complement vector c of length n.

The algorithm has two parts. First, we find a 1-permutable set of blocks. Second, we construct a schedule for the entire permutation given any 1-permutable set of blocks. After presenting these two parts, we shall see how to remove the restriction that $B = 1$.

2.5.1 Finding a 1-permutable set of blocks

Our description of finding a 1-permutable set of blocks starts with a simple observation: we can ignore all but the first d rows of the characteristic matrix A and all but the first d positions of the complement vector c. Why? The last $n - d$ rows of A and positions of c determine only target-address track numbers, which do not affect 1-permutability.

We find a 1-permutable set of blocks in four steps:

1. Find a set S of d "basis" columns for the first d rows of A.

2. Given the basis set S, define three sets of columns T, U, and V.

3. Based on the sets T and U, define a permutation R on the set $\{0, 1, \ldots, D - 1\}$.

4. Given all of the above, define a set of source addresses $\{x^{(0)}, x^{(1)}, \ldots, x^{(D-1)}\}$, which constitutes a 1-permutable set of blocks.

After describing these steps, we shall then prove that the set $\{x^{(0)}, x^{(1)}, \ldots, x^{(D-1)}\}$ is indeed a 1-permutable set of blocks. We use a running example to help describe the four steps.

Finding a set S of basis columns

We start by finding a set S of d columns such that the submatrix $A_{0..d-1,S}$ is nonsingular. Such a set exists by the following argument. Because A is nonsingular, all of its rows are linearly independent. In particular, rows $0, 1, \ldots, d-1$ are. Thus, the submatrix $A_{0..d-1,0..n-1}$ has full row rank. Hence, there exists a subset of column indices $S \subseteq \{0, 1, \ldots, n-1\}$ such that $|S| = d$ and the $d \times d$ submatrix $A_{0..d-1,S}$ is nonsingular. Let us define

$$Q = A_{0..d-1,S} \ ,$$

so that Q^{-1} exists.

The basis S can be determined by performing Gaussian elimination to create an LUP decomposition of A^{T}. (See [CLR90, pp. 749–761] for example.) The sequential running time of Gaussian elimination is $\Theta(n^3) = \Theta(\lg^3 N)$. In an LUP decomposition of A^{T}, we compute a permutation matrix P, a unit lower triangular matrix[8] L, and an upper triangular matrix U such that $PA^{\mathrm{T}} = LU$. The permutation matrix P specifies the basis S, as we shall show. Since $P^{-1} = P^{\mathrm{T}}$ for any permutation matrix P, we have

$$(PA^{\mathrm{T}})^{\mathrm{T}} = AP^{\mathrm{T}} = AP^{-1} \ .$$

Since transposing a matrix does not affect its invertibility, for any k, the leading $k \times k$ submatrix of PA^{T} is nonsingular if and only if the leading $k \times k$ submatrix of AP^{-1} is nonsingular. Because P^{-1} is a permutation matrix, the product AP^{-1} consists of the matrix A with its columns rearranged. In particular, if $(P^{-1})_{i,j} = 1$, then the ith column of A is the jth column of AP^{-1}. If we can show that the leading $d \times d$ submatrix of PA^{T} is nonsingular, then by letting

$$S = \{i : (P^{-1})_{i,j} = 1 \text{ for some } 0 \leq j \leq d-1\} \ ,$$

[8]A *lower triangular matrix* has all 0s above the main diagonal, and a *upper triangular matrix* has all 0s below the main diagonal. A *unit triangular matrix* is a triangular matrix with 1s along the main diagonal.

we will have our basis.

We make use of the following facts from linear algebra.[9]

Lemma 2.10 *Any triangular matrix whose diagonal elements are all nonzero is nonsingular.* ■

Lemma 2.11 *Any leading submatrix of a nonsingular triangular matrix is nonsingular.* ■

Lemma 2.12 *In an LUP decomposition $PA^{\mathrm{T}} = LU$ of a nonsingular matrix A^{T}, the upper triangular matrix U is nonsingular.* ■

We now show that the leading $d \times d$ submatrix of PA^{T} is nonsingular, thus obtaining our basis.

Theorem 2.13 *In an LUP decomposition $PA^{\mathrm{T}} = LU$ of a nonsingular matrix A^{T}, any leading square submatrix of PA^{T} is nonsingular.*

Proof: We shall show that for any k, the leading $k \times k$ submatrix of PA^{T} is nonsingular. Let $X = PA^{\mathrm{T}}$, so that $X = LU$. Let us divide X, L, and U to separate out their leading $k \times k$ submatrices χ, λ, and υ, respectively:

$$\left[\begin{array}{c|c} \chi & \beta \\ \hline \gamma & \delta \end{array}\right] = \left[\begin{array}{c|c} \lambda & 0 \\ \hline \mu & \xi \end{array}\right] \left[\begin{array}{c|c} \upsilon & \sigma \\ \hline 0 & \tau \end{array}\right] .$$

The submatrix λ is unit lower triangular, so by Lemma 2.10, it is nonsingular. By Lemmas 2.11 and 2.12, the submatrix υ is nonsingular, too. Since $\chi = \lambda\,\upsilon$, the leading $k \times k$ submatrix χ is nonsingular. ■

Running example: As an example, let $n = 5$ and $d = 3$, and consider the characteristic matrix

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix} .$$

[9] Lemmas 2.10–2.12 and Theorem 2.13 apply to standard matrices over the reals as well.

We choose the basis $S = \{1, 3, 4\}$, so that

$$Q = A_{0..d-1,S} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix} \qquad \text{and} \qquad Q^{-1} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} .$$

Defining the sets T, U, and V

Given a set of basis columns S, we define

$$\begin{aligned} T &= \{0, 1, \ldots, d-1\} - S \ , \\ U &= S \cap \{0, 1, \ldots, d-1\} \ , \\ V &= \{0, 1, \ldots, n-1\} - (S \cup T) \ . \end{aligned}$$

The sets T and U form a partition of $\{0, 1, \ldots, d-1\}$, and the sets S, T, and V form a partition of $\{0, 1, \ldots, n-1\}$. Think of T as the subset of the first d columns that are not in the basis S and of U as the set of basis columns that are also among the first d columns. We order T and U into increasing order (we don't need to order V) so that

$$\begin{aligned} T = \{t_0, t_1, \ldots, t_{|T|-1}\} \ , \quad & t_j > t_{j-1} \quad \text{for } j = 1, 2, \ldots, |T| - 1 \\ U = \{u_0, u_1, \ldots, u_{|U|-1}\} \ , \quad & u_j > u_{j-1} \quad \text{for } j = 1, 2, \ldots, |U| - 1 \ . \end{aligned}$$

Running example: With $S = \{1, 3, 4\}$, we have $T = \{0, 2\}$, $U = \{1\}$, and $V = \emptyset$.

Defining the permutation R

For the sake of clarity, we shall be more explicit in the remainder of this section about conversions between integers and their binary representations. Let us denote the d-bit representation of an integer $i \in \{0, 1, \ldots, D-1\}$ by $\text{bin}(i)$; this representation can be thought of as a vector of length d. Let us also denote the jth bit of this representation, for $j = 0, 1, \ldots, d-1$, by $\text{bin}_j(i)$. (Thus, $\text{bin}_j(i) = \lfloor i/2^j \rfloor \bmod 2$.) We extend this notation from individual bits to sets of bits in the natural way.

We define the permutation R on $\{0, 1, \ldots, D-1\}$ as follows. Let $i \in \{0, 1, \ldots, D-1\}$, and let the binary representation of i be $\text{bin}(i) = (i_0, i_1, \ldots, i_{d-1})$. Then $R(i) = k$, where the binary representation of k is $\text{bin}(k) = (k_0, k_1, \ldots, k_{d-1})$ and

$$\begin{aligned} k_j &= i_{u_j} \quad \text{for } j = 0, 1, \ldots, |U|-1 \ , \\ k_{|U|+j} &= i_{t_j} \quad \text{for } j = 0, 1, \ldots, |T|-1 \ . \end{aligned}$$

Because R is a BPC permutation on $\text{bin}(i)$, it defines a permutation on $\{0, 1, \ldots, D-1\}$. Note that R is constructed so that

$$\text{bin}_{0..|U|-1}(R(i)) = \text{bin}_U(i) \tag{2.12}$$

for $i = 0, 1, \ldots, D-1$.

Running example: Since $d = 3$, we have $D = 8$. One way to see how to form the permutation R is to write out 0 through $D-1$ in binary and label each of the d columns of bits according to whether it belongs to T or U. Then rearrange the columns so that all the columns of U precede all the columns of T:

	T	U	T
i	0	1	2
0	0	0	0
1	1	0	0
2	0	1	0
3	1	1	0
4	0	0	1
5	1	0	1
6	0	1	1
7	1	1	1

$\Longrightarrow$

U	T	T	
1	0	2	$R(i)$
0	0	0	0
0	1	0	2
1	0	0	1
1	1	0	3
0	0	1	4
0	1	1	6
1	0	1	5
1	1	1	7

Defining the set of source addresses $\{\boldsymbol{x}^{(0)}, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(D-1)}\}$

We define the set $\{x^{(0)}, x^{(1)}, \ldots, x^{(D-1)}\}$ according to bit indices:

$$\begin{aligned} x_S^{(i)} &= Q^{-1} A_{0..d-1,T} \operatorname{bin}_T(i) \oplus \operatorname{bin}(R(i)) \oplus c_{0..d-1} \ , && (2.13) \\ x_T^{(i)} &= \operatorname{bin}_T(i) \ , && (2.14) \\ x_V^{(i)} &= 0 && (2.15) \end{aligned}$$

for $i = 0, 1, \ldots, D-1$.

Running example: We let the complement vector c be all 0s to keep the example simple. Let us see how to compute $x^{(6)}$. We start by computing the bits in positions 1, 3, and 4, since $S = \{1, 3, 4\}$. Writing lower-order bits on the left, and indexing from zero, we have that $\operatorname{bin}(6) = 011$ and, taking bits 0 and 2 because $T = \{0, 2\}$, we have $\operatorname{bin}_T(6) = 01$. From above, we have $\operatorname{bin}(R(6)) = 101$, and so

$$\begin{aligned} x_{\{1,3,4\}}^{(6)} &= Q^{-1} A_{0..2,T} \operatorname{bin}_T(6) \oplus \operatorname{bin}(R(6)) \\ &= \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} . \end{aligned}$$

Filling these values into the positions indexed by set S in the source address, we have that $x^{(6)} = ?0?01$, where the positions marked by question marks correspond to set T. For these remaining positions, we use the bits of $\operatorname{bin}_T(6) = 01$, and so $x^{(6)} = 00101$. We also have that $y^{(6)} = Ax^{(6)} = 11110$.

If we were to do this calculation for $i = 0, 1, \ldots, 7$, we would compute the following source and target addresses:

	0	1	2	3	4
$x^{(0)}$	0	0	0	0	0
$x^{(1)}$	1	0	0	1	0
$x^{(2)}$	0	1	0	0	0
$x^{(3)}$	1	1	0	1	0
$x^{(4)}$	0	1	1	0	1
$x^{(5)}$	1	1	1	1	1
$x^{(6)}$	0	0	1	0	1
$x^{(7)}$	1	0	1	1	1

	0	1	2	3	4
$y^{(0)}$	0	0	0	0	0
$y^{(1)}$	1	1	0	1	0
$y^{(2)}$	1	0	1	1	0
$y^{(3)}$	0	1	1	0	0
$y^{(4)}$	0	1	0	0	0
$y^{(5)}$	1	0	0	1	0
$y^{(6)}$	1	1	1	1	0
$y^{(7)}$	0	0	1	0	0

Note that for both the set of source addresses and the set of target addresses, the disk numbers, which appear in the leftmost three columns, are permutations of $\{0, 1, \ldots, 7\}$.

Proving that $\{x^{(0)}, x^{(1)}, \ldots, x^{(D-1)}\}$ is a 1-permutable set of blocks

It remains to show that the set $\{x^{(0)}, x^{(1)}, \ldots, x^{(D-1)}\}$ is a 1-permutable set of blocks. Letting $y^{(i)} = Ax^{(i)} \oplus c$ for $i = 0, 1, \ldots, D-1$, it suffices to show that both $\{x^{(0)}_{0..d-1}, x^{(1)}_{0..d-1}, \ldots, x^{(D-1)}_{0..d-1}\}$ and $\{y^{(0)}_{0..d-1}, y^{(1)}_{0..d-1}, \ldots, y^{(D-1)}_{0..d-1}\}$ are permutations of $\{0, 1, \ldots, D-1\}$. The following two theorems prove these properties.

Theorem 2.14 *Let $b = 0$, and define $\{x^{(0)}, x^{(1)}, \ldots, x^{(D-1)}\}$ according to equations (2.13)–(2.15). Then $\{x^{(0)}_{0..d-1}, x^{(1)}_{0..d-1}, \ldots, x^{(D-1)}_{0..d-1}\}$ is a permutation of $\{0, 1, \ldots, D-1\}$.*

Proof: It suffices to show that for all $k, l \in \{0, 1, \ldots, D-1\}$, $k \neq l$ implies $x^{(k)}_{0..d-1} \neq x^{(l)}_{0..d-1}$. Since $T \cup U = \{0, 1, \ldots, d-1\}$, then $k \neq l$ implies that $\mathrm{bin}_T(k) \neq \mathrm{bin}_T(l)$ or $\mathrm{bin}_U(k) \neq \mathrm{bin}_U(l)$. If $\mathrm{bin}_T(k) \neq \mathrm{bin}_T(l)$, then by equation (2.14), we have $x^{(k)}_T \neq x^{(l)}_T$ and hence $x^{(k)}_{0..d-1} \neq x^{(l)}_{0..d-1}$.

Now suppose that $\mathrm{bin}_T(k) = \mathrm{bin}_T(l)$. Then $U \neq \emptyset$ and $\mathrm{bin}_U(k) \neq \mathrm{bin}_U(l)$. By equation (2.13), we have

$$x^{(k)}_S = Q^{-1} A_{0..d-1,T}\ \mathrm{bin}_T(k) \oplus \mathrm{bin}(R(k)) \oplus c_{0..d-1} \ ,$$

$$x_S^{(l)} = Q^{-1} A_{0..d-1,T} \operatorname{bin}_T(l) \oplus \operatorname{bin}(R(l)) \oplus c_{0..d-1} ,$$

which implies that

$$x_S^{(k)} \oplus x_S^{(l)} = \operatorname{bin}(R(k)) \oplus \operatorname{bin}(R(l))$$

since $\operatorname{bin}_T(k) = \operatorname{bin}_T(l)$. Because $U = S \cap \{0, 1, \ldots, d-1\}$, if we were to order the column numbers in S, the first $|U|$ of them would be the set U. Thus, the first $|U|$ positions of $x_S^{(k)}$ comprise $x_U^{(k)}$, and the same holds for $x^{(l)}$. We therefore have

$$\begin{aligned} x_U^{(k)} \oplus x_U^{(l)} &= \operatorname{bin}_{0..|U|-1}(R(k)) \oplus \operatorname{bin}_{0..|U|-1}(R(l)) \\ &= \operatorname{bin}_U(k) \oplus \operatorname{bin}_U(l) \qquad \text{(by equation (2.12))} \\ &\neq 0 \end{aligned}$$

since $\operatorname{bin}_U(k) \neq \operatorname{bin}_U(l)$. We conclude that $x_U^{(k)} \neq x_U^{(l)}$ and thus $x_{0..d-1}^{(k)} \neq x_{0..d-1}^{(l)}$. ■

Theorem 2.15 *Let $b = 0$, define $\{x^{(0)}, x^{(1)}, \ldots, x^{(D-1)}\}$ according to equations (2.13)–(2.15), and let $y^{(i)} = Ax^{(i)} \oplus c$ for $i = 0, 1, \ldots, D-1$. Then $\{y_{0..d-1}^{(0)}, y_{0..d-1}^{(1)}, \ldots, y_{0..d-1}^{(D-1)}\}$ is a permutation of $\{0, 1, \ldots, D-1\}$.*

Proof: It suffices to show that for all $k, l \in \{0, 1, \ldots, D-1\}$, $k \neq l$ implies $y_{0..d-1}^{(k)} \neq y_{0..d-1}^{(l)}$. We start by noting that

$$\begin{aligned} x_S^{(k)} \oplus x_S^{(l)} &= (Q^{-1} A_{0..d-1,T} \operatorname{bin}_T(k) \oplus \operatorname{bin}(R(k)) \oplus c_{0..d-1}) \\ &\quad \oplus (Q^{-1} A_{0..d-1,T} \operatorname{bin}_T(l) \oplus \operatorname{bin}(R(l)) \oplus c_{0..d-1}) \\ &= Q^{-1} A_{0..d-1,T} (\operatorname{bin}_T(k) \oplus \operatorname{bin}_T(l)) \oplus \operatorname{bin}(R(k)) \oplus \operatorname{bin}(R(l)) . \end{aligned} \tag{2.16}$$

Thus,

$$\begin{aligned} y_{0..d-1}^{(k)} \oplus y_{0..d-1}^{(l)} &= (A_{0..d-1,0..n-1}\, x^{(k)} \oplus c_{0..d-1}) \oplus (A_{0..d-1,0..n-1}\, x^{(l)} \oplus c_{0..d-1}) \\ &= (A_{0..d-1,S}\, x_S^{(k)} \oplus A_{0..d-1,T}\, x_T^{(k)} \oplus A_{0..d-1,V}\, x_V^{(k)}) \\ &\quad \oplus (A_{0..d-1,S}\, x_S^{(l)} \oplus A_{0..d-1,T}\, x_T^{(l)} \oplus A_{0..d-1,V}\, x_V^{(l)}) \end{aligned}$$

$$\begin{aligned}
&= Q(x_S^{(k)} \oplus x_S^{(l)}) \oplus A_{0..d-1,T}(x_T^{(k)} \oplus x_T^{(l)}) \oplus A_{0..d-1,V} \cdot 0 \\
&= Q[Q^{-1}A_{0..d-1,T}(\mathrm{bin}_T(k) \oplus \mathrm{bin}_T(l)) \oplus \mathrm{bin}(R(k)) \oplus \mathrm{bin}(R(l))] \\
&\qquad \oplus A_{0..d-1,T}(\mathrm{bin}_T(k) \oplus \mathrm{bin}_T(l)) \qquad \text{(by equations (2.14) and (2.16))} \\
&= QQ^{-1}A_{0..d-1,T}(\mathrm{bin}_T(k) \oplus \mathrm{bin}_T(l)) \oplus Q(\mathrm{bin}(R(k)) \oplus \mathrm{bin}(R(l))) \\
&\qquad \oplus A_{0..d-1,T}(\mathrm{bin}_T(k) \oplus \mathrm{bin}_T(l)) \\
&= Q(\mathrm{bin}(R(k)) \oplus \mathrm{bin}(R(l))) \\
&\neq 0 \ ,
\end{aligned}$$

since $k \neq l$ implies $R(k) \neq R(l)$ and the submatrix Q is nonsingular. We conclude that $y_{0..d-1}^{(k)} \neq y_{0..d-1}^{(l)}$, which completes the proof. ■

2.5.2 Devising a schedule given a 1-permutable set of blocks

Having seen how to find a 1-permutable set of blocks, we now show how to use this set to decompose the BMMC permutation into 1-permutable sets of blocks that together form a schedule. We still assume that $B = 1$ and thus $b = 0$.

The method is simple. We define a set of $N/D - 1$ n-vectors $\{p^{(0)}, p^{(1)}, \ldots, p^{(N/D-1)}\}$ such that $p_{0..d-1}^{(j)} = 0$ and $p_{d..n-1}^{(j)}$ is the binary representation of j for $j = 0, 1, \ldots, N/D - 1$. Given a 1-permutable set of blocks $\{x^{(0)}, x^{(1)}, \ldots, x^{(D-1)}\}$, we form the following N/D sets:

$$\begin{gathered}
\{x^{(0)} \oplus p^{(0)}, x^{(1)} \oplus p^{(0)}, \ldots, x^{(D-1)} \oplus p^{(0)}\} \ , \\
\{x^{(0)} \oplus p^{(1)}, x^{(1)} \oplus p^{(1)}, \ldots, x^{(D-1)} \oplus p^{(1)}\} \ , \\
\{x^{(0)} \oplus p^{(2)}, x^{(1)} \oplus p^{(2)}, \ldots, x^{(D-1)} \oplus p^{(2)}\} \ , \\
\vdots \\
\{x^{(0)} \oplus p^{(N/D-1)}, x^{(1)} \oplus p^{(N/D-1)}, \ldots, x^{(D-1)} \oplus p^{(N/D-1)}\} \ .
\end{gathered} \tag{2.17}$$

Theorem 2.16 *Let $B = 1$, and let $\{x^{(0)}, x^{(1)}, \ldots, x^{(D-1)}\}$ be a 1-permutable set of blocks for a BMMC permutation with characteristic matrix A and complement vector c. For $j = 0, 1, \ldots, N/D-1$, let $p^{(j)}$ be an n-vector for which $p_{0..d-1}^{(j)} = 0$ and $p_{d..n-1}^{(j)}$ is the binary representation* *of j. Then the N/D sets (2.17) form a schedule to perform the BMMC permutation in one pass.*

Proof: We start by showing that each set is a 1-permutable set of blocks. For $i = 0, 1, \ldots, D-1$ and $j = 0, 1, \ldots, N/D-1$, let $y^{(i)} = Ax^{(i)} \oplus c$, $z^{(i,j)} = x^{(i)} \oplus p^{(j)}$, and $w^{(i,j)} = Az^{(i,j)} \oplus c$. For all j, we need to show that the sets $\{z^{(0,j)}_{0..d-1}, z^{(1,j)}_{0..d-1}, \ldots, z^{(D-1,j)}_{0..d-1}\}$ of source addresses and $\{w^{(0,j)}_{0..d-1}, w^{(1,j)}_{0..d-1}, \ldots, w^{(D-1,j)}_{0..d-1}\}$ of target addresses are permutations of $\{0, 1, \ldots, D-1\}$. Because $p^{(j)}_{0..d-1} = 0$ for $j = 0, 1, \ldots, N/D-1$, we have $z^{(i,j)}_{0..d-1} = x^{(i)}_{0..d-1}$ for all i and j. Since $\{x^{(0)}_{0..d-1}, x^{(1)}_{0..d-1}, \ldots, x^{(D-1)}_{0..d-1}\}$ is a permutation of $\{0, 1, \ldots, D-1\}$, so is $\{z^{(0,j)}_{0..d-1}, z^{(1,j)}_{0..d-1}, \ldots, z^{(D-1,j)}_{0..d-1}\}$. For $\{w^{(0,j)}_{0..d-1}, w^{(1,j)}_{0..d-1}, \ldots, w^{(D-1,j)}_{0..d-1}\}$, we have that for all i and j,

$$\begin{aligned} w^{(i,j)} &= Az^{(i,j)} \oplus c \\ &= A(x^{(i)} \oplus p^{(j)}) \oplus c \\ &= Ax^{(i)} \oplus Ap^{(j)} \oplus c \\ &= y^{(i)} \oplus Ap^{(j)} \; . \end{aligned}$$

Because $\{x^{(0)}, x^{(1)}, \ldots, x^{(D-1)}\}$ is a 1-permutable set of blocks, $\{y^{(0)}_{0..d-1}, y^{(1)}_{0..d-1}, \ldots, y^{(D-1)}_{0..d-1}\}$ is a permutation of $\{0, 1, \ldots, D-1\}$. The vector $Ap^{(j)}$ that we XOR into each $y^{(i)}$ depends only on j and not on i, and thus we XOR the same vector into $\{y^{(0)}_{0..d-1}, y^{(1)}_{0..d-1}, \ldots, y^{(D-1)}_{0..d-1}\}$. Therefore, $\{w^{(0,j)}_{0..d-1}, w^{(1,j)}_{0..d-1}, \ldots, w^{(D-1,j)}_{0..d-1}\}$ is a permutation of $\{0, 1, \ldots, D-1\}$.

It remains to show that each source block appears exactly once in the sets (2.17). Consider a block whose source address is the n-vector z. We must show that $z = x^{(i)} \oplus p^{(j)}$ for some i and j. Choose $i \in \{0, 1, \ldots, D-1\}$ so that $z_{0..d-1} = x^{(i)}_{0..d-1}$. Having chosen i, choose $j \in \{0, 1, \ldots, N/D-1\}$ so that $p^{(j)}_{d..n-1} = x^{(i)}_{d..n-1} \oplus z_{d..n-1}$. ■

Thus we see that, assuming that $B = 1$, once we have found *any* 1-permutable set of blocks, we can construct an entire schedule by simply XORing the most signficant t bits by $0, 1, \ldots, N/D-1$ in turn.

2.5.3 Removing the restriction on the block size

Removing the restriction that $B = 1$ is easy. By the structure of block BMMC matrices, the b least significant bits of a target address depend only on the b least significant bits of the corresponding source address. (Recall that the lower left $(n-b) \times b$ submatrix of the characteristic matrix is 0.) Moreover, the leading $b \times b$ submatrix of the characteristic matrix is nonsingular. Therefore, records that start in the same block end up in the same block. Likewise, the most significant $n-b$ bits of a target address depend only on the most significant $n-b$ bits of the corresponding source address. Blocks, therefore, are permuted according only to their block addresses. Thus, the "permute the records read in RAM" part of performing a pass consists of simply permuting records within the boundaries of their current blocks.

We have thus shown how to perform block BMMC permutations in one pass, using exactly $2N/BD$ parallel I/Os.

2.5.4 Block BPC permutations

For block BPC permutations, which are included in the class of block BMMC permutations, some of the mathematics in finding an initial 1-permutable set of blocks becomes considerably simpler. Let us again work under the assumption that $B = 1$; we use the method we have just seen to extend the scheme to the case in which $B > 1$. We are given a characteristic matrix $A = (a_{ij})$ that is a permutation matrix and a complement vector c.

Finding the set of basis columns S is then easy. We don't need to perform Gaussian elimination. We only need to find the columns that have a 1 within the first d rows:

$$S = \{j : a_{ij} = 1 \text{ for some } 0 \leq i \leq d-1\} \ .$$

(Note that $Q = A_{0..d-1,S}$ is a permutation matrix.) Since $S \cap T = \emptyset$ and all d of the 1s in the first d rows of A are contained in $A_{0..d-1,S}$, we have that $A_{0..d-1,T} = 0$. The computation of $x_S^{(i)}$, for $i = 0, 1, \ldots, D-1$, becomes simpler, involving no matrix operations. Starting from equation (2.13), we have

$$\begin{aligned} x_S^{(i)} &= Q^{-1} A_{0..d-1,T} \; \mathrm{bin}_T(i) \oplus \mathrm{bin}(R(i)) \oplus c_{0..d-1} \\ &= \mathrm{bin}(R(i)) \oplus c_{0..d-1} \; . \end{aligned}$$

The definitions of $x_T^{(i)} = \mathrm{bin}_T(i)$ and $x_V^{(i)} = 0$ remain the same.

2.6 BMMC permutations

In this section, we show how to perform BMMC permutations using MRC, block BPC, and BPC permutations. As usual, we assume that the BMMC permutation is given by an $n \times n$ characteristic matrix A and a complement vector c of length n. The number of parallel I/Os is at most

$$\frac{2N}{BD}\left(2\left\lceil \frac{\lg M - \mathrm{rank}(A_{0..\lg M-1,0..\lg M-1})}{\lg(M/B)} \right\rceil + H(N,M,B)\right) \; ,$$

where

$$H(N,M,B) = \begin{cases} 4\left\lceil \dfrac{\lg B}{\lg(M/B)} \right\rceil + 9 & \text{if } M \le \sqrt{N} \; , \\ 4\left\lceil \dfrac{\lg(N/B)}{\lg(M/B)} \right\rceil + 1 & \text{if } \sqrt{N} < M < \sqrt{NB} \; , \\ 5 & \text{if } \sqrt{NB} \le M \; . \end{cases} \tag{2.18}$$

Our strategy is to factor the matrix A into a product of matrices, each of which is the characteristic matrix of an MRC, block BPC, or BPC permutation. For now, we ignore the complement vector c. We read the factors right to left to determine the order in which to perform these permutations. For example, if we factor $A = V\,W$, then we perform A by performing the permutation with characteristic matrix W and then performing the permutation with characteristic matrix V. The reason for this right-to-left order is that if $y = Ax$, then we first multiply Wx, giving y', and then multiply Vy', giving y. Factoring the matrix A in this way will make it easy to count how many passes the BMMC permutation takes.

Permuting columns to create a nonsingular leading submatrix

We start by permuting the columns of A so that the leading $m \times m$ submatrix is nonsingular. That is, we factor

$$A = \widehat{A}\,\Pi\ ,$$

where Π is a permutation matrix and the submatrix $\widehat{A}_{0..m-1,0..m-1}$ is nonsingular. We would like to permute the columns of A so that the number of I/Os required to perform Π is minimum. We call such a permutation a *minimum-impact* permutation. The size of the largest set S of columns of $A_{0..m-1,0..m-1}$ that is linearly independent is equal to $\text{rank}(A_{0..m-1,0..m-1})$. As the following theorem shows, we can come up with a minimum-impact permutation by choosing a set T of $m - \text{rank}(A_{0..m-1,0..m-1})$ columns from the rightmost $n-m$ columns of A that, along with S, provide the needed set of m linearly independent columns.

Theorem 2.17 *Let A be a nonsingular $n \times n$ matrix, m be an integer such that $1 \le m \le n$, and $r = \text{rank}(A_{0..m-1,0..m-1})$. Then for any set S of r linearly independent columns of $A_{0..m-1,0..m-1}$, there is a set $T \subseteq \{m, m+1, \ldots, n-1\}$ such that $|T| = m - r$ and $A_{0..m-1,S\cup T}$ is a nonsingular $m \times m$ matrix.*

Proof: We start by showing that for any $k < m$, if $S' \subseteq \{0, 1, \ldots, n-1\}$ is such that $|S'| = k$ and the columns of $A_{0..m-1,S'}$ are linearly independent, then there exists a column $j \notin S'$ such that the columns of $A_{0..m-1,S'\cup\{j\}}$ are linearly independent. Suppose that no such j exists. Then all columns in $\{0, 1, \ldots, n-1\} - S'$ are linearly dependent on columns in S'. Because $|S'| < m$, there is no set of m linearly independent columns of $A_{0..m-1,0..n-1}$. The column rank of $A_{0..m-1,0..n-1}$ is therefore less than m, which implies that the row rank is as well. But then there are linearly dependent rows in $A_{0..m-1,0..n-1}$, contradicting our assumption that A is nonsingular.

We can therefore choose $m - r$ columns not in the set S of the theorem statement one at a time, to form a set T such that $A_{0..m-1,S\cup T}$ is nonsingular. Since $r = \text{rank}(A_{0..m-1,0..m-1})$, all the columns in $A_{0..m-1,\{0,1,\ldots,m-1\}-S}$ are linearly dependent on columns in $A_{0..m-1,S}$. The column indices in T, therefore, must be drawn from $\{m, m+1, \ldots, n-1\}$. ■

To find these sets S and T, we can use Gaussian elimination on A^{T}, just as we did in performing block BMMC permutations in Section 2.5. In Gaussian elimination, we choose a pivot row by finding a row with a nonzero diagonal entry at or below the current row position. In matrices with entries drawn from the real numbers, we often choose the diagonal with the greatest absolute value for the sake of numerical stability. Here, however, matrix entries are 0 or 1. By choosing the first row at or below the current row position that has a 1 in the diagonal, we get an LUP decomposition $PA^{\mathrm{T}} = LU$ in which the permutation matrix P has the minimum-impact property with respect to its rows. By setting

$$\Pi = P^{\mathrm{T}} = P^{-1} , \tag{2.19}$$

the matrix Π has the minimum-impact property with respect to its columns, as desired.

How many I/Os are required to perform the permutation Π? Since Π specifies that $m - \operatorname{rank}(A_{0..m-1,0..m-1})$ columns move from the upper $n-m$ positions to the lower m positions, we have $\rho_m(\Pi) = m - \operatorname{rank}(A_{0..m-1,0..m-1})$. We claim that $\rho_b(\Pi) \le \rho_m(\Pi)$. Why? For each column that moves from the upper $n-b$ positions to the lower b positions, there is a column that moves from the upper $n-m$ positions to the lower m positions. A column that moves from the upper $n-m$ positions to the lower b positions is moving into the lower m positions as well, since $m \ge b$. A column that moves from positions $\{b, b+1, \ldots, m-1\}$ into the lower b positions vacates a position that is filled by a column from the upper $n-m$ positions. Thus, $\rho(\Pi) = \rho_m(\Pi) = m - \operatorname{rank}(A_{0..m-1,0..m-1})$. By Theorem 2.8, therefore, we can perform Π using at most

$$\frac{2N}{BD}\left(2\left\lceil\frac{\lg M - \operatorname{rank}(A_{0..\lg M-1,0..\lg M-1})}{\lg(M/B)}\right\rceil + 1\right) \tag{2.20}$$

parallel I/Os.

Factoring the remaining matrix

Our next task is to factor the matrix $\widehat{A}$, which is nonsingular since A and Π are. We divide $\widehat{A}$ between the lower m rows and columns and the upper $n-m$ rows and columns:

$$\widehat{A} = \begin{array}{cc} \begin{array}{c|c} m & n-m \end{array} & \\ \left[\begin{array}{c|c} \alpha & \beta \\ \hline \gamma & \delta \end{array}\right] & \begin{array}{c} m \\ n-m \end{array} \end{array} ,$$

so that the leading $m \times m$ submatrix α is nonsingular. Then we factor $\widehat{A}$ as

$$\widehat{A} = V\,W = \begin{array}{c} \begin{array}{cc} m & n-m \end{array} \\ \left[\begin{array}{c|c} I & 0 \\ \hline \gamma\,\alpha^{-1} & I \end{array}\right] \end{array} \begin{array}{cc} \begin{array}{cc} m & n-m \end{array} & \\ \left[\begin{array}{c|c} \alpha & \beta \\ \hline 0 & \gamma\,\alpha^{-1}\,\beta \oplus \delta \end{array}\right] & \begin{array}{c} m \\ n-m \end{array} \end{array} . \quad (2.21)$$

Because the matrix V is unit lower triangular, Lemma 2.10 says that it is nonsingular. Therefore, so is the matrix W. In fact, it is easy to show that W characterizes an MRC permutation, which requires only one pass. The leading $m \times m$ submatrix α is nonsingular by construction. The trailing $(n-m) \times (n-m)$ submatrix $\gamma\,\alpha^{-1}\,\beta \oplus \delta$ must be nonsingular as well, for if it contained linearly dependent rows, then so would W, contradicting the nonsingularity of W.

We still have to factor the matrix V, which would be an identity matrix if the lower left submatrix, equal to $\gamma\,\alpha^{-1}$, were 0. The location of this submatrix is exactly the location that is required to be all 0s in both MRC and block BMMC permutations. If $\gamma = 0$, then $\gamma\,\alpha^{-1} = 0$, and we don't need to factor any further. In general, however, we factor V by applying BPC permutations to move the entries of this submatrix to locations that allow us to perform MRC or block BPC permutations. The way in which we factor V depends on whether $m \le n-m$, which is equivalent to $m \le n/2$ or $M \le \sqrt{N}$.

First, we handle the case in which $m > n/2$. We divide the $(n-m) \times m$ submatrix product $\gamma\,\alpha^{-1}$ as

$$\gamma\,\alpha^{-1} = \begin{array}{cc} \begin{array}{cc} 2m-n & n-m \end{array} & \\ \left[\begin{array}{c|c} \sigma & \tau \end{array}\right] & n-m \end{array}$$

so that we divide V as

$$V = \begin{array}{c} \\ \\ \\ \end{array}\begin{array}{ccc|c} 2m-n & n-m & n-m & \\ \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & I & 0 \\ \hline \sigma & \tau & I \end{array}\right] & & & \begin{array}{c} 2m-n \\ n-m \\ n-m \end{array} \end{array} .$$

We factor V into

$$V = \Pi' \, Z \, \Pi' \; ,$$

where

$$\Pi' = \begin{array}{cc} \begin{array}{ccc} 2m-n & n-m & n-m \end{array} & \\ \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & 0 & I \\ \hline 0 & I & 0 \end{array}\right] & \begin{array}{c} 2m-n \\ n-m \\ n-m \end{array} \end{array} \tag{2.22}$$

and

$$Z = \begin{array}{cc} \begin{array}{ccc} 2m-n & n-m & n-m \end{array} & \\ \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline \sigma & I & \tau \\ \hline 0 & 0 & I \end{array}\right] & \begin{array}{c} 2m-n \\ n-m \\ n-m \end{array} \end{array} . \tag{2.23}$$

The matrix Z is characteristic of an MRC permutation, since its leading $m \times m$ submatrix is unit lower triangular and by Lemma 2.10, is thus nonsingular. The permutation characterized by Z can therefore be performed in one pass.

There are two cases in the analysis for the permutation matrix Π':

1. If $b \le 2m - n$ or, equivalently, $\sqrt{NB} \le M$, then Π' is the characteristic matrix of a block BPC permutation, so Π' can be performed in only one pass.

2. If $b > 2m - n$ or, equivalently, $\sqrt{NB} > M$, then we use Theorem 2.8 to obtain a bound. We have

$$\begin{aligned} \rho_m(\Pi') &= n - m \; , \\ \rho_b(\Pi') &= b - (2m - n) \\ &\le n - m \; , \end{aligned}$$

so that $\rho(\Pi') = n - m$. Therefore, we can perform Π' in at most

$$\begin{aligned} 2\left\lceil \frac{n-m}{m-b} \right\rceil + 1 &= 2\left\lceil \frac{n-b}{m-b} - 1 \right\rceil + 1 \\ &= 2\left\lceil \frac{\lg(N/B)}{\lg(M/B)} \right\rceil - 1 \end{aligned} \tag{2.24}$$

passes.

Now we factor V for the case in which $m \le n/2$ or, equivalently, $M \le \sqrt{N}$. We divide $\gamma\,\alpha^{-1}$ as

$$\gamma\,\alpha^{-1} = \begin{array}{c} m \\ \left[\begin{array}{c} \phi \\ \hline \psi \end{array}\right] \end{array} \begin{array}{l} \\ m \\ n-2m \end{array}$$

so that we divide V as

$$V = \begin{array}{cl} \begin{array}{ccc} m & m & n-2m \end{array} & \\ \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline \phi & I & 0 \\ \hline \psi & 0 & I \end{array}\right] & \begin{array}{l} m \\ m \\ n-2m \end{array} \end{array} .$$

We factor V into

$$V = \Pi''\,Y\,\Pi''\ ,$$

where

$$\Pi'' = \begin{array}{cl} \begin{array}{ccc} m & m & n-2m \end{array} & \\ \left[\begin{array}{c|c|c} 0 & I & 0 \\ \hline I & 0 & 0 \\ \hline 0 & 0 & I \end{array}\right] & \begin{array}{l} m \\ m \\ n-2m \end{array} \end{array} \tag{2.25}$$

characterizes a BPC permutation and

$$Y = \begin{array}{cl} \begin{array}{ccc} m & m & n-2m \end{array} & \\ \left[\begin{array}{c|c|c} I & \phi & 0 \\ \hline 0 & I & 0 \\ \hline 0 & \psi & I \end{array}\right] & \begin{array}{l} m \\ m \\ n-2m \end{array} \end{array} . \tag{2.26}$$

Because the trailing $(n-m) \times (n-m)$ submatrix of Y, equal to $\left[\begin{array}{c|c} I & 0 \\ \hline \psi & I \end{array}\right]$, is unit lower triangular, this submatrix is nonsingular. Hence, Y characterizes an MRC permutation, which can be performed in one pass. For the permutation matrix Π'', we have

$$\rho_b(\Pi'') = b \leq m = \rho_m(\Pi'') \ ,$$

which implies that the number of passes to perform Π'' is at most

$$\begin{aligned} 2\left\lceil \frac{m}{m-b} \right\rceil + 1 &= 2\left\lceil \frac{b}{m-b} + 1 \right\rceil + 1 \\ &= 2\left\lceil \frac{\lg B}{\lg(M/B)} \right\rceil + 3 \ . \end{aligned} \tag{2.27}$$

Analysis

Putting all the factors together, we get the following theorem.

Theorem 2.18 *A BMMC permutation with characteristic matrix A and complement vector c can be performed using at most*

$$\frac{2N}{BD}\left(2\left\lceil \frac{\lg M - \operatorname{rank}(A_{0..\lg M-1,0..\lg M-1})}{\lg(M/B)} \right\rceil + H(N,M,B)\right)$$

parallel I/Os, where

$$H(N,M,B) = \begin{cases} 4\left\lceil \dfrac{\lg B}{\lg(M/B)} \right\rceil + 9 & \text{if } M \leq \sqrt{N} \ , \\ 4\left\lceil \dfrac{\lg(N/B)}{\lg(M/B)} \right\rceil + 1 & \text{if } \sqrt{N} < M < \sqrt{NB} \ , \\ 5 & \text{if } \sqrt{NB} \leq M \ . \end{cases}$$

Proof: Assume for the moment that the complement vector c is all 0s. We factor the matrix A into a product of matrices and perform the permutations given by the factors from right to left.

If $M \geq \sqrt{N}$, we factor A as

$$A = \Pi' \, Z \, \Pi' \, W \, \Pi \ ,$$

where the factors are defined in equations (2.19) and (2.21)–(2.23). By expression (2.20), at most $2\left\lceil\frac{\lg M - \operatorname{rank}(A_{0..\lg M-1,0..\lg M-1})}{\lg(M/B)}\right\rceil + 1$ passes are needed to perform the BPC permutation Π. If $\sqrt{NB} \le M$, then the remaining four factors can be performed in one pass each. If $\sqrt{N} < M < \sqrt{NB}$, then by expression (2.24), the two Π' BPC permutations can be performed in at most

$$2\left(2\left\lceil\frac{\lg(N/B)}{\lg(M/B)}\right\rceil - 1\right) = 4\left\lceil\frac{\lg(N/B)}{\lg(M/B)}\right\rceil - 2$$

passes. Adding in the two passes for W and Z achieves the stated bound.

If $M < \sqrt{N}$, we factor A as

$$A = \Pi'' \, Y \, \Pi'' \, W \, \Pi \; ,$$

where the factors are defined by equations (2.19), (2.21), (2.25), and (2.26). Again, expression (2.20) gives the I/O bound to perform Π. By expression (2.27), we can perform the two Π'' BPC permutations in at most

$$2\left(2\left\lceil\frac{\lg B}{\lg(M/B)}\right\rceil + 3\right) = 4\left\lceil\frac{\lg B}{\lg(M/B)}\right\rceil + 6$$

passes. Adding in the two passes for W and Y achieves the stated bound.

To handle cases in which the complement vector c is nonzero, we include c in the last BPC permutation performed, either Π' or Π''. If $V = I$, in which case we perform neither Π' nor Π'', we complement when performing the MRC permutation W. ■

We conclude this section with an observation about the function $H(N, M, B)$. It is $\Theta\left(\frac{\lg B}{\lg(M/B)} + 1\right)$ if $M \le \sqrt{N}$ and $\Theta\left(\frac{\lg(N/B)}{\lg(M/B)} + 1\right)$ if $\sqrt{N} < M < \sqrt{NB}$. Although the latter bound is asymptotically no better than we get by sorting, we can actually write the asymptotic bound as $\Theta\left(\frac{\lg \min(B, N/B)}{\lg(M/B)} + 1\right)$ for $M < \sqrt{NB}$. If $M \le \sqrt{N}$, then we have $B \le M \le \sqrt{N}$, which implies that $N/B \ge \sqrt{N} \ge B$. If $\sqrt{N} < M < \sqrt{NB}$, then $\frac{\lg(N/B)}{\lg(M/B)} = O\left(\frac{\lg B}{\lg(M/B)} + 1\right)$, as follows. Let $M = \sqrt{N}B^{\epsilon}$, where $0 < \epsilon < 1/2$. Then,

$$\frac{\lg(N/B)}{\lg(M/B)} = \frac{\lg(N/M)}{\lg(M/B)} + 1$$

$$\begin{aligned}
&= \frac{\lg(\sqrt{N}/B^{\epsilon})}{\lg(\sqrt{N}/B^{1-\epsilon})} + 1 \\
&= \log_{\sqrt{N}/B^{1-\epsilon}} \frac{\sqrt{N}}{B^{\epsilon}} + 1 \\
&= \log_{\sqrt{N}/B^{1-\epsilon}} B^{1-2\epsilon} + 2 \\
&< \frac{\lg B}{\lg(M/B)} + 2 \ .
\end{aligned}$$

Although our method for performing BMMC permutations is not asymptotically faster than sorting when $\sqrt{N} < M < \sqrt{NB}$, the value of $\frac{\lg(N/B)}{\lg(M/B)}$ is relatively small, being bounded asymptotically by $\frac{\lg B}{\lg(M/B)}$ in this range.

2.7 Arbitrary block permutations

In this section, we show how to perform arbitrary block permutations in one pass with off-line scheduling. By "off-line," we mean that, unlike all the other algorithms in this chapter, the schedule is not easily computed on-line; the amount of data required to compute a schedule for an arbitrary block permutation may exceed the RAM available in the computer.

Let us assume that we are given a permutation $\pi : \{0, 1, \ldots, N/B-1\} \overset{1\text{-}1}{\rightarrow} \{0, 1, \ldots, N/B-1\}$ of block addresses. We shall compute off-line a schedule for π consisting of a set of N/BD 1-permutable sets of blocks. The key idea is to view the permutation as an undirected bipartite multigraph[10] and find a good edge coloring on the multigraph. A *k-edge coloring* of a (multi)graph $G = (V, E)$ is an assignment of k colors to the edges of G such that no two edges incident on a common vertex have the same color.

We interpret a block permutation π as a bipartite multigraph as follows. The multigraph is $G_\pi = (V, E)$, where $V = S \cup T$, $S = \{s_0, s_1, \ldots, s_{D-1}\}$, $T = \{t_0, t_1, \ldots, t_{D-1}\}$, and E contains one edge (s_i, t_j) for each block whose source address is on disk $\mathcal{D}_i$ and target address is on disk $\mathcal{D}_j$. Since each disk contains N/BD source addresses and N/BD target addresses, each

[10]A multigraph is a graph that can have multiple edges between vertices.

vertex of G_π has exactly N/BD incident edges. That is, G_π is N/BD-regular. By a well-known theorem (see Bondy and Murty [BM76, p. 93] for example), G_π has an N/BD-edge coloring. The set of edges colored by any given color in such a coloring corresponds directly to a 1-permutable set of blocks: each edge in the set corresponds to a block, and each disk appears as a source exactly once and as a target exactly once.

The problem then is to find an N/BD-edge coloring. Unfortunately, this problem can get big. There are N/B edges, requiring $\Theta(N/B)$ storage; if $N/B = \omega(M)$, then we cannot fit G_π into RAM.[11] If $N/B = \omega(D^2)$, we can reduce this storage to $\Theta(D^2)$ by representing G_π by a $D \times D$ integer-valued matrix in which each entry represents the number of duplicate edges connecting two vertices. If $D^2 = \omega(M)$, however, this scheme does not work. And, even if we can fit the representation of G_π into RAM, finding an N/BD-edge coloring can be time-consuming. The sequential algorithm by Gabow and Kariv [GK82] finds the coloring in $O(\min(|E| \lg^2 |V|, |V|^2 \lg |V|, |V| \, |E_{\text{unique}}| \lg \mu))$ time, where E_{unique} is the set of "unique" edges of G_π (all edges between the same pair of vertices are coalesced into one) and μ is the maximum edge multiplicity (the maximum over all pairs of vertices of the number of multiedges between them).

2.8 Lower bounds

In this section, we prove lower bounds on parallel I/Os for BPC and BMMC permutations. We show that any algorithm that performs a BMMC permutation with characteristic matrix A requires $\Omega\left(\frac{N}{BD}\left(1 + \frac{\operatorname{rank}(A_{b..n-1,0..b-1})}{\lg(M/B)}\right)\right)$ parallel I/Os. We then use this lower bound to show that our BPC algorithm in Section 2.2 is asymptotically optimal. We also use this bound to prove an $\Omega\left(\frac{N}{BD}\frac{\lg M - \operatorname{rank}(A_{0..\lg M-1,0..\lg M-1})}{\lg(M/B)}\right)$ bound for BMMC permutations. The additive term of $\Theta\left(\frac{N}{BD}H(N,M,B)\right)$ represents a gap between the lower and upper bounds.

We will rely heavily on the technique used by Aggarwal and Vitter [AV88] for a lower bound

[11]We use the condition $N/B = \omega(M)$ rather than $N/B > M$ to gloss over the difference between the size of the data records to be permuted and the size of the representation of edges of G_π.

on I/Os in matrix transposition; their proof is based in turn on a method by Floyd [Flo72]. We prove the lower bound for the case in which $D = 1$, the general case following by dividing by D. We consider only I/Os that are *simple*. An input is simple if each record read is removed from the disk and moved into an empty location in RAM. An output is simple if the records are removed from the RAM and written to empty locations on the disk. When all I/Os are simple, exactly one copy of each record exists at any time during the execution of an algorithm. The following lemma, proven by Aggarwal and Vitter, allows us to consider only simple I/Os when proving lower bounds.

Lemma 2.19 *For each computation that implements a permutation of records, there is a corresponding computation strategy involving only simple I/Os such that the total number of I/Os is no greater.* ■

Potential function

The basic scheme of the proof uses a potential function argument. We call the time interval starting when the qth I/O occurs and ending just before the $(q+1)$st I/O *time* q. We define a potential function Φ so that $\Phi(q)$ is the *potential* at time q. We compute the initial and final potentials and bound the amount that the potential can increase in each I/O operation. The lower bound then follows.

To be more precise, we start with some definitions. For $i = 0, 1, \ldots, N/B - 1$, we define the ith *target group* to be the set of records that belong in block i according to the permutation π. (Remember that $D = 1$.) We denote by $g_{\text{block}}(i, k, q)$ the number of records in the ith target group that are in block k at time q, and $g_{\text{mem}}(i, q)$ denotes the number of records in the ith target group that are in RAM at time q. We define the continuous function

$$f(x) = \begin{cases} x \lg x & \text{if } x > 0 \ , \\ 0 & \text{if } x = 0 \ , \end{cases}$$

and we define *togetherness functions*

$$G_{\text{block}}(k,q) = \sum_{i=0}^{N/B-1} f(g_{\text{block}}(i,k,q))$$

for each block k at time q and

$$G_{\text{mem}}(q) = \sum_{i=0}^{N/B-1} f(g_{\text{mem}}(i,q))$$

for RAM at time q. Finally, we define the *potential* at time q, denoted $\Phi(q)$, as the sum of the togetherness functions:

$$\Phi(q) = G_{\text{mem}}(q) + \sum_{k=0}^{N/B-1} G_{\text{block}}(k,q) \ . \tag{2.28}$$

Aggarwal and Vitter embed the following key lemma in their lower-bound argument.

Lemma 2.20 *If $D = 1$, any algorithm that performs a permutation uses $\Omega\left(\frac{N}{B} + \frac{N \lg B - \Phi(0)}{B \lg(M/B)}\right)$ parallel I/Os, where the function Φ is defined in equation (2.28).* ∎

Domains

To prove the lower bounds, we shall require the following lemma. For a $p \times q$ matrix A and a p-vector $y \in \mathcal{R}(A)$, we define the *domain* of y as

$$\text{Dom}(A,y) = \{x : Ax = y\} \ .$$

That is, $\text{Dom}(A,y)$ is the set of q-vectors x that map to y when multiplied by A.

Lemma 2.21 *Let A be a $p \times q$ matrix whose entries are drawn from $\{0,1\}$, let y be any p-vector in $\mathcal{R}(A)$, and let $r = \text{rank}(A)$. Then $|\text{Dom}(A,y)| = 2^{q-r}$.*

Proof: Let S index a maximal set of linearly independent columns of A, so that $S \subseteq \{0,1,\ldots,q-1\}$, $|S| = r$, the columns of the submatrix $A_{0..p-1,S}$ are linearly independent, and for any column number $j \notin S$, the column $A_{0..p-1,j}$ is linearly dependent on the columns of $A_{0..p-1,S}$. Let $S' = \{0, 1, \ldots, q-1\} - S$.

We claim that for any value $i \in \{0, 1, \ldots, 2^{q-r} - 1\}$, there is a unique q-vector $x^{(i)}$ for which $x^{(i)}_{S'}$ is the binary representation of i and $y = Ax^{(i)}$. Why? The columns of $A_{0..p-1,S}$ span $\mathcal{R}(A)$, which implies that for all $z \in \mathcal{R}(A)$, there is a unique r-vector w such that $z = A_{0..p-1,S}\, w$. Letting $z = y \oplus A_{0..p-1,S'}\, x^{(i)}_{S'}$ and $x^{(i)}_{S} = w$ proves the claim.

Thus, we have shown that $|\text{Dom}(A, y)| \geq 2^{q-r}$. If we had $|\text{Dom}(A, y)| > 2^{q-r}$, then because y is arbitrarily chosen from $\mathcal{R}(A)$, we would have that $\sum_{y' \in \mathcal{R}(A)} |\text{Dom}(A, y')| > 2^q$. But this inequality contradicts there being only 2^q possible domain vectors. We conclude that $|\text{Dom}(A, y)| = 2^{q-r}$. ∎

The key lemma

The following lemma is the basis for the lower bounds.

Lemma 2.22 *Any algorithm that performs a BMMC permutation with characteristic matrix A requires $\Omega\left(\frac{N}{BD}\left(1 + \frac{\text{rank}(A_{b..n-1,0..b-1})}{\lg(M/B)}\right)\right)$ parallel I/Os.*

Proof: We prove the lower bound for the case in which $D = 1$, which implies that $d = 0$, the general case following by dividing by D. We assume that all I/Os are simple and transfer exactly B records, some possibly empty. Since all records start on disk and I/Os are simple, RAM is initially empty.

We need to compute the initial potential in order to apply Lemma 2.20. Let us view the binary representation of an $(n-b)$-bit block number k as being b-indexed rather than 0-indexed. That is, we define

$$\text{bin}(k) = (k_b, k_{b+1}, \ldots, k_{n-1}) ,$$

where k_b is the least significant bit of the $(n-b)$-bit binary representation of k and k_{n-1} is the most significant bit.

The initial potential depends on the number of records that start in the same source block and are in the same target group. Consider a record with source address $x = (x_0, x_1, \ldots, x_{n-1})$.

For a record with source address $w = (w_0, w_1, \ldots, w_{n-1})$ to be in the same source block as x, we must have that $w_{b..n-1} = x_{b..n-1}$. Also, if we let $y = Aw$ and $z = Ax$, then x and w are in the same target group if $y_{b..n-1} = z_{b..n-1}$, or

$$A_{b..n-1,0..n-1}\, w_{0..n-1} = A_{b..n-1,0..n-1}\, x_{0..n-1} \ ,$$

which is equivalent to

$$A_{b..n-1,0..b-1}\, w_{0..b-1} = A_{b..n-1,0..b-1}\, x_{0..b-1} \ ,$$

since $w_{b..n-1} = x_{b..n-1}$. Letting $r = \operatorname{rank}(A_{b..n-1,0..b-1})$, Lemma 2.21 says that there are 2^{b-r} distinct records w in $\operatorname{Dom}(A, A_{b..n-1,0..b-1}x_{0..b-1})$. Thus, there are $2^{b-r} = B/2^{\operatorname{rank}(A_{b..n-1,0..b-1})}$ records in x's source block that are in x's target group.

Now we compute $g_{\text{block}}(i, k, 0)$ for all blocks i and k. Block k contains some record that maps to block i if and only if there exists some source address $x = (x_0, x_1, \ldots, x_{n-1})$ such that $x_{b..n-1} = \operatorname{bin}(k)$ (i.e., x is in source block k) and

$$\begin{aligned}
\operatorname{bin}(i) &= A_{b..n-1,0..n-1}\, x_{0..n-1} \\
&= A_{b..n-1,0..b-1}\, x_{0..b-1} \oplus A_{b..n-1,b..n-1}\, x_{b..n-1} \\
&= A_{b..n-1,0..b-1}\, x_{0..b-1} \oplus A_{b..n-1,b..n-1}\, \operatorname{bin}(k) \ .
\end{aligned}$$

For each block number k, Lemma 2.1 says that there are $2^{\operatorname{rank}(A_{b..n-1,0..b-1})}$ different values of $A_{b..n-1,0..b-1}x_{0..b-1}$ as $x_{0..b-1}$ takes on all values in $\{0, 1, \ldots, B-1\}$. Since k is fixed, there is only 1 value of $A_{b..n-1,b..n-1}\operatorname{bin}(k)$. Thus, there are $2^{\operatorname{rank}(A_{b..n-1,0..b-1})}$ different values of $\operatorname{bin}(i)$.

Now we can compute $\Phi(0)$. Since RAM is initially empty, $g_{\text{mem}}(i, 0) = 0$ for all blocks i, which implies that $G_{\text{mem}}(0) = 0$. Letting $r = \operatorname{rank}(A_{b..n-1,0..b-1})$, we have

$$\Phi(0) \quad = \quad G_{\text{mem}}(0) + \sum_{k=0}^{N/B-1} G_{\text{block}}(k, 0)$$

$$\begin{aligned}
&= 0 + \sum_{k=0}^{N/B-1} \sum_{i=0}^{N/B-1} f(g_{\text{block}}(i,k,0)) \\
&= \sum_{k=0}^{N/B-1} 2^r \frac{B}{2^r} \lg \frac{B}{2^r} \\
&= \frac{N}{B} B \lg \frac{B}{2^r} \\
&= N(\lg B - r) \ . \qquad (2.29)
\end{aligned}$$

Combining Lemma 2.20 and equation (2.29), we get a lower bound of

$$\Omega\left(\frac{N}{B} + \frac{N \lg B - N(\lg B - r)}{B \lg(M/B)}\right) = \Omega\left(\frac{N}{B}\left(1 + \frac{\operatorname{rank}(A_{b..n-1,0..b-1})}{\lg(M/B)}\right)\right)$$

parallel I/Os. Dividing through by D yields a bound of

$$\Omega\left(\frac{N}{BD}\left(1 + \frac{\operatorname{rank}(A_{b..n-1,0..b-1})}{\lg(M/B)}\right)\right) \ ,$$

which completes the proof. ■

A lower bound for BPC permutations

Using Lemma 2.22, we can prove that the PERFORM-BPC procedure in Section 2.2, which uses $O\left(\frac{N}{BD}\left(1 + \frac{\rho(A)}{\lg(M/B)}\right)\right)$ parallel I/Os, is asymptotically optimal.

Theorem 2.23 *Any algorithm that performs a BPC permutation with characteristic matrix A requires $\Omega\left(\frac{N}{BD}\left(1 + \frac{\rho(A)}{\lg(M/B)}\right)\right)$ parallel I/Os.*

Proof: From equation (2.5), we have $\rho_b(A) = \operatorname{rank}(A_{b..n-1,0..b-1})$. Because the characteristic matrix is a permutation matrix, the rank of any submatrix is equal to the number of 1s in the submatrix. Looking at submatrices as sets of positions, we have that

$$A_{m..n-1,0..m-1} = A_{b..n-1,0..b-1} - A_{b..m-1,0..b-1} \cup A_{m..n-1,b..m-1} \ ,$$

which implies that

$$\begin{aligned}\rho_m(A) &= \operatorname{rank}(A_{m..n-1,0..m-1}) \\ &\leq \operatorname{rank}(A_{b..n-1,0..b-1}) + \operatorname{rank}(A_{m..n-1,b..m-1}) \\ &\leq \rho_b(A) + (m-b) ,\end{aligned}$$

since $A_{m..n-1,b..m-1}$ contains $m-b$ columns and therefore has rank at most $m-b$. Thus,

$$\frac{\rho_m(A)}{\lg(M/B)} \leq \frac{\rho_b(A)}{\lg(M/B)} + 1 .$$

Since $\rho(A) = \max(\rho_m(A), \rho_b(A))$, we have shown that $\frac{\rho_b(A)}{\lg(M/B)} = \Theta\left(\frac{\rho(A)}{\lg(M/B)}\right)$. Combined with Lemma 2.22, we get a lower bound of $\Omega\left(\frac{N}{BD}\left(1 + \frac{\rho(A)}{\lg(M/B)}\right)\right)$ parallel I/Os. ∎

A lower bound for BMMC permutations

The following theorem gives a lower bound for BMMC permutations that matches one term in the upper bound proven in Section 2.6.

Theorem 2.24 *Any algorithm that performs a BMMC permutation with characteristic matrix A requires $\Omega\left(\frac{N}{BD}\left(\frac{\lg M - \operatorname{rank}(A_{0..\lg M-1,0..\lg M-1})}{\lg(M/B)}\right)\right)$ parallel I/Os.*

Proof: By Lemma 2.22, it suffices to show that

$$\frac{\operatorname{rank}(A_{b..n-1,0..b-1})}{\lg(M/B)} \geq \frac{m - \operatorname{rank}(A_{0..m-1,0..m-1})}{\lg(M/B)} - 1 .$$

For any subset of row indices S, any subset of column indices T, and any column index j, we have

$$\operatorname{rank}(A_{S,T-\{j\}}) \geq \operatorname{rank}(A_{S,T}) - 1 , \tag{2.30}$$

since $\operatorname{rank}(A_{S,T-\{j\}}) = \operatorname{rank}(A_{S,T})$ if $A_{S,j}$ is linearly dependent on other columns in $A_{S,T-\{j\}}$ and $\operatorname{rank}(A_{S,T-\{j\}}) = \operatorname{rank}(A_{S,T}) - 1$ otherwise. Applying equation (2.30) once for each column

$j = b, b+1, \ldots, m-1$, we get

$$\mathrm{rank}(A_{m..n-1,0..b-1}) \geq \mathrm{rank}(A_{m..n-1,0..m-1}) - (m-b) \ . \tag{2.31}$$

Because A is nonsingular, the submatrix $A_{0..n-1,0..m-1}$ contains a subset of m linearly indepedent rows. The quantity $\mathrm{rank}(A_{m..n-1,0..m-1})$ is the cardinality of a maximal set of linearly independent rows in $(A_{m..n-1,0..m-1})$; there are at least $m - \mathrm{rank}(A_{0..m-1,0..m-1})$ such rows. Thus,

$$\mathrm{rank}(A_{m..n-1,0..m-1}) \geq m - \mathrm{rank}(A_{0..m-1,0..m-1}) \ . \tag{2.32}$$

We also have

$$\mathrm{rank}(A_{b..n-1,0..b-1}) \geq \mathrm{rank}(A_{m..n-1,0..b-1}) \tag{2.33}$$

since appending rows to any submatrix cannot decrease the rank. Putting equations (2.31)–(2.33) together, we have

$$\begin{aligned}
\frac{\mathrm{rank}(A_{b..n-1,0..b-1})}{m-b} &\geq \frac{\mathrm{rank}(A_{m..n-1,0..b-1})}{m-b} \\
&\geq \frac{\mathrm{rank}(A_{m..n-1,0..m-1}) - (m-b)}{m-b} \\
&\geq \frac{m - \mathrm{rank}(A_{0..m-1,0..m-1})}{m-b} - 1 \ ,
\end{aligned}$$

which completes the proof. ■

We do not know how tight a bound the additive $\Theta\left(\frac{N}{BD} H(N, M, B)\right)$ term is. There are some BMMC permutations, such as Π' in equation (2.22) and Π'' in equation (2.25), that require $\Omega\left(\frac{N}{BD} H(N, M, B)\right)$ parallel I/Os, but there are other BMMC permutations, such as BPC permutations with low cross-ranks, that can be performed with fewer. We'll have a little more to say on this topic in Section 2.10.

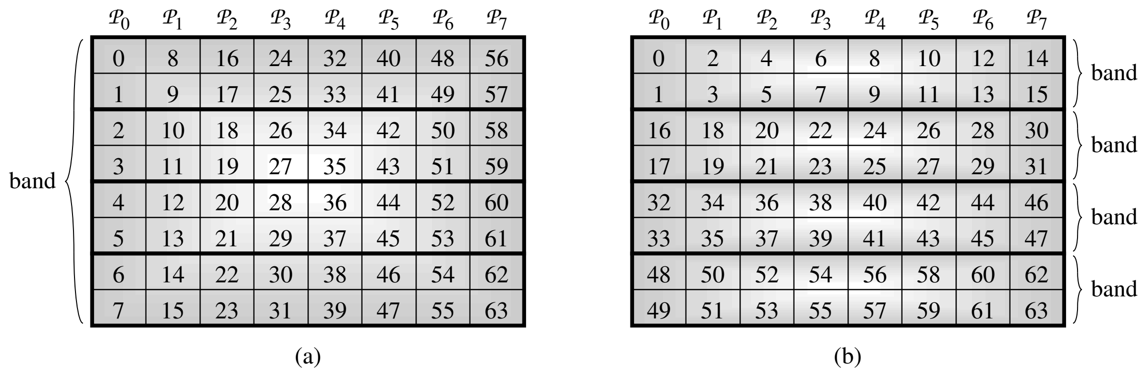
2.9 Performing BMMC and BPC permutations with other vector layouts

The algorithms in this chapter assume that the vectors they permute are ordered on the parallel disk system according to the Vitter-Shriver scheme shown in Figure 2.1. In practice, however, vectors might not be organized in this fashion. This section shows that when the difference between the actual vector organization and the Vitter-Shriver scheme is itself a BMMC permutation, we can perform BMMC and BPC permutations by running the algorithms in this chapter without changing the code. We compensate instead by altering the input characteristic matrix and complement vector. Before seeing how to alter the inputs in general, we examine a case in which we want to do so.

Converting banded layouts to the Vitter-Shriver scheme

Chapter 4 describes situations in which vector layouts other than the Vitter-Shriver scheme may occur. We summarize them here. In a *banded layout*, we divide a vector of length N into *bands* of β elements each. We restrict the band size β to be a power of 2 times the number of processors. Figure 2.3 shows two examples of banded layout with $P = 8$ processors and a track size of $BD = 16$. In Figure 2.3(a), the band size is $\beta = 64$, and so each band occupies $\beta/BD = 4$ tracks. In Figure 2.3(b), the band size equals the track size—$\beta = BD = 16$—and so each band occupies one track. This case is precisely the Vitter-Shriver layout. Each row in Figure 2.3 contains one element per processor. Element indices vary most rapidly within each processor, then among processors within a band, and they vary least rapidly from band to band. Within each band, elements are in column-major order.

The mapping from a banded layout with band size $\beta > BD$, such as in Figure 2.3(a), to the Vitter-Shriver layout with band size BD is itself a BPC permutation. The complement vector


band
band
band
band
band

(a)

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

(b)

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

Figure 2.3: Two banded layouts with $P = 8$ processors and a track size of $BD = 32$. Track boundaries are drawn with heavy lines. **(a)** The band size is $\beta = 64$. **(b)** The band size is $\beta = BD = 16$.

is all 0s, and the bit permutation $\pi_{\beta > BD}$ is given by

$$\pi_{\beta>BD}(j) = \begin{cases} j & \text{if } 0 \le j \le \lg \dfrac{BD}{P} - 1 \text{ or } \lg\beta \le j \ , \\[2ex] \lg \dfrac{BD}{P} + \left((j - \lg BD) \bmod \lg \dfrac{\beta P}{BD} \right) & \text{if } \lg \dfrac{BD}{P} \le j \le \lg\beta - 1 \ . \end{cases} \tag{2.34}$$

Let us see why mapping (2.34) works. To start, observe that each record stays within its group of β records. That is, records do not cross band boundaries. The only address bits that change, therefore, are the least significant $\lg\beta$. Hence, $\pi_{\beta>BD}(j) = j$ for $j \ge \lg\beta$.

We treat each β-record band as a matrix in which each entry actually contains several elements, and we transpose this matrix. Elements that are in the same track and processor travel together in the mapping. That is, each group of BD/P elements (2 in Figure 2.3) travels together. Because the BD/P elements stay in the same order relative to each other, the least significant $\lg(BD/P)$ address bits do not change, and so $\pi_{\beta>BD}(j) = j$ for $0 \le j \le \lg(BD/P) - 1$. We treat each β-record band as a $\beta/BD \times P$ matrix, with each matrix entry consisting of BD/P records. We transpose this matrix. In the example of Figure 2.3, we perform the following transpose of a 4×8 matrix, where each entry consists of two records:

$$\begin{bmatrix} (0,1) & (8,9) & (16,17) & (24,25) & (32,33) & (40,41) & (48,49) & (56,57) \\ (2,3) & (10,11) & (18,19) & (26,27) & (34,35) & (42,43) & (50,51) & (58,59) \\ (4,5) & (12,13) & (20,21) & (28,29) & (36,37) & (44,45) & (52,53) & (60,61) \\ (6,7) & (14,15) & (22,23) & (30,31) & (38,39) & (46,47) & (45,55) & (62,63) \end{bmatrix}$$
$$\Downarrow$$
$$\begin{bmatrix} (0,1) & (2,3) & (4,5) & (6,7) \\ (8,9) & (10,11) & (12,13) & (14,15) \\ (16,17) & (18,19) & (20,21) & (22,23) \\ (24,25) & (26,27) & (28,29) & (30,31) \\ (32,33) & (34,35) & (36,37) & (38,39) \\ (40,41) & (42,43) & (44,45) & (46,47) \\ (48,49) & (50,51) & (52,53) & (54,55) \\ (56,57) & (58,59) & (60,61) & (62,63) \end{bmatrix}$$

Each consecutive group of BD/P rows (2 in this example) of this transposed matrix comprises a track of the result.

The transpose is of a $\beta/BD \times P$ matrix, affecting address bits $\lg(BD/P)$ through $\lg\beta - 1$. We adapt equation (2.2), but with P columns, $\beta P/BD$ elements altogether, and adjusting the first bit position to $\lg(BD/P)$ instead of 0:

$$\begin{aligned} \pi_{\beta>BD}(j) &= \lg\frac{BD}{P} + \left(\left(\left(j - \lg\frac{BD}{P}\right) - \lg P\right) \bmod \lg\frac{\beta P}{BD}\right) && (2.35) \\ &= \lg\frac{BD}{P} + \left((j - \lg BD) \bmod \lg\frac{\beta P}{BD}\right) && (2.36) \end{aligned}$$

for $\lg(BD/P) \le j \le \lg\beta - 1$.

If instead we had $\beta < BD$ and we wished to map a vector with band size β to one with band size BD, we would interchange BD and β in equations (2.34)–(2.36). We would also have to change equation (2.35) to also add $\lg P$ instead of subtract it:

$$\begin{aligned} \pi_{\beta<BD}(j) &= \lg\frac{\beta}{P} + \left(\left(\left(j - \lg\frac{\beta}{P}\right) + \lg P\right) \bmod \lg\frac{BDP}{\beta}\right) \\ &= \lg\frac{\beta}{P} + \left(\left(j + \lg\frac{P^2}{\beta}\right) \bmod \lg\frac{BDP}{\beta}\right) \end{aligned}$$

for $\lg(\beta/P) \le j \le \lg BD - 1$. We then have

$$\pi_{\beta<BD}(j) = \begin{cases} j & \text{if } 0 \le j \le \lg \dfrac{\beta}{P} - 1 \text{ or } \lg BD \le j \ , \\ \lg \dfrac{\beta}{P} + \left(\left(j + \lg \dfrac{P^2}{\beta}\right) \bmod \lg \dfrac{BDP}{\beta}\right) & \text{if } \lg \dfrac{\beta}{P} \le j \le \lg BD - 1 \ . \end{cases} \tag{2.37}$$

Performing BMMC and BPC permutations by adjusting the characteristic matrix and complement vector

In the remainder of this section, we solve the following problem. We are given a mapping f to perform; this mapping is a BMMC or BPC permutation. The vector we wish to permute is laid out according to some scheme other than the Vitter-Shriver one. A BMMC mapping g maps this *actual layout* to the Vitter-Shriver layout. How do we perform the mapping f under these conditions?

In effect, we wish to simulate the BMMC mapping f by another mapping h that takes the actual layout into account. We shall see in a moment that h is a BMMC permutation if f and g are and, in addition, h is a BPC permutation if f and g are. We describe the mapping h in three steps. First, we permute the vector according to mapping g to convert the vector from the actual layout to Vitter-Shriver order. Second, we perform mapping f. Third, we permute the vector according to g^{-1} to restore the actual layout order. Thus,

$$h(x) = g^{-1}(f(g(x))) \ . \tag{2.38}$$

To determine the nature of the mapping h, we define the characteristic matrices and complement vectors for the BMMC mappings f and g by

$$\begin{aligned} f(x) &= A_f x \oplus c_f \ , \\ g(x) &= A_g x \oplus c_g \ . \end{aligned}$$

Note that because g is a BMMC permutation, the matrix A_g^{-1} exists. Since $g(x) = A_g x \oplus c_g$, we have that $g^{-1}(x) = A_g^{-1}(x \oplus c_g)$. We are now ready to expand equation (2.38):

$$\begin{aligned} h(x) &= g^{-1}(f(g(x))) \\ &= A_g^{-1}((A_f(A_g x \oplus c_g) \oplus c_f) \oplus c_g) \\ &= A_g^{-1}(A_f A_g x \oplus A_f c_g \oplus c_f \oplus c_g) \\ &= (A_g^{-1} A_f A_g)x \oplus A_g^{-1}((A_f \oplus I)c_g \oplus c_f) \ . \end{aligned}$$

We conclude that the mapping h is a BMMC permutation whose characteristic matrix A_h is the product $A_g^{-1} A_f A_g$ and whose complement vector c_h is $A_g^{-1}((A_f \oplus I)c_g \oplus c_f)$. Moreover, if f and g are BPC permutations, then each of the matrices A_g^{-1}, A_f, and A_g is a permutation matrix. The product A_h is then also a permutation matrix. In this case, the mapping h is also a BPC permutation.

Thus, we can perform a BMMC or BPC permutation with a different layout order without changing the algorithm.

2.10 Conclusions

In this section, we recap the results in this chapter and suggest related research topics.

We have shown that many permutations can be performed on parallel disk systems faster than sorting. In particular, we generalized the matrix-tranpose algorithm of Vitter and Shriver to BPC permutations and BMMC permutations. These are broad classes that include many commonly used permutations: matrix transpose (or cyclic rotation of address bits), bit-reversal, hypercube, vector-reversal, Gray codes, inverse Gray codes, and permutations that help reduce data-access conflicts. Along the way, we came up with restricted permutations—MRC, block BPC, and block BMMC—that can be performed in just one pass over the records. We also showed how to perform arbitrary block permutations with off-line scheduling. We proved lower bounds for BPC and BMMC permutations, showing that our algorithm for BPC permutations is asymptotically optimal. Finally, we showed how to perform BMMC and BPC permutations without changing the algorithm when the vector to be permuted violates the Vitter-Shriver layout scheme.

The VM-DP system includes an implementation of the BPC algorithm. As we shall see in Chapter 3, its actual performance matches the performance predicted in this chapter exactly.

Future research

We have seen that the $\Omega\left(\min\left(\frac{N}{D}, \frac{N}{BD}\frac{\lg(N/B)}{\lg(M/B)}\right)\right)$ lower bound for general permutations fails to apply for a fairly broad class of permutations. We shall see even more classes that we can perform quickly in Chapter 3. How broadly applicable then is this lower bound? Can we succinctly characterize classes of permutations for which it does not apply?

Is the BPC algorithm in this chapter optimal? It is asymptotically optimal, but what about the constant factors? They are small, but are they the best possible? The lower bounds proven in Section 2.8 are only asymptotic lower bounds. Can we determine the constants? The key to doing so, at least using the framework of Section 2.8, is to replace the asymptotic notation in Lemma 2.20 by an expression with exact constants. How can we adapt the Aggarwal-Vitter [AV88] argument that establishes this lemma to use constants rather than asymptotics?

There is a gap in the known asymptotics for BMMC permutations. Where in this gap is their true complexity? It depends on the exact nature of the submatrix denoted by $\gamma\alpha^{-1}$ in equation (2.21). As we observed in Section 2.6, when $\gamma\alpha^{-1} = 0$, the cost of performing $\widehat{A}$ includes only the cost of performing the MRC permutation W, and so it is constant. It seems that the fewer nonzero rows or columns $\gamma\alpha^{-1}$ contains, the faster it can be performed. Can we precisely characterize this property?

Chapter 3

Performing General and Special Permutations

We saw in Chapter 2 that if we carefully choreograph the disk I/O in virtual-memory systems for data-parallel computing, we can reduce the cost of performing bit-defined permutations. This chapter continues our examination of the differences between performing general and special permutations.

This chapter takes a more practical and slightly less theoretical approach than Chapter 2. It presents a simple, though asymptotically suboptimal, algorithm to perform general permutations. This algorithm uses $\Theta\left(\frac{N}{BD}\frac{\lg N}{\lg(M/BD)}\right)$ parallel I/Os by performing external radix sort on target addresses. Then this chapter explores several classes of special permutations that we can perform much faster than general permutations:

- Monotonic routes, which we can perform with $\lceil N_s/BD \rceil + 2\lceil N_t/BD \rceil$ parallel I/Os for a source vector of N_s elements and a target vector of N_t elements.

- k-monotonic routes, which are the superposition of k monotonic routes and can be performed with $\lceil N_s/BD \rceil + 2k\lceil N_t/BD \rceil$ parallel I/Os, subject to the restriction that $(k+1)BD \leq M$.

- Mesh permutations, which are a special case of monotonic routes and can be performed with $3\lceil N/BD \rceil$ parallel I/Os.

- Torus permutations in d dimensions, which are a special case of 2^d-monotonic routes and can be performed with $(2^{d+1}+1)\lceil N/BD \rceil$ parallel I/Os if $(2^d+1)BD \leq M$.

- BMMC and BPC permutations, which we saw in Chapter 2.

- General matrix transpose of any $R \times S$ matrix, which we can perform with less than $9\,\frac{RS}{BD}\left\lceil\frac{\lg\min(R,S,B,RS/B)}{\lg(M/B)}\right\rceil + \frac{53}{2}\,\frac{RS}{BD} + 11$ parallel I/Os, and often fewer.

These classes arise frequently. We show, for each of these classes, not only how to perform them quickly, but also how to detect them at run-time given a vector of target addresses using barely more than N/BD parallel I/Os.

We then focus on the question of how to invoke special permutations. We shall argue that although we can detect these special permutations quickly at run time, it is better to invoke them by specifying them in the source code. We support this argument with empirical data for BPC permutations.

Importance of permuting in a data-parallel system with virtual memory

Just as communication costs dominate parallel computers, permuting costs dominate data-parallel computing with virtual memory. Common operations, such as elementwise operations, scans (parallel prefix), and reduces, can be performed in one pass through the data. As we saw in Section 1.4, however, permuting in general is much more expensive, requiring more than a constant number of passes.

Suppose we examine a data-parallel program running on a parallel machine at the level of VCODE or Paris on the CM-2 [Thi89]. Although it may be the case that only a few operations permute data, the cost of the entire computation may very well be dominated by the cost of performing the few permutations. When the data must reside on disk, the relative cost of permuting is even greater. For any hope of reasonable performance, we must perform permutations as fast as possible.

Outline of this chapter

The remainder of this chapter is organized as follows. Section 3.1 presents how the VM-DP system performs general permutations, sorting target addresses by an external radix sort. Section 3.2 is an overview of several special permutations that Sections 3.3–3.5 show how to perform faster than general permutations and also detect quickly at run time. We argue in Section 3.6 that it is best to invoke these permutations at the source-code level. The argument includes a case study of how the VM-DP system performs BPC permutations, with empirical statistics. Finally, Section 3.7 contains some concluding remarks and discusses future work.

3.1 Performing general permutations

The VM-DP system performs general permutations by sorting records according to their associated target addresses. The asymptotically optimal algorithms of Vitter and Shriver [VS90a, VS90b] and Nodine and Vitter [NV90, NV91, NV92] sort N records using $\Theta\left(\frac{N}{BD}\frac{\lg(N/B)}{\lg(M/B)}\right)$ parallel I/Os. These algorithms, however, are rather complicated to implement. We instead opted for simplicity, sorting target addresses with an external radix sort. This method uses $\Theta\left(\frac{N}{BD}\frac{\lg N}{\lg(M/BD)}\right)$ parallel I/Os.

Radix sort is an old sorting algorithm. See [CLR90] for a detailed description. It works as follows. The N keys to be sorted are assumed to be k-bit integers. When the keys are indices of an N-element vector, $k = \lceil \lg N \rceil$. We form β buckets, where β is an integer power of 2. We then make $\lceil k/\lg\beta \rceil$ passes over the keys, placing each key into a bucket according to the value of $\lg\beta$ of its bits. In particular, in the jth pass (starting from 0), a record with key x is placed into bucket number $\lfloor x/\beta^j \rfloor \bmod \beta$. This scheme sorts from the least significant bits first. As long as each pass is *stable*—if key x preceeds key y in a bucket after the $(j+1)$st pass, then x preceeded y in the total order after the jth pass—this method correctly sorts the keys in $\lceil k/\lg\beta \rceil$ passes.

External radix sort

The VM-DP system uses an external version of this algorithm that performs only striped I/O. Each key is a target address and has some associated data. Although we move each key and its data together, they are elements of separate vectors. For the purpose of this exposition, we assume that keys are 4-byte integers and so is the data. A *track frame* consists of BD records in RAM, and each bucket is two track frames, one for the data and one for the keys, occupying $2BD$ records in RAM. Although it seems that we ought to be able to set the number of buckets β to $M/2BD$, we cannot. We must instead reserve an additional bucket's worth of track frames

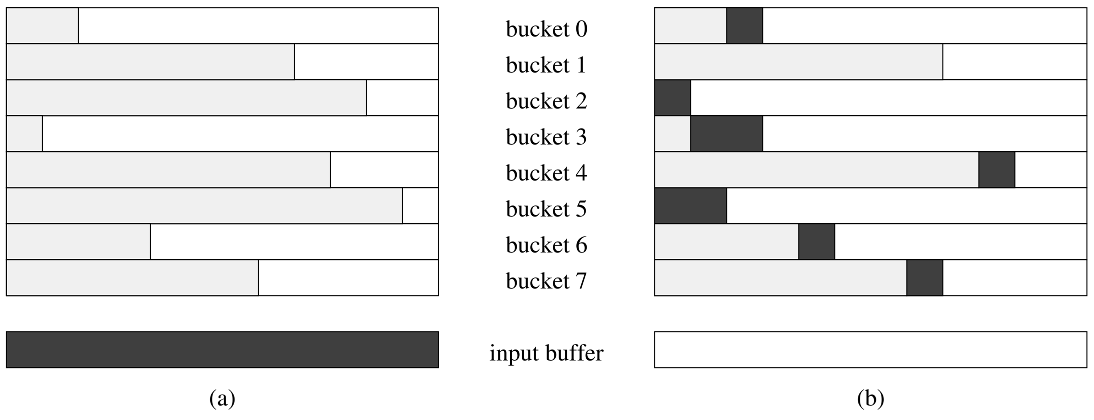


Figure 3.1: The state of RAM **(a)** before and **(b)** after distributing the records of an input buffer to $\beta = 8$ buckets. Light shading represents records in buckets before distribution, and dark shading represents the new records to be distributed. Buckets 2 and 5 have only new records, and fewer of them, afterward because they were filled during the distribution, written out to disk, emptied in RAM, and then more records were routed to them.

as input buffers for keys and data. In order to keep everything a power of 2, therefore, we set $\beta = M/4BD$.

Each pass processes $\lg \beta = \lg(M/4BD)$ bits of the keys. Initially, all N records are in one bucket. The jth pass, for $j = 0, 1, \ldots, \lceil(\lg N)/(\lg \beta)\rceil - 1$, processes β *input buckets*, which are the *output buckets* of the $(j-1)$st pass, in order. That is, it processes input bucket 0 from beginning to end, then input bucket 1, and so on up to input bucket $\beta - 1$. (The pass for $j = 0$ processes the lone initial bucket.) All β output buckets are empty at the beginning of each pass. Input buckets are read into the input buffers a track at a time; we read a track of BD keys and a track of their associated BD data elements. As an optimization, we can check whether all keys are sorted after each pass, thereby avoiding unnecessary passes when the sorting completes before all $\lceil \lg N \rceil$ bits are processed.

After reading each track, we distribute its records to the β buckets according to the appropriate bits of the keys. As Figure 3.1 shows, we loop through buckets $i = 0, 1, \ldots, \beta - 1$, finding all the records in the input buffer with key x such that $\lfloor x/\beta^j \rfloor \bmod \beta = i$. That is, we find the keys for which the jth group of $\lg \beta$ bits, starting from the least significant position, is the binary representation of i. We perform an internal (in-RAM) route of these keys and their data into the next available positions in bucket i.[1] If this route fills up bucket i in RAM, we write it out to disk, empty it in RAM, and continue on.

All writes are to a different set of tracks from the tracks containing the input buckets. We call the tracks containing the input buckets the *input portion* of the parallel disk system, and the *output portion* contains the output tracks. Since we need the output buckets of a pass only as the input buckets to the next pass, we swap the input and output portions from pass to pass.

Having read in and processed all N records in the input buckets, we are left with many tracks having been written out to disk and up to β buckets that are partially full in RAM. We write these partially full buckets out to disk, and we are then ready for the next pass.

We must perform some special processing before the first pass and after the last pass. In order to know where on disk to write each track of each output bucket, we need to know how much disk space is needed to hold each bucket. Moreover, we need to know this information in each pass. Before the first pass (pass $j = 0$), therefore, we read all the keys, creating a census of how many records will end up in each bucket in each pass. With one read pass through the records, we create a $\lceil (\lg N)/(\lg \beta) \rceil \times \beta$ array of this census information, which we use to allocate disk space. After the last pass, we have β buckets on disk, but the last track of many of these buckets may be only partially full.[2] Therefore, we perform one more pass through the buckets to compact the keys and data.

Analysis of external radix sort

We now analyze the asymptotic I/O behavior of external radix sort. Although it is not difficult to determine the constant factors in general, it is somewhat tedious and yields little additional insight. In Section 3.6, we determine the constant factors for the case in which the keys are the set $\{0, 1, \ldots, N-1\}$ and N is a power of 2.

External radix sort uses $\Theta\left(\frac{N}{BD}\frac{\lg N}{\lg(M/BD)}\right)$ parallel I/Os. Each bucket may have one partially full track, so the number of parallel reads for keys and for data per pass is at most $\lceil N/BD \rceil + \beta$

[1]Because we require radix sort to be stable, this route must be monotonic (see Section 3.3).

[2]If N is a power of 2 and the keys are the set $\{0, 1, \ldots, N-1\}$, all buckets will be full after each pass. The routines that call external radix sort in the VM-DP system, however, cannot assume that N is a power of 2, that all keys are distinct, or even that all values between 0 and $N-1$ appear in the set of keys.

each. The same holds for the number of parallel writes. Thus, there are $\Theta(N/BD + \beta)$ parallel I/Os per pass. Presumably, we invoke external radix sort only if N is large enough that the keys and data do not fit in RAM. Let us assume therefore that $2N > M$, which implies that $\beta = M/4BD < N/2BD$ and thus $\Theta(N/BD + \beta) = \Theta(N/BD)$. Thus, there are $\Theta\left(\frac{N}{BD}\frac{\lg N}{\lg(M/BD)}\right)$ parallel I/Os in all $\left\lceil\frac{\lg N}{\lg(M/4BD)}\right\rceil$ passes together. The remaining parallel I/Os number only $\Theta(N/BD)$. There are $\lceil N/BD \rceil$ parallel reads in the census count, and compaction takes $\Theta(N/BD)$ parallel I/Os. The total number of I/Os used by external radix sort is therefore $\Theta\left(\frac{N}{BD}\frac{\lg N}{\lg(M/BD)}\right)$.

Asymptotically, the external radix sort bound is worse than the optimal I/O bound because it has a numerator of $\lg N$ rather than $\lg(N/B)$ and a denominator of $\lg(M/BD)$ rather than $\lg(M/B)$. External radix sort is much simpler to implement than the optimal sorting algorithms, however. Moreover, it uses only striped I/O, unlike the optimal algorithms, which use independent I/O.

3.2 Special permutations

In the three following sections, we explore some classes of permutations that can be performed faster than general permutations. In particular, we examine monotonic routes, mesh and torus permutations, BMMC and BPC permutations, and general matrix transpose. In addition to being performed quickly, these permutations have two additional qualities: they occur frequently and we can detect them efficiently at run time. We will see how to both perform and detect these classes.

Each permutation is specified by a *source vector*, a *target vector*, and a *mapping* from indices in the source vector to indices in the target vector. Two of them—monotonic routes and mesh permutations—are partial permutations rather than full permutations. In a *partial permutation*, the mapping does not necessarily cover all indices in the source vector in its domain, nor does it necessarily cover all indices in the target vector in its range.

Many of the run-time detection schemes take as input a vector of N target addresses and follow a three-stage strategy:

1. Decide which class of special permutation we are trying to detect. The detection schemes vary among the special permutations.

2. Read target addresses for a small number of individual elements. From these target addresses, hypothesize the particular parameters of the permutation.

3. Read target addresses for all elements in order to verify that the permutation is indeed described by the hypothesized parameters. We can terminate this stage early, thus saving parallel I/Os, if any source-target address pair fails the verification test. We must check all N elements, however, if all pairs pass the test. Although checking all N elements entails $\lceil N/BD \rceil$ parallel I/Os, we more than compensate for this cost by running the appropriate special permutation code rather than the general permutation routine.

To minimize the number of passes through the target-address vector, we can perform the verification step simultaneously for all the permutation classes under consideration.

3.3 Monotonic routes, mesh permutations, and torus permutations

In this section, we see how to perform and detect monotonic routes and a generalization called k-monotonic routes. For a source vector of N_s elements and a target vector of N_t elements, we can perform a monotonic route using $\lceil N_s/BD \rceil + 2 \lceil N_t/BD \rceil$ parallel I/Os, and we can perform a k-monotonic route using $\lceil N_s/BD \rceil + 2k \lceil N_t/BD \rceil$ parallel I/Os. We shall see how to detect monotonic and k-monotonic routes in one pass over the mapping. We then see how mesh permutations are special cases of monotonic routes and d-dimensional torus permutations are special cases of 2^d-monotonic routes. Therefore, we can detect mesh and torus permutations in one pass, we can perform mesh permutations using $3 \lceil N/BD \rceil$ parallel I/Os, and we can perform torus permutations using $(2^{d+1} + 1) \lceil N/BD \rceil$ parallel I/Os.

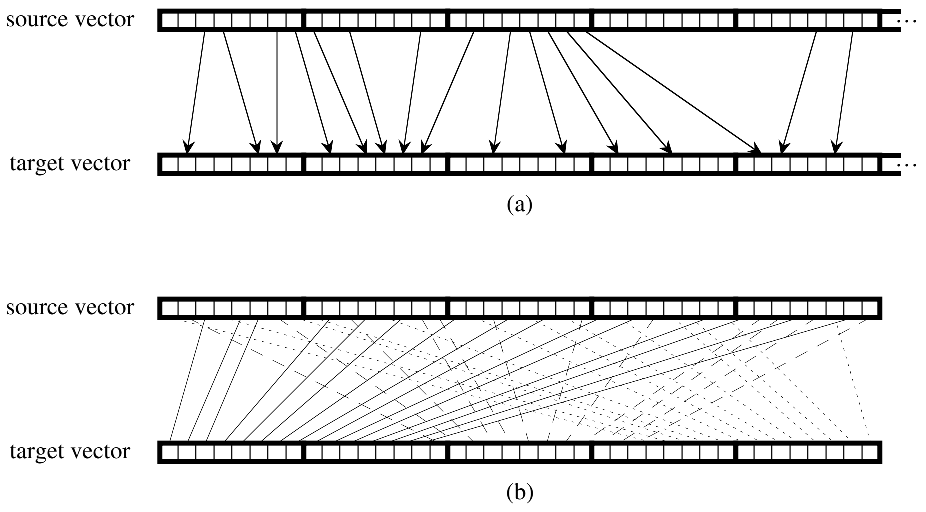


Figure 3.2: **(a)** A monotonic route, with each element mapping indicated by an arrow. No arrows cross in a monotonic route. Heavy lines indicate track boundaries. We can perform any monotonic route by making just one pass over the source and target vectors. **(b)** A k-way split operation is an example of a k-monotonic route, shown here for $k = 3$. The three component monotonic routes are drawn with solid, dashed, and dotted lines, and arrowheads are omitted.

Monotonic routes

A *monotonic route* is a partial permutation in which if elements i and j are mapped to positions i' and j', respectively, then

$$i < j \text{ if and only if } i' < j' \ . \tag{3.1}$$

If we view each individual mapping in a monotonic route as one of the arrows in Figure 3.2(a), then no arrows cross.

Because elements that are routed appear in the same relative order in the source and target vectors, we can perform a monotonic route in only one pass over the source and target vectors. We perform the individual mappings (arrows) in order, reading and writing tracks of the source and target vectors as necessary. Since no arrows cross, tracks are read and written in order. We read each track of the source vector once, and we read and write each track of the target vector once. (We have to read target-vector tracks to avoid overwriting data in positions that are not routed to.) Thus, we can perform a monotonic route with a source vector of length N_s and a target vector of length N_t using at most $\lceil N_s/BD \rceil + 2 \lceil N_t/BD \rceil$ parallel I/Os.

k-monotonic routes

A *k-monotonic route* is the superposition of k monotonic routes. The source and target vectors of the k routes are the same, but the domains of the routes are disjoint, as are their ranges. As Figure 3.2(b) shows, a k-monotonic route can be used to implement a *k-way split operation.* Here, each element is assigned to one of k partitions, and the vector is permuted to place all elements in the same partition together.

We can perform a k-monotonic route as k individual (1-)monotonic routes, using a total of $k\lceil N_s/BD\rceil + 2k\lceil N_t/BD\rceil$ parallel I/Os, but in fact we can do somewhat better. Observe that if there is enough RAM to hold one track of the source vector plus k tracks of the target vector (one for each component monotonic route), we can arrange for one read pass to serve all k of the component monotonic routes. We have to take care to ensure that each target track appears in RAM only once at a time, and we read and write each target track up to k times. As long as $(k+1)BD \le M$, therefore, we can perform a k-monotonic route with at most $\lceil N_s/BD\rceil + 2k\lceil N_t/BD\rceil$ parallel I/Os.

For a k-way split operation, as in Figure 3.2(b), we can do even better. Each target track that contains at most one of the split partitions needs to be read and written only once. Target tracks that contain portions of more than one partition need to be read and written more than once, but there are at most $k-1$ such reads and writes. We can perform a k-way split, therefore, with at most $\lceil N_s/BD\rceil + 2\lceil N_t/BD\rceil + 2(k-1)$ parallel I/Os. When $k=1$, this bound is the same as the one above for monotonic routes.

Detecting monotonic and k-monotonic routes

Monotonic routes are the simplest of all permutations or partial permutations to detect. We simply verify that condition (3.1) holds for all source-target pairs.

Detecting k-monotonic routes, for $k>1$, is harder. Trivially, we can let k be the size of the domain of the mapping, but the domain size is often too large for this value of k to be useful in performing the route. To determine a more reasonable value of k, we adapt Pinter's method [Pin82, Section V.2.2] for sequentially determining the minimum number of layers needed for river routing two-terminal nets in a channel. We consider only the domain of the mapping, i.e., indices in the source vector that map to the target vector. For the ith such index in the source vector, let $\pi(i)$ be the position it maps to in the target vector. We will assign to each mapping i the number $R(i)$ of the monotonic route it belongs to. At each point in the algorithm, we maintain the k component routes formed so far. For the jth such monotonic route, we define $\pi_{\max}(j) = \max\{\pi(i) : R(i) = j\}$. In other words, $\pi_{\max}(j)$ is the highest target address in the jth component route. When we process a mapping of i to $\pi(i)$, we assign i to the component route j for which $\pi_{\max}(j) < \pi(i)$ (so that the component route, extended by this mapping, remains monotonic) and $\pi_{\max}(j)$ is maximum over all such routes. If no existing component routes can be extended by the mapping, we create a new one and increment k. The number k of component routes at the end is the minimum possible.

Pinter's method requires only one pass over the target-address vector, so it is efficient in terms of disk-access cost. We omit discussion of the processing cost, since we are more concerned with I/O. Pinter observed, however, that we can use binary search to find the appropriate component route in $O(\lg k)$ time sequentially for each individual mapping.

Mesh and torus permutations

Many applications use data that is organized into multidimensional grids, or meshes. A class of permutation commonly performed in d-dimensional meshes adds an offset $o = (o_1, o_2, \ldots, o_d)$ to the element originally in position $p = (p_1, p_2, \ldots, p_d)$, mapping it to position

$$\text{mesh}(p, o) = (p_1 + o_1, p_2 + o_2, \ldots, p_d + o_d) \ .$$

When the number of dimensions is $d = 1$, this type of permutation is a shift operation. Some people think of mesh permutations exclusively in terms of the offset o as a unit vector; the above definition generalizes this view.

There are two common ways to handle boundary conditions. Let the dimensions of the mesh be $m = (m_1, m_2, \ldots, m_d)$, with positions in dimension i indexed from 0 to $m_i - 1$, so that $-m_i < o_i < m_i$ for $i = 1, 2, \ldots, d$. One choice is to not map elements that would be mapped across a boundary. That is, map only elements p for which $0 \leq p_i + o_i < m_i$. In this case, only $(m_1 - |o_1|)(m_2 - |o_2|) \cdots (m_d - |o_d|)$ elements are actually mapped, and the mapping does not cover all indices in the source or target vectors. We call this type of partial permutation a *mesh permutation*. The other common choice is a full permutation called a *torus permutation*, in which we wrap around at the boundaries:

$$\text{torus}(p, o, m) = ((p_1 + o_1) \bmod m_1, (p_2 + o_2) \bmod m_2, \ldots, (p_d + o_d) \bmod m_d) \ .$$

We will show that mesh permutations are monotonic routes and that torus permutations are 2^d-monotonic routes. First, however, we need to define the mapping from grid locations to positions in the underlying one-dimensional vector. We assume that we store a d-dimensional grid in row-major order, with 0-origin indexing in each dimension. Under these assumptions, we define the indexing function ι mapping a grid position $p = (p_1, p_2, \ldots, p_d)$ to its corresponding index in row-major order:

$$\iota(p, m) = \sum_{i=1}^{d} \left[\left(\prod_{j=i+1}^{d} m_j \right) p_i \right] \ . \tag{3.2}$$

For a 2-dimensional mesh, for example, equation (3.2) reduces to the familiar form $\iota(p, m) = m_2 p_1 + p_2$, where m_2 is the number of columns.

Mesh permutations as monotonic routes

We now show that mesh permutations are monotonic routes. In fact, as the following lemma shows, mesh permutations are a special type of monotonic route in which the difference between the positions in the row-major order of the source and target addresses is the same for each element permuted.

Lemma 3.1 *Let $p = (p_1, p_2, \ldots, p_d)$ be any grid location mapped by a mesh permutation with offset $o = (o_1, o_2, \ldots, o_d)$ on a d-dimensional grid with dimensions $m = (m_1, m_2, \ldots, m_d)$. Then $\iota(\text{mesh}(p, o), m) - \iota(p, m) = \iota(o, m)$.*

Proof: We have

$$\begin{aligned}
\iota(\text{mesh}(p,o),m) - \iota(p,m) &= \sum_{i=1}^{d}\left[\left(\prod_{j=i+1}^{d} m_j\right)(p_i + o_i)\right] - \iota(p,m) \\
&= \sum_{i=1}^{d}\left[\left(\prod_{j=i+1}^{d} m_j\right)p_i\right] + \sum_{i=1}^{d}\left[\left(\prod_{j=i+1}^{d} m_j\right)o_i\right] - \iota(p,m) \\
&= \iota(p,m) - \iota(o,m) - \iota(p,m) \\
&= \iota(o,m) \ ,
\end{aligned}$$

which completes the proof. ■

Because mesh permutations are monotonic routes, we can perform them by reading each track at most once and writing each track at most once. The total number of parallel I/Os is thus at most $3\lceil N/BD \rceil$, where $N = m_1 m_2 \cdots m_d$ is the total number of elements in the grid.

Torus permutations as 2^d-monotonic routes

We now show that torus permutations are not monotonic routes, but they are 2^d-monotonic routes. They are not monotonic routes because of wraparound. Consider a 1-dimensional torus, for example, with dimension m. In a 1-dimensional torus, $\iota(p,m) = p$ for all $p = 0, 1, \ldots, m-1$. Consider the source addresses 0 and 1, so that $\iota(0,m) = 0$ and $\iota(1,m) = 1$. Let the offset of the permutation be $o = m-1$. Then $\iota(\text{torus}(0, m-1, m), m) = \text{torus}(0, m-1, m) = m-1$ and $\iota(\text{torus}(1, m-1, m), m) = \text{torus}(1, m-1, m) = 0$. Thus, we see that $\iota(0,m) < \iota(1,m)$ but $\iota(\text{torus}(0, m-1, m), m) > \iota(\text{torus}(1, m-1, m), m)$, and so the torus permutation is not a monotonic route.

Intuitively, a torus permutation is a 2^d-monotonic route because if we partition the source addresses into 2^d sets according to whether or not they wrap around in each dimension, each such set forms a monotonic route. To formalize this notion, we assume without loss of generality that for each dimension $i = 1, 2, \ldots, d$, offset o_i is nonnegative. For each dimension $i = 1, 2, \ldots, d$, we define

$$\mathrm{wrap}_i(p, o, m) = \begin{cases} 0 & \text{if } (p_i + o_i) \bmod m_i \geq p_i \ , \\ 1 & \text{if } (p_i + o_i) \bmod m_i < p_i \ , \end{cases}$$

and we define the vector

$$\mathrm{wrap}(p, o, m) = (\mathrm{wrap}_1(p, o, m), \mathrm{wrap}_2(p, o, m), \ldots, \mathrm{wrap}_d(p, o, m)) \ .$$

For each vector $j = (j_1, j_2, \ldots, j_d) \in \{0, 1\}^d$, we define the set

$$W(j, o, m) = \{p : \mathrm{wrap}(p, o, m) = j\} \ ;$$

that is, $W(j, o, m)$ is the set of grid locations p for which the vector j describes whether the torus permutation causes it to wrap around in each dimension. Next, we define modified offset values $o'(j, o, m)$, which may now be negative, that describe the actual offsets used for each set $W(j, o, m)$. For $j = (j_1, j_2, \ldots, j_d) \in \{0, 1\}^d$ and $i = 1, 2, \ldots, d$,

$$o'_i(j, o, m) = \begin{cases} o_i & \text{if } j_i = 0 \ , \\ o_i - m_i & \text{if } j_i = 1 \ , \end{cases}$$

and we define

$$o'(j, o, m) = (o'_1(j, o, m), o'_2(j, o, m), \ldots, o'_d(j, o, m)) \ .$$

The following lemma shows that for each set $W(j, o, m)$, the offset vector $o'(j, o, m)$ describes a mesh permutation that is equivalent to the torus permutation performed on the grid locations in the set.

Lemma 3.2 *For each $j = (j_1, j_2, \ldots, j_d) \in \{0, 1\}^d$ and for all grid locations $p \in W(j, o, m)$,*

$$\mathrm{torus}(p, o, m) = \mathrm{mesh}(p, o'(j, o, m)) \ .$$

Proof: Consider a given value of $j \in \{0, 1\}^d$ and any dimension $i \in \{1, 2, \ldots, d\}$.

If $j_i = 0$, then $o'_i(j, o, m) = o_i$ and so the ith position of $\text{mesh}(p, o'(j, o, m))$ is $p_i + o_i$. But by the definition of the set $W(j, o, m)$, if $j_i = 0$ then $\text{wrap}_i(p, o, m) = 0$. This in turn implies that $(p_i + o_i) \bmod m_i \geq p_i$, and thus $(p_i + o_i) \bmod m_i = p_i + o_i$. Therefore, $\text{torus}(j, o, m)$ and $\text{mesh}(p, o'(j, o, m))$ agree in the ith dimension.

If instead $j_i = 1$, then $o'_i(j, o, m) = o_i - m_i$ and so the ith position of $\text{mesh}(p, o'(j, o, m))$ is $p_i + o_i - m_i$. By the definition of the set $W(j, o, m)$, if $j_i = 1$ then $\text{wrap}_i(p, o, m) = 1$. This in turn implies that $(p_i + o_i) \bmod m_i < p_i$, and thus $(p_i + o_i) \bmod m_i = p_i + o_i - m_i$. Therefore, $\text{torus}(j, o, m)$ and $\text{mesh}(p, o'(j, o, m))$ agree in the ith dimension.

In either case, we have shown that $\text{torus}(j, o, m)$ and $\text{mesh}(p, o'(j, o, m))$ are equal in each dimension. ■

Finally, we show that each set $W(j, o, m)$ defines a monotonic route.

Lemma 3.3 *For each $j = (j_1, j_2, \ldots, j_d) \in \{0, 1\}^d$ and for all grid locations $p \in W(j, o, m)$,*

$$\iota(\text{torus}(p, o, m), m) - \iota(p, m) = \iota(o'(j, o, m), m) \ .$$

Proof: By Lemmas 3.1 and 3.2,

$$\begin{aligned} \iota(\text{torus}(p, o, m), m) - \iota(p, m) &= \iota(\text{mesh}(p, o'(j, o, m))) - \iota(p, m) \\ &= \iota(o'(j, o, m), m) \ , \end{aligned}$$

which completes the proof. ■

Because each set $W(j, o, m)$ defines a monotonic route and there are 2^d such sets, we conclude that a d-dimensional torus permutation is a 2^d-monotonic route. As long as $(2^d + 1)BD \leq M$, therefore, we can perform a d-dimensional torus permutation on $N = m_1 m_2 \cdots m_d$ elements using at most $(2^{d+1} + 1) \lceil N/BD \rceil$ parallel I/Os.

In fact, when all dimensions are powers of 2, the number of parallel I/Os needed to perform a torus permutation is independent of the number of dimensions. Section 4.3 shows that as long as vectors are laid out with element X_k residing in track number $\lfloor k/BD \rfloor$, the elements of each source-vector track map to at most 2 target-vector tracks for any torus permutation regardless of the number of dimensions. When all dimensions are powers of 2, therefore, we can perform any torus permutation using only $5N/BD$ parallel I/Os.

3.4 BMMC and BPC permutations

Chapter 2 defined the classes of BMMC and BPC permutations, demonstrated that these classes include many useful classes of permutations, and presented efficient algorithms to perform them. In this section, we show how to detect BMMC permutations using only $N/BD + \left\lceil \frac{\lg(N/B)+1}{D} \right\rceil$ parallel reads.

Once we can detect BMMC permutations, we can detect the other classes in Chapter 2 that are defined by characteristic matrices. The BMMC detection method below determines a candidate characteristic matrix and a candidate complement vector. To detect a subclass of BMMC permutations, we check that the characteristic matrix is of the correct form for the subclass. For BPC permutations, for example, we would check that each row and each column contains exactly one 1.

Detecting BMMC permutations

We detect BMMC permutations by first checking that N is a power of 2; if not, the permutation cannot be BMMC. We then form a candidate characteristic matrix A and complement vector c by a method we are about to see that uses only $\left\lceil \frac{\lg(N/B)+1}{D} \right\rceil$ parallel reads. Next, we check that the matrix A is nonsingular. Many data-parallel machines have an attached front-end machine, which is where this nonsingularity check would typically take place. If there is no front end and the check must occur on the data-parallel machine, A and c together are described by only $\lg^2 N + \lg N$ bits, so they easily fit in RAM. Finally, we verify that A and c define a BMMC permutation by checking for all N source addresses x and all N target addresses y whether $y = Ax \oplus c$. Verification takes N/BD parallel reads, for a total of $N/BD + \left\lceil \frac{\lg(N/B)+1}{D} \right\rceil$ parallel reads for BMMC detection.

The method for forming the candidate characteristic matrix A and candidate complement vector c is based on two observations. First, if the permutation is BMMC, then the complement vector c must be the target address corresponding to source address 0. This relationship holds because $x = 0$ and $y = Ax \oplus c$ imply that $y = c$.

The second observation is as follows. Consider a source address x with binary representation $\text{bin}(x) = (x_0, x_1, \ldots, x_{\lg N-1})$, and suppose that bit position k holds a 1, i.e., $x_k = 1$. Let us denote the jth column for matrix A by A_j. Also, let S_k denote the set of bit positions other than k that hold a 1: $S_k = \{j : j \neq k \text{ and } x_j = 1\}$. If $y = Ax \oplus c$, then we have

$$y = \left(\bigoplus_{j \in S_k} A_j \right) \oplus A_k \oplus c \; , \tag{3.3}$$

since only the bit positions j for which $x_j = 1$ contribute a column of A to the sum of columns that forms the matrix-vector product. If we know the target address y, the complement vector c and the columns A_j for all $j \neq k$, we can rewrite equation (3.3) to yield the kth column of A:

$$A_k = y \oplus \left(\bigoplus_{j \in S_k} A_j \right) \oplus c \; . \tag{3.4}$$

We shall compute the complement vector c first and then the columns of the characteristic matrix A one at a time, from A_0 up to $A_{\lg N-1}$. When computing A_k, we will have already computed $A_0, A_1, \ldots, A_{k-1}$, and these will be the only columns we need in order to apply equation (3.4). In other words, $S_k \subseteq \{0, 1, \ldots, k-1\}$. It is useful to think of each source address as being comprised of three fields of bits: the lower $\lg B$ bits give the record's offset within its block, the middle $\lg D$ bits give the disk number, and the upper $\lg(N/BD)$ bits give the track number.

From equation (3.4), it would be easy to compute A_k if S_k were empty. The set S_k is empty if the source address is a unit vector, with its only 1 in position k. If we look at these addresses, however, we find that the target addresses for a disproportionate number—all but $\lg D$ of them—reside on disk $\mathcal{D}_0$. The block whose disk and track fields are all zero contains $\lg B$

such addresses, so they can be fetched in one disk read. A problem arises for the $\lg(N/BD)$ source addresses with one 1 in the track field: their target addresses all reside on different blocks of disk $\mathcal{D}_0$. Each must be fetched in a separate read. The total number of parallel reads to fetch all these addresses is $\lg(N/BD)+1$.

To achieve only $\left\lceil \frac{\lg(N/B)+1}{D} \right\rceil$ parallel reads, each read fetches one block from each of the D disks. The first parallel read determines the complement vector, the first $\lg B + \lg D = \lg BD$ columns, and the next $D - \lg D - 1$ columns. Each subsequent read determines another D columns, until all $\lg N$ columns have been determined.

In the first parallel read, we do the same as above for the first $\lg BD$ bits. That is, we fetch blocks containing target addresses whose corresponding source addresses are unit vectors with one 1 in the first $\lg BD$ positions. As before, $\lg B$ of them are in the same block on disk $\mathcal{D}_0$. This block also contains address 0, which we need to compute the complement vector. The remaining $\lg D$ are in track number 0 of disks $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_4, \mathcal{D}_8, \ldots, \mathcal{D}_{D/2}$. Having fetched the corresponding target addresses, we have all the information we need to compute the complement vector c and columns $A_0, A_1, \ldots, A_{\lg BD-1}$.

The columns we have yet to compute correspond to bit positions in the track field. If we were to compute these columns in the same fashion as the first $\lg BD$, we would again encounter the problem that all the blocks we need to read are on disk $\mathcal{D}_0$. In the first parallel read, the only unused disks remaining are those whose numbers are not a power of 2 ($\mathcal{D}_3, \mathcal{D}_5, \mathcal{D}_6, \mathcal{D}_7, \mathcal{D}_9, \ldots$). The key observation is that we have already computed all $\lg D$ columns corresponding to the disk field, and we can thus apply equation (3.4). For example, let us compute column $A_{\lg BD}$, which corresponds to the first bit of the track number. We read track 1 on disk $\mathcal{D}_3$ and find the first target address y in this block. Disk number 3 corresponds to the first two disk-number columns, $A_{\lg B}$ and $A_{\lg B+1}$. Applying equation (3.4) with $S_{\lg BD} = \{\lg B, \lg B + 1\}$, we compute $A_{\lg BD} = y \oplus A_{\lg B} \oplus A_{\lg B+1} \oplus c$. The next column we compute is $A_{\lg BD+1}$. Reading the block at track 2 on disk $\mathcal{D}_5$, we fetch a target address y and then compute $A_{\lg BD+1} = y \oplus A_{\lg B} \oplus A_{\lg BD+2} \oplus c$. Continuing on in this fashion, we compute a total of $D - \lg D - 1$ track-bit columns from the first parallel read.

The remaining parallel reads compute the remaining track-bit columns. We follow the track-bit pattern of the first read, but we use all disks, not just those whose disk numbers are not powers of 2. Each block read fetches a target address y, which we exclusive-or with a set of columns from the disk field and with the complement vector to compute a new column from the track field. The first parallel read computes $\lg B + D - 1$ columns and all subsequent parallel reads compute D columns. The total number of parallel reads is thus

$$\begin{aligned} 1 + \left\lceil \frac{\lg N - (\lg B + D - 1)}{D} \right\rceil &= 1 + \left\lceil \frac{\lg(N/B) - D + 1}{D} \right\rceil \\ &= \left\lceil \frac{\lg(N/B) + 1}{D} \right\rceil . \end{aligned}$$

3.5 General matrix transpose

We can transpose any matrix efficiently, even when the number of elements is not a power of 2. This section presents a method that uses less than $9\,\frac{RS}{BD}\left\lceil\frac{\lg\min(R,S,B,RS/B)}{\lg(M/B)}\right\rceil + \frac{53}{2}\,\frac{RS}{BD} + 11$ parallel I/Os to transpose any $R \times S$ matrix, and it often uses fewer. Detection of general matrix transpose is also easy, requiring only $\lceil N/BD \rceil$ parallel reads, where $N = RS$.

Performing general matrix transpose

We perform a general matrix transpose in three steps:

1. Partition the given matrix X into four submatrices, each with dimensions that are powers of 2.

2. Use the BPC algorithm to transpose each submatrix.

3. Reassemble the transposed submatrices to form X^{T}.

We determine the submatrix dimensions as follows. Let X have R rows and S columns, and define $R' = 2^{\lfloor \lg R \rfloor}$ and $S' = 2^{\lfloor \lg S \rfloor}$. The value R' is the closest power of 2 less than R unless R itself is a power of 2, in which case $R' = R$. The same holds for S' and S. Note that $R/2 < R' \le R$ and $S/2 < S' \le S$, which imply that $R - R' < R/2$ and $S - S' < S/2$. We partition X into four submatrices with the following dimensions:

$$X = \begin{array}{cc|cc} & S' & S-S' & \\ \left[\right. & X_{11} & X_{12} & \left.\right] R' \\ \hline & X_{21} & X_{22} & R-R' \end{array} .$$

The transpose of X is then

$$X^{\mathrm{T}} = \begin{array}{cc|cc} & R' & R-R' & \\ \left[\right. & X_{11}^{\mathrm{T}} & X_{21}^{\mathrm{T}} & \left.\right] S' \\ \hline & X_{12}^{\mathrm{T}} & X_{22}^{\mathrm{T}} & S-S' \end{array} .$$

We can use the BPC algorithm to compute X_{11}^{T} because the dimensions $R' \times S'$ of X_{11} are powers of 2. To compute X_{12}^{T}, X_{21}^{T}, and X_{22}^{T} using the BPC algorithm, however, we need to pad the dimensions of X_{12}, X_{21}, and X_{22} to powers of 2. That is, we define the following matrices:

- $\widehat{X}_{12}$ is an $R' \times S/2$ matrix whose leading $R' \times (S - S')$ submatrix is equal to X_{12}.
- $\widehat{X}_{21}$ is an $R/2 \times S'$ matrix whose leading $(R - R') \times S'$ submatrix is equal to X_{21}.
- $\widehat{X}_{22}$ is an $R/2 \times S/2$ matrix whose leading $(R - R') \times (S - S')$ submatrix is equal to X_{22}.

We then run the BPC algorithm on $\widehat{X}_{12}$, $\widehat{X}_{21}$, and $\widehat{X}_{22}$ to compute their transposes. After computing $\widehat{X}_{12}^{\mathrm{T}}$, $\widehat{X}_{21}^{\mathrm{T}}$, and $\widehat{X}_{22}^{\mathrm{T}}$, we set

- X_{12}^{T} to the leading $(S - S') \times R'$ submatrix of $\widehat{X}_{12}^{\mathrm{T}}$,
- X_{21}^{T} to the leading $S' \times (R - R')$ submatrix of $\widehat{X}_{21}^{\mathrm{T}}$, and
- X_{22}^{T} to the leading $(S - S') \times (R - R')$ submatrix of $\widehat{X}_{22}^{\mathrm{T}}$.

We create the matrices X_{11}, $\widehat{X}_{12}$, $\widehat{X}_{21}$, and $\widehat{X}_{22}$ using a particularly efficient version of a 4-way split. As we saw in Section 3.3, we can perform a k-way split in at most $\lceil N_s/BD \rceil + 2\lceil N_t/BD \rceil + 2(k-1)$ parallel I/Os. Here we have $k = 4$, $N_s = RS$, and

$$\begin{aligned} N_t &= R'S' + R'(S - S') + (R - R')S' + (R - R')(S - S') \\ &< RS + RS/2 + RS/2 + RS/4 \end{aligned}$$

$$= \frac{9}{4} RS \; .$$

We can reduce the constants in the 4-way split in two ways. First, there is no need to read tracks of the matrices we create, since we don't care about the values in their pad regions. This observation allows to us to reduce the term $2\lceil N_t/BD \rceil$ to only $\lceil N_t/BD \rceil$. The second way is to start each of the four matrices on a track boundary, which we need to do anyway. The $2(k-1)$ term comes only from target tracks that share partitions in a k-way split. Since there are no such tracks, this term drops out altogether. The total number of parallel I/Os to create the four matrices is thus at most

$$\begin{aligned}\left\lceil \frac{N_s}{BD} \right\rceil + \left\lceil \frac{N_t}{BD} \right\rceil &\le \left\lceil \frac{RS}{BD} \right\rceil + \left\lceil \frac{\frac{9}{4} RS}{BD} \right\rceil \\ &< \frac{13}{4} \frac{RS}{BD} + 2 \; .\end{aligned}$$

We then transpose the four matrices X_{11}, $\widehat{X}_{12}$, $\widehat{X}_{21}$, and $\widehat{X}_{22}$ using the BPC algorithm. We'll analyze the I/O requirements for these transpose operations in a moment.

Having computed X_{11}^{T}, $\widehat{X}_{12}^{\mathrm{T}}$, $\widehat{X}_{21}^{\mathrm{T}}$, and $\widehat{X}_{22}^{\mathrm{T}}$, we need to reassemble them into the matrix X^{T}. The reassembly is simply a 4-monotonic route with the source and target vectors reversing their roles from the partitioning. Thus, we have $k = 4$, $N_s \le \frac{9}{4} RS$, and $N_t = RS$, so the I/O count for reassembly is at most

$$\begin{aligned}\left\lceil \frac{N_s}{BD} \right\rceil + 2k \left\lceil \frac{N_t}{BD} \right\rceil &= \left\lceil \frac{\frac{9}{4} RS}{BD} \right\rceil + 8 \left\lceil \frac{RS}{BD} \right\rceil \\ &< \frac{41}{4} \frac{RS}{BD} + 9 \; .\end{aligned}$$

Note that this method is *not* recursive; we only partition and reassemble once.

It remains for us to compute the I/O cost of performing the four submatrix transpose operations. Before doing so, we present a more concise characterization of the cross-rank for matrix transpose.

The cross-rank of matrix transpose

It turns out that the cross-rank of a matrix transpose has a simple characterization when we normalize it by $\lg(M/B)$. We can use this characterization to give a tight bound on the number of parallel I/Os for the transpose of any matrix. We start by proving a useful fact about cross-ranks. Since cross-ranks apply only to BPC permutations, which require the number of elements to be a power of 2, we assume for now that the matrix dimensions are powers of 2.

Lemma 3.4 *For any matrix A that characterizes a BPC permutation, $\rho_{\lg B}(A) - \rho_{\lg M}(A) \le \lg(M/B)$ and $\rho_{\lg M}(A) - \rho_{\lg B}(A) \le \lg(M/B)$.*

Proof: Because A characterizes a BPC permutation, it is a permutation matrix. The rank of any $R \times S$ submatrix of a permutation matrix is the number of 1s in the submatrix, and it is no greater than $\min(R, S)$. Thus,

$$\begin{aligned} \rho_{\lg B}(A) &= \operatorname{rank}(A_{0..\lg B-1, \lg B..\lg N-1}) \\ &= \operatorname{rank}(A_{0..\lg B-1, \lg B..\lg M-1}) + \operatorname{rank}(A_{0..\lg B-1, \lg M..\lg N-1}) \end{aligned}$$

and

$$\begin{aligned} \rho_{\lg M}(A) &= \operatorname{rank}(A_{0..\lg M-1, \lg M..\lg N-1}) \\ &= \operatorname{rank}(A_{0..\lg B-1, \lg M..\lg N-1}) + \operatorname{rank}(A_{\lg B..\lg M-1, \lg M..\lg N-1}) \ , \end{aligned}$$

and so

$$\begin{aligned} \rho_{\lg B}(A) - \rho_{\lg M}(A) &= [\operatorname{rank}(A_{0..\lg B-1, \lg B..\lg M-1}) + \operatorname{rank}(A_{0..\lg B-1, \lg M..\lg N-1})] \\ &\qquad - [\operatorname{rank}(A_{0..\lg B-1, \lg M..\lg N-1}) + \operatorname{rank}(A_{\lg B..\lg M-1, \lg M..\lg N-1})] \\ &= \operatorname{rank}(A_{0..\lg B-1, \lg B..\lg M-1}) - \operatorname{rank}(A_{\lg B..\lg M-1, \lg M..\lg N-1}) \\ &\le \lg(M/B) - 0 \end{aligned}$$

and

$$\begin{aligned}\rho_{\lg M}(A) - \rho_{\lg B}(A) &= \operatorname{rank}(A_{\lg B..\lg M-1,\lg M..\lg N-1}) - \operatorname{rank}(A_{0..\lg B-1,\lg B..\lg M-1}) \\ &\leq \lg(M/B) - 0\ ,\end{aligned}$$

which completes the proof. ■

Next, we show that the $(\lg B)$-cross-rank of a matrix transpose is concisely characterized.

Lemma 3.5 *Let A be a $(\lg N) \times (\lg N)$ permutation matrix characterizing a BPC permutation that is the transpose of an $R \times S$ matrix. Then $\rho_{\lg B}(A) = \lg \min(R, S, B, N/B)$.*

Proof: We use the address-bit permutation formulation of matrix transpose in equation (2.2). We start by showing that $\rho_{\lg B}(\pi) = \lg \min(R, S, B, N/B)$. We consider each of the four cases separately.

1. $\min(R, S, B, N/B) = R$:
   As Figure 3.3(a) shows, $\lg R$ address bits cross from the upper side of position $\lg B$ to the lower side. The remaining $\lg(B/R)$ bits that end up on the lower side of position $\lg B$ start on the lower side. We have $\rho_{\lg B}(\pi) = \lg R$.

2. $\min(R, S, B, N/B) = S$:
   As Figure 3.3(b) shows, $\lg S$ address bits cross from the lower side of position $\lg B$ to the upper side. The remaining $\lg(N/BS)$ bits that end up on the upper side of position $\lg B$ start on the upper side. We have $\rho_{\lg B}(\pi) = \lg S$.

3. $\min(R, S, B, N/B) = B$:
   As Figure 3.3(c) shows, all of the lower $\lg B$ bits of the column number cross position $\lg B$. We have $\rho_{\lg B}(\pi) = \lg B$.

4. $\min(R, S, B, N/B) = N/B$:
   As Figure 3.3(d) shows, all of the upper $\lg(N/B)$ bits of the row number cross position $\lg B$. We have $\rho_{\lg B}(\pi) = \lg(N/B)$.

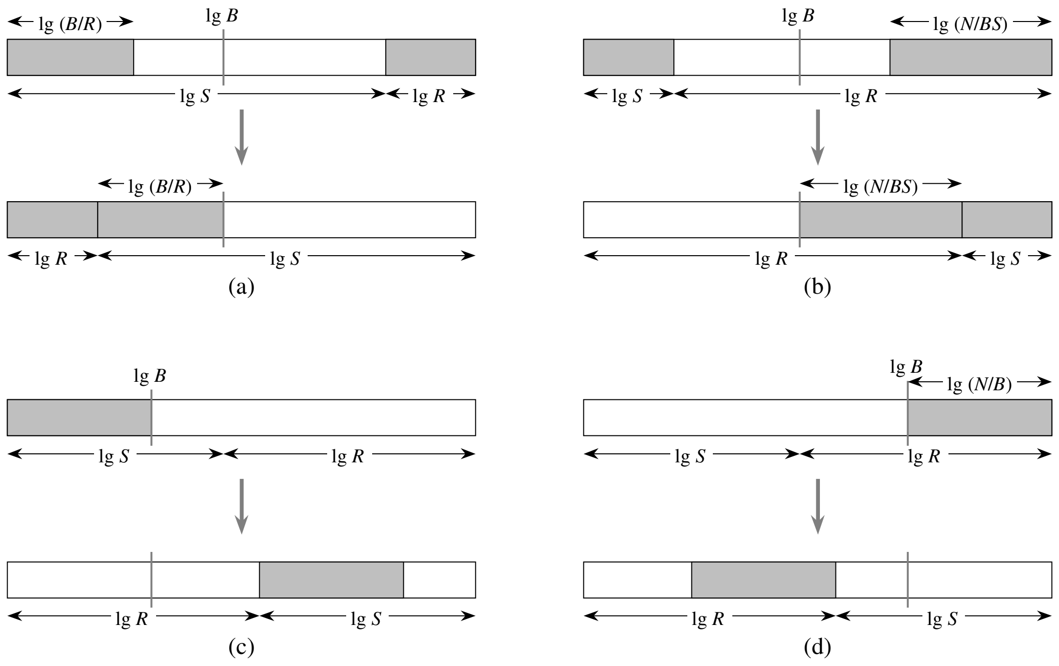


Figure 3.3: Cases in the proof of Lemma 3.5. Lower-order bit positions are on the left. **(a)** When $\min(R, S, B, N/B) = R$, the $\lg R$ bits that end up on the left must come from the right of position $\lg B$, and the remaining leftmost $\lg(B/R)$ bits start on the left of position $\lg B$. **(b)** When $\min(R, S, B, N/B) = S$, the $\lg S$ bits that end up on the right must come from the left of position $\lg B$, and the remaining rightmost $\lg(N/BS)$ bits start on the right of position $\lg B$. **(c)** When $\min(R, S, B, N/B) = B$, all of the leftmost $\lg B$ bits cross position $\lg B$. **(d)** When $\min(R, S, B, N/B) = N/B$, all of the rightmost $\lg(N/B)$ bits cross position $\lg B$.

When the address-bit permutation π and the characteristic matrix A are related by equations (2.1) (page 29) and (2.3) (page 31), we have that $\rho_k(\pi) = \rho_k(A)$ for all k, which completes the proof. ■

We are now ready to give a simple characterization of the cross-rank for matrix transpose. This form matches the bound for matrix transpose given by Vitter and Shriver [VS90a, VS90b].

Theorem 3.6 *We can transpose any $R \times S$ matrix, where R and S are powers of 2, using between $\frac{2N}{BD}\left(2\left\lceil\frac{\lg\min(R,S,B,N/B)}{\lg(M/B)}\right\rceil + 1\right)$ and $\frac{2N}{BD}\left(2\left\lceil\frac{\lg\min(R,S,B,N/B)}{\lg(M/B)}\right\rceil + 3\right)$ parallel I/Os.*

Proof: Let A be the $(\lg N) \times (\lg N)$ characteristic matrix for the matrix transpose, where $N = RS$. By equation (2.6) (page 38), $\rho(A)$ equals either $\rho_{\lg B}(A)$ or $\rho_{\lg M}(A)$. In the former case, we apply Lemma 3.5 to the bound of $\frac{2N}{BD}\left(2\left\lceil\frac{\rho(A)}{\lg(M/B)}\right\rceil+1\right)$ parallel I/Os for BPC permutations from Theorem 2.8 (page 44) to prove the theorem. In the latter case, we apply Lemmas 3.4 and 3.5:

$$\begin{aligned}\frac{\rho(A)}{\lg(M/B)} &\leq \frac{\rho_{\lg B}(A)+\lg(M/B)}{\lg(M/B)} \\ &= \frac{\lg\min(R,S,B,N/B)}{\lg(M/B)}+1 \ . \end{aligned} \tag{3.5}$$

Plugging inequality (3.5) into the bound from Theorem 2.8, we get that the number of parallel I/Os is at most

$$\frac{2N}{BD}\left(2\left\lceil\frac{\lg\min(R,S,B,N/B)}{\lg(M/B)}+1\right\rceil+1\right)=\frac{2N}{BD}\left(2\left\lceil\frac{\lg\min(R,S,B,N/B)}{\lg(M/B)}\right\rceil+3\right) \ ,$$

which completes the proof. ■

Total I/O count for general matrix transpose

We are now ready to total up the number of parallel I/Os for general matrix transpose.

Theorem 3.7 *We can transpose any $R \times S$ matrix, for any dimensions R and S, using less than*

$$9\,\frac{RS}{BD}\left\lceil\frac{\lg\min(R,S,B,RS/B)}{\lg(M/B)}\right\rceil+\frac{53}{2}\,\frac{RS}{BD}+11$$

parallel I/Os.

Proof: We counted the parallel I/Os for partitioning and reassembly above, so we only need to count the I/Os for the four submatrix transpose operations and then sum all the I/O counts together:

1. The matrix X_{11} has dimensions $R' \times S'$, so by Theorem 3.6, computing X_{11}^{T} requires at most

$$\frac{2R'S'}{BD}\left(2\left\lceil\frac{\lg\min(R',S',B,R'S'/B)}{\lg(M/B)}\right\rceil+3\right)\leq\frac{2RS}{BD}\left(2\left\lceil\frac{\lg\min(R,S,B,RS/B)}{\lg(M/B)}\right\rceil+3\right)$$

parallel I/Os.

2. The matrix $\widehat{X}_{12}$ has dimensions $R' \times (S-S')$, so by Theorem 3.6, computing $\widehat{X}_{12}^{\mathrm{T}}$ requires at most

$$\begin{aligned} \frac{2R'(S-S')}{BD}\left(2\left\lceil\frac{\lg\min(R', S-S', B, R'(S-S')/B)}{\lg(M/B)}\right\rceil+3\right) \\ < \frac{RS}{BD}\left(2\left\lceil\frac{\lg\min(R,S,B,RS/B)}{\lg(M/B)}\right\rceil+3\right) \end{aligned}$$

parallel I/Os.

3. The matrix $\widehat{X}_{21}$ has dimensions $(R-R') \times S'$, so by Theorem 3.6, computing $\widehat{X}_{21}^{\mathrm{T}}$ requires at most

$$\begin{aligned} \frac{2(R-R')S'}{BD}\left(2\left\lceil\frac{\lg\min(R-R', S', B, (R-R')S'/B)}{\lg(M/B)}\right\rceil+3\right) \\ < \frac{RS}{BD}\left(2\left\lceil\frac{\lg\min(R,S,B,RS/B)}{\lg(M/B)}\right\rceil+3\right) \end{aligned}$$

parallel I/Os.

4. The matrix $\widehat{X}_{22}$ has dimensions $(R-R') \times (S-S')$, so by Theorem 3.6, computing $\widehat{X}_{22}^{\mathrm{T}}$ requires at most

$$\begin{aligned} \frac{2(R-R')(S-S)'}{BD}\left(2\left\lceil\frac{\lg\min(R-R', S-S', B, (R-R')(S-S')/B)}{\lg(M/B)}\right\rceil+3\right) \\ < \frac{RS}{2BD}\left(2\left\lceil\frac{\lg\min(R,S,B,RS/B)}{\lg(M/B)}\right\rceil+3\right) \end{aligned}$$

parallel I/Os.

Now we total up all the I/Os. Totaling up the coefficients, we see that the four submatrix transpose operations require less than

$$\frac{9}{2}\,\frac{RS}{BD}\left(2\left\lceil\frac{\lg\min(R,S,B,RS/B)}{\lg(M/B)}\right\rceil+3\right)$$

parallel I/Os. Adding in the I/Os for partitioning and reassembly, we get a total count of less than

$$\left(\frac{13}{4}\frac{RS}{BD}+2\right)+\left(\frac{9}{2}\frac{RS}{BD}\left(2\left\lceil\frac{\lg\min(R,S,B,RS/B)}{\lg(M/B)}\right\rceil+3\right)\right)+\left(\frac{41}{9}\frac{RS}{BD}+9\right)$$
$$= 9\frac{RS}{BD}\left\lceil\frac{\lg\min(R,S,B,RS/B)}{\lg(M/B)}\right\rceil+\frac{53}{2}\frac{RS}{BD}+11\ ,$$

which completes the proof. ■

Although the I/O count of Theorem 3.7 has relatively large constant factors—9 and 53/2—this method is still likely to be practical. The constants are derived from taking upper bounds. For the partitioning and reassembly analyses, the tightness of some bounds depends on how close R and S are to their next lower powers of 2. We bounded $R - R'$ by $R/2$ and $S - S'$ by $S/2$. If R is just above a power of 2, then $R - R'$ is close to 0, and the same holds for S and $S - S'$. A more careful analysis would then yield lower constants. In a more extreme case, either R or S is a power of 2. Two of the submatrices would then drop out of the method altogether, and the constants would then be much lower.

Detecting general matrix transpose

General matrix transpose is much simpler to detect than BMMC permutations. We can determine from just one target address the number of columns S that the transposed N-element matrix must have. The number of rows R must then equal N/S. Given these dimensions, we can easily verify that each target address corresponds to where it belongs in a row-major ordering of the transpose.

If the target-address vector is X, the one target address we need is X_1. Why? Indexing rows and columns from 0, this element is the target address of matrix entry $(0,1)$. Thus it is the row-major mapping of entry $(1,0)$ in the transpose. If there are S columns in the transpose, entry $(1,0)$ maps to S.

Thus, we can detect a general matrix transpose using only $\lceil N/BD \rceil$ parallel reads. In the first read, we determine candidate values of R and S and verify that all the target addresses of the track read are consistent with the transpose of an $R \times S$ matrix. The remaining $\lceil N/BD \rceil - 1$ parallel reads are used to verify the remaining target addresses.

3.6 Invoking special permutations

There are three ways for a system to handle special permutations such as those we examined in Sections 3.3–3.5:

1. Treat them as general permutations.
2. Detect them at run time and then call special routines to perform them.
3. Specify them in the source code and have the compiled code call special routines to perform them.

In this section, we argue in favor of the third option, specifying them in the source code, whenever possible. We shall do so by examining the advantages and disadvantages of each method.

Treating special permutations as general permutations

The only advantages of treating special permutations as general permutations are that no additional source language constructs or library functions are required and that there is no overhead for detection.

The disadvantage is obvious: general permutations take longer to perform than special permutations (except for the worst cases of BPC and BMMC permutations). This holds true even if we perform general permutations by sorting target addresses using the asymptotically sorting algorithms of Nodine and Vitter [NV90, NV91, NV92] or Vitter and Shriver [VS90a, VS90b] rather than the easier-to-code external radix sort of Section 3.1.

Detecting special permutations at run time

As we saw in Sections 3.3–3.5, the overhead for detecting special permutations is typically quite low. In the worst case, for BMMC permutations, we had to spend $N/BD + \left\lceil \frac{\lg(N/B)+1}{D} \right\rceil$ parallel I/Os to generate a hypothesized BMMC permutation and then verify it. The savings due to performing the permutation with special routines more than make up for the additional parallel I/Os for detection. Another advantage of run-time detection is that, like treating special permutations as general ones, no additional source language constructs or library functions are required.

There are two disadvantages to run-time detection. First, we pay an overhead penalty when the permutation we are given turns out to be none of the special ones we can detect and perform quickly. Second, although each detection scheme in Sections 3.3–3.5 is relatively inexpensive, when one adds up the costs of detecting all the different special permutations, the total cost might be substantial. We can defray up to $\lceil N/BD \rceil$ parallel I/Os of the total cost, however, by performing the census count of external radix sort at the same time as we attempt to detect special permutations.

Specifying special permutations in the source code

There is only one reason not to specify special permutations in the source code: it requires additional language constructs or library functions. There are, however, four compelling arguments in favor of source-level specification. We save the best one for last, because we back it up with a case study.

First, source-level specification often allows a very compact representation of the permutation. We represent a general permutation on N elements by $N \lg N$ bits: N target addresses, each of which is $\lg N$ bits long. Mesh and torus permutations in d dimensions require only d integers to specify the offsets, one offset per dimension, for a total of $d \lg N$ bits. Matrix transpose requires only two $(\lg N)$-bit integers. BPC and BMMC permutations require only $\lg^2 N + \lg N$ bits to specify the characteristic matrix and complement vector. For about one billion records ($N = 2^{30}$) this amounts to only 930 bits compared to about 30 billion bits for general permutations. For about one trillion records ($N = 2^{40}$) the comparison is 1640 bits vs. about 40 trillion bits.

Second, when the representation is compact enough that it fits in the front-end machine, we can get higher RAM utilization in the parallel machine. Consider, for example, BPC permutations with data elements that are 4-byte integers. If we treat them as general permutations and sort them based on their 4-byte target addresses, at least half of the RAM is filled by target addresses rather than the data that we wish to move. On the other hand, the BPC algorithm of Chapter 2 uses RAM only for data. In essence, the RAM size is at least twice that available to a general-permutation routine.

Third, source-level specification often matches what the programmer has in mind. Only the most oblivious programmer would generate the target addresses for a mesh permutation without realizing that he wanted to perform a special type of permutation. The same goes for torus permutations and matrix transpose. It even applies to many BPC permutations, such as bit-reversal, vector-reversal, and hypercube permutations.

Fourth, the overhead of generating all N target addresses is often avoided by source-level specification. Generating the target addresses can take much longer than performing the permutation. This can be true even when we use the general permutation routine to perform it! The following case study shows that generating the target addresses for BPC permutations can take $\Theta\left(\frac{N}{BD} \lg N\right)$ parallel I/Os, which is significantly worse than even the general permutation I/O cost of $\Theta\left(\frac{N}{BD} \frac{\lg N}{\lg(M/BD)}\right)$.

A case study: BPC permutations in VM-DP

In remainder of this section, we report empirical data from the VM-DP system on performing BPC permutations, drawing the following conclusions:

1. We can predict the performance of external radix sort and of the BPC algorithm of Chapter 2 very accurately.

2. If we are not clever in how we generate the target addresses, the cost of doing so is exorbitant. If we are clever, the cost is merely high. Target-address generation can be

expensive and should be avoided.

This case study is small but conclusive. It is small because the simulations performed by the VM-DP system are limited by the address-space restrictions of the workstation and by the slow speed of the simulation. It is conclusive because it strongly supports both of the above conclusions for all problem sizes examined.

The permutations have the following characteristics in common:

- Each element is a 4-byte integer.
- Each block is 128 bytes and thus can hold 32 4-byte integers.
- The RAM is 16384 bytes and thus can hold 4192 4-byte integers. We then have $\lg(M/B) = \lg(4192/32) = 7$.
- There are $D = 4$ disks and $P = 16$ processors.
- Each permutation is the transpose of an $R \times 16384$ matrix, where the number of rows R varies but is always a power of 2.

We examine two ways to perform the transpose permutations, specified by a run-time flag that is checked by the VM-DP system. It can treat them as general permuations, or it can attempt to detect BPC permutations and perform them specially. As explained in Sections 2.9 and 4.6, when it performs a BPC permutation specially, the VM-DP system actually performs a slightly different BPC permutation than it is given. This discrepancy is because integer vectors are laid out within tracks in a slightly different fashion than the algorithms of Chapter 2 expect. The difference between the permutation the system is given and the one it performs is so small that their cross-ranks differ by at most two.

We also examine two ways to generate the target addresses. The less clever method alluded to above generates them one bit at a time. That is, from equations (2.1) and (2.2) (page 31), we set bit target-address bit $y_{(j+\lg R) \bmod N}$ to source-address bit x_j for $j = 0$, then for $j = 1$, and so on up to $j = \lg N - 1$. Each bit position requires making one pass over the source-address

N	R	observed BPC I/Os	observed radix sort I/Os
32768	2	2317	6408
65536	4	4621	14856
131072	8	9230	29704
262144	16	18446	59400
524288	32	36878	135176
1048576	64	73742	270344
2097152	128	212999	540680
4194304	256	425991	1212424

Table 3.1: Observed I/O counts for the BPC algorithm and external radix sort.

and target-address vectors, for a total of $\Theta\left(\frac{N}{BD}\lg N\right)$ parallel I/Os. As N increases, this cost becomes exorbitant.

The more clever method of generating target addresses generates several bits at a time by using arithmetic on the number of rows and columns. A source address for row i and column j has the value $x = iS + j$ in row-major ordering, and its corresponding target address has the row-major value $y = jS + i$. Since $i = \lfloor x/S \rfloor$ and $j = x \bmod S$, we can compute the target addresses in a constant number of passes, or $\Theta(N/BD)$ parallel I/Os, by the following sequence of vector operations:

$$\begin{aligned} i &\leftarrow \lfloor x/S \rfloor \\ j &\leftarrow x \bmod S \\ y &\leftarrow j * S \\ y &\leftarrow y + j\ . \end{aligned}$$

We can predict the I/O counts of the BPC algorithm quite accurately. Because of aliasing within the VCODE interpreter, the VM-DP system always works with a copy of the vector. The predicted number of parallel reads is

- N/BD to make the copy,

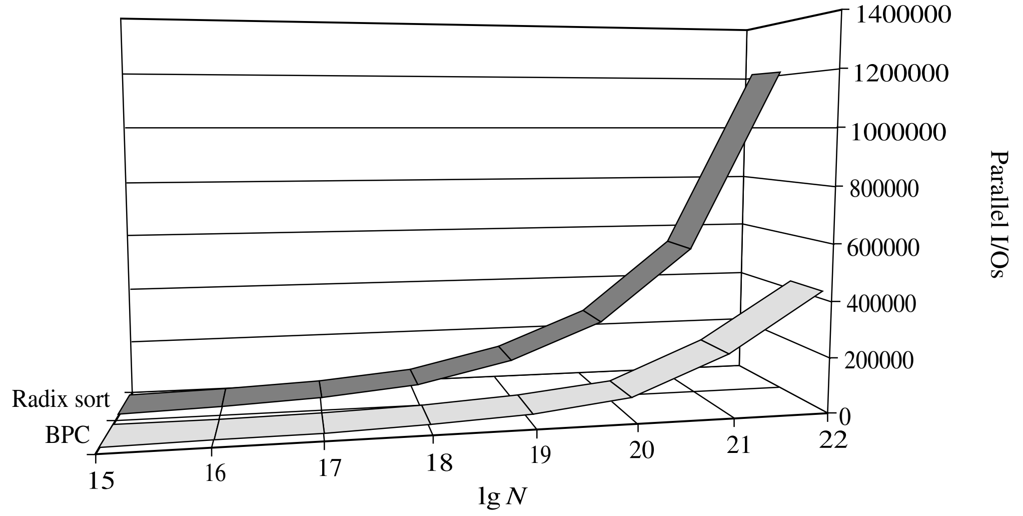


Figure 3.4: A plot of Table 3.1.

- $\left\lceil \frac{\lg(N/B)+1}{D} \right\rceil$ to form the candidate characteristic matrix and complement vector,
- N/BD to verify them, and
- $\left(2\left\lceil \frac{\rho}{\lg(M/B)} \right\rceil + 1\right)\frac{N}{BD}$ for all passes of the algorithm.

The predicted number of parallel writes is N/BD to make the copy and $\left(2\left\lceil \frac{\rho}{\lg(M/B)} \right\rceil + 1\right)\frac{N}{BD}$ for each pass of the algorithm. Here, $\lg(M/B) = 7$. The predicted read count matches the observed read count exactly. The predicted and observed write counts differ by at most 10, this difference being due to writes of changed tracks when RAM is cleared out prior to running the BPC algorithm.

We can predict the I/O counts of external radix sort with equal accuracy. Because the target addresses and records are separate vectors, we perform twice as many I/Os per pass. Because of aliasing within the VCODE interpreter, the system works with a copy of the target-address vector and of the vector containing the records to be permuted. The predicted number of parallel reads is

- $2N/BD$ to make the copies,

N	R	tracks	other reads, fast method	other reads/track, fast method	other reads, slow method	other reads/track/lg N, slow method
32768	2	256	4873	19.035	60232	15.685
65536	4	512	9737	19.018	128076	15.634
131072	8	1024	19465	19.009	271441	15.593
262144	16	2048	38921	19.004	573525	15.558
524288	32	4096	77833	19.002	1208409	15.527
1048576	64	8192	155657	19.001	2539614	15.501
2097152	128	16384	311305	19.001		
4194304	256	32768	622601	19.000		

Table 3.2: Read counts excluding the actual permuting for the faster and slower target-address generation methods. The faster method is proportional to N/BD, and the slower method is proportional to $(N/BD)\lg N$. There are no results for the slower-method simulation with $N \geq 2097152$ because the simulation took so long that it was killed.

- N/BD to form the census counts for buckets,
- $2\left\lceil \frac{\lg N}{\lg(M/4BD)} \right\rceil \frac{N}{BD}$ for all passes, and
- N/BD during compaction (only the records are compacted, not the keys).

The predicted number of parallel writes is the same, minus the N/BD for forming the census counts. The total number of parallel I/Os is thus $\left(4\left\lceil \frac{\lg N}{\lg(M/4BD)} \right\rceil + 7\right)\frac{N}{BD}$. Again, the predicted and observed read counts match exactly, and the write counts differ by at most 8.

As Table 3.1 and Figure 3.4 show, external radix sort uses many more I/Os than the specialized BPC algorithm. For these examples, external radix sort uses between 2.5 and 4 times as many parallel I/Os. Some of these additional I/Os are due to external radix sort having to read and write the target addresses, and the remainder are due to the BPC algorithm being more asymptotically efficient.

Table 3.2 and Figure 3.5 show the number of parallel reads performed other than in actually performing the permutation. Most of these reads occur during generation of the target addresses. Some of them, however, are due to system overhead and appear to be unavoidable. The parallel-write counts are comparable. Observe that with the faster target-address-generation method, the number of reads exceeds the number used to perform even external radix sort. The

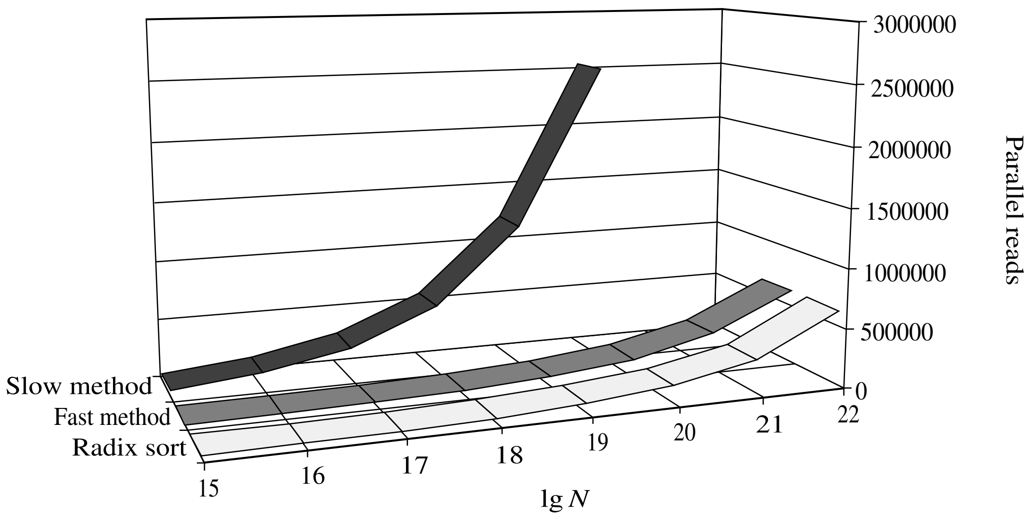


Figure 3.5: A plot of Table 3.2. Even with the faster target-address-generation method, the read counts slightly exceed those incurred during external radix sort.

table shows that the ratio of the number of reads to the number of tracks (which is directly proportional to N with B and D fixed in this example) is just over 19. Thus the number of reads grows linearly in N. With the slower method, we see that the number of reads per track, divided by $\lg N$, is just over 15.5; it grows as $(N/BD) \lg N$. Here, the cost of generating the target addresses dwarfs all other costs, including the cost of permuting.

Recap

This case study provides strong evidence that generating target addresses, even when done cleverly, can be very costly.

Why should the programmer go through the effort of writing code to generate target addresses just so the run-time system can go through the effort of detecting special permutations, when a compact, direct method is available? Source-level specification is clearly the method of choice.

3.7 Conclusions

In this section, we present some conclusions and discuss future work.

The principal lesson of this chapter is that in data-parallel systems with virtual memory, permuting is so expensive that any special permutations that can be performed faster than general permutations should be specified at the source level. Source-level specification allows for direct invocation of the special routines to perform the permutation, permits a more compact representation of the permutation and hence greater utilization of the available RAM, and avoids the high expense of target-address generation.

We saw how to perform and detect several such permutations: monotonic routes, mesh and torus permutations, BMMC and BPC permutations, and general matrix transpose. Unfortunately, NESL, which is the source language for the VM-DP system, has no way to invoke special routines for permutations, with one exception. That exception is the `pack` function, a type of monotonic route in which every position of the target vector receives an element. This function, useful for load-balancing, has its own VCODE instruction and CVL functions, which is why we can invoke it specially. There is no VCODE instruction or CVL entry point for other special permutations. Modification of NESL and VCODE to permit special library functions is highly desirable from the point of view of the VM-DP system.

VM-DP includes run-time detection of BPC permutations, and it performs them with special routines. We have yet to add code to detect and specially perform mesh, torus, BMMC, and general matrix-transpose permutations. (The BPC detection code only needs a nonsingularity check added to be BMMC detection code.) VM-DP includes a low-level monotonic route function, which is invoked not only by the `pack` function, but as a subroutine by other CVL functions.

There are two other directions for future work. One is to implement other sorting algorithms to perform general permutations. Obvious candidates are the most recent one by Nodine and Vitter [NV92] and the Vitter-Shriver randomized algorithm [VS90a, VS90b]. Another interesting candidate would be an external version of sample sort [BLM$^+$91], such as Smith's implementation [Smi92].

The other direction for future work is to extend the catalog of special permutation classes that we can perform quickly. Ideally, we should be able to detect such special permutations efficiently as well, but if they are specified at the source level, detection becomes moot.

Chapter 4

Vector Layout

In a data-parallel computer with virtual memory, the way in which vectors are laid out on the disk system affects the performance of data-parallel operations. This chapter presents a general method of vector layout called banded layout, in which we divide a vector into bands of a number of consecutive vector elements laid out in column-major order. This chapter analyzes the effect of band size β on upper bounds for the major classes of data-parallel operations, deriving the following results:

- For permuting operations, the best upper bounds come from band sizes that are a track or smaller. There is no optimal band size for random permutations or BPC permutations. Regardless of the number of grid dimensions, we can perform mesh and torus permutations efficiently when all dimensions are powers of 2. Specifically, these permutations require at most $5N/BD$ parallel I/Os when the band size is at most the track size BD, and they require at most $9N/BD$ parallel I/Os otherwise. The smaller upper bound for $\beta \leq BD$ suggests that we should use small band sizes for permuting.

- For scan operations, the best upper bounds occur when the band size equals the size of the I/O buffer, and these sizes depend on several machine parameters. In particular, the best band size is a power of 2 that is near $\sqrt{\frac{NBD}{\alpha}\frac{S}{IO}}$ and is between BD and M, where S is the time to perform a physical scan operation, IO is the time to perform a parallel I/O operation, and α is the fraction of tracks in the I/O buffer that are in the vector being scanned.

- The band size has no effect on the performance of reduce operations.

- When there are several different record sizes, band sizes for elementwise operations should be based on the largest record size, which has the smallest band size.

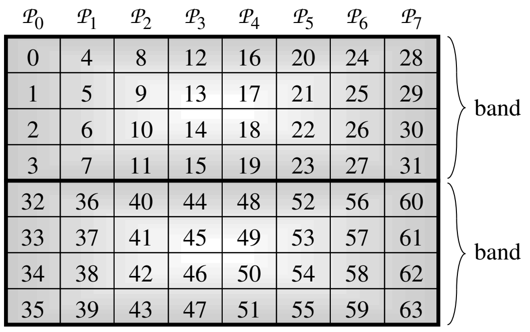
This chapter focuses on vectors that occupy several tracks. An N-record vector starts at the beginning of a track and occupies exactly $\lceil N/BD \rceil$ consecutive tracks on the disk array. For simplicity in exposition and analysis we shall assume that N is an integer multiple of BD and hence of P.

In this chapter, we relax the assumption of the machine model from Section 1.3 that vectors are laid out a track at a time. As we shall see, when the band size exceeds the size of a track, we do not have the first BD elements of a vector in track 0, the next BD in track 1, and so on. The track that a given elements maps to will depend on the band size among other parameters.

This chapter is organized as follows. Section 4.1 presents how to lay out vectors in terms of a parameterized band size. Section 4.2 summarizes the three major classes of data-parallel operations considered in Sections 4.3–4.6. Section 4.3 discusses the effect of band size on the performance of several permutation classes. Section 4.4 presents an algorithm for the scan operation on a banded vector and gives a precise formula for the performance of the operation under a certain set of assumptions. This section also briefly analyzes reduce operations. Section 4.5 analyzes the scan formula to derive the optimal band size. Section 4.6 looks at the effect of band size on elementwise operations. Finally, Section 4.7 contains some concluding remarks.

4.1 Banded vector layout

This section presents banded vector layout, which is a general framework for laying out vectors on a parallel disk system for parallel processing. A banded layout is characterized by a parameter we call the band size. After defining banded layout in general, we shall how some specific layout methods result from particular choices for the band size. Then we shall see the one-to-one mapping between a record's index in its vector and its location (track number, row within track, and processor number) in the banded layout.



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

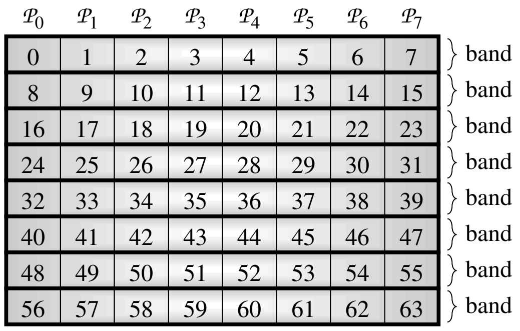
Figure 4.1: Banded layout of a vector, shown for $N = 64$ elements in the vector, $P = 8$ processors, and $\beta = 32$ elements per band. Each position shows the index of the element mapped to that position. If each band is also a track, it is a Vitter-Shriver layout as well.

Banded layout

In a *banded layout*, we divide a vector of length N into *bands* of β elements each. We restrict the *band size* β to be a power of 2 times the number of processors. Figure 4.1 shows an example of banded layout for $P = 8$ and $\beta = 32$. Each row in Figure 4.1 contains one element per processor. Element indices vary most rapidly within each processor, then among processors within a band, and they vary least rapidly from band to band. Within each band, elements are in column-major order. The mapping of elements to disk locations follows directly from this mapping of elements to processors according to the scheme of Figure 1.1. We use processor rather than disk mapping because, as we shall see, both processing and disk I/O time are criteria for comparing layout methods. The total number of rows is sometimes called the *virtual processor ratio*, and it equals N/P. The number of bands for a vector of length N is N/β. Each band contains β/P rows.

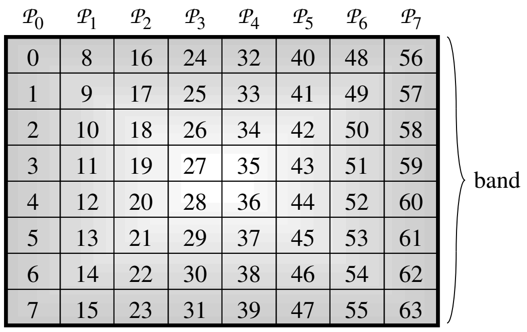
Particular banded layouts

The specific choice of the band parameter β determines the exact layout of a vector. Three particular choices yield familar vector layouts: row-major, column-major, and the Vitter-Shriver scheme used in Chapter 2. We shall see in later sections that for some operations, the band size doesn't affect the performance but for others it does. In particular, the best layout is either



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

Figure 4.2: Row-major layout of a vector, shown for $P = 8$ and $N = 64$ elements in the vector.



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

Figure 4.3: Column-major layout of a vector, shown for $P = 8$ and $N = 64$ elements in the vector.

the Vitter-Shriver one or a less common style that seems to be rarely used.

When $\beta = P$, we have *row-major layout*, shown in Figure 4.2. Each row is a band, and there are N/P bands. Element X_i is in track $\lfloor i/BD \rfloor$, processor $i \bmod P$, and row $\lfloor (i \bmod BD)/P \rfloor$ within its track. Row-major layout has two advantages. First, each track contains a contiguous subset of the vector, so that we can access the entire vector from beginning to end by reading it track by track. Second, the mapping of elements to processors depends only on the number of processors and the element index; it does not depend on any machine parameters or the vector length. As we shall see in Section 4.5, row-major order suffers from the disadvantage that it requires many physical scans during scan operations.

When $\beta = N$, we have *column-major layout*, shown in Figure 4.3. The entire vector forms one band. Element X_i is in track $\lfloor P(i \bmod (N/P))/BD \rfloor$, processor $\lfloor iP/N \rfloor$, and row $i \bmod (BD/P)$ within its track. Column-major order is a good way to lay out vectors when all data fits in RAM because, as we shall see in Section 4.5, it requires only one physical scan during a scan operation. It is a poor choice when data does not fit in RAM, because it can lead to additional I/O costs during scans.

The *Vitter-Shriver layout* uses $\beta = BD$. That is, each band is exactly one track, and there are N/BD bands. Figure 4.1 is a banded layout with a track size of $BD = \beta = 32$. Element X_i is in track $\lfloor i/BD \rfloor$, processor $\lfloor (i \bmod BD)P/BD \rfloor$, and row $i \bmod (BD/P)$. Like row-major order, each track contains a contiguous subset of the vector. In addition, each disk block does, too.

Mapping indices to banded layout locations

For a given band size β and machine parameters P, B, and D, there is a one-to-one mapping between the index of each element and a location within the banded layout, specified by a track number, a row within the track, and a processor number. We just saw examples of these mappings for row-major, column-major, and Vitter-Shriver layouts. We now present a mapping scheme that applies to any band size. Although it is interesting in its own right, we shall use this scheme in Section 4.3 to prove that we can efficiently perform a mesh or torus permutation on any vector with a banded layout regardless of the dimension of the underlying grid, and that the efficiency is better for small band sizes.

Given the $(\lg N)$-bit index i of an element, the following scheme determines the number of the element's track (between 0 and $N/BD - 1$), the number of its processor (between 0 and $P - 1$), and the number of its row within its track (between 0 and $BD/P - 1$). The scheme has two cases, depending on the relative sizes of the band size β and the track size BD.

Figure 4.4(a) shows the scheme for the case in which $\beta \leq BD$:

- The track number is given by the most significant $\lg(N/BD)$ bits, i.e., bits $\lg BD, \lg BD + 1, \ldots, \lg N - 1$. Thus, the track number of the ith element is $\lfloor i/BD \rfloor$.
- The processor number is given by the $\lg P$ bits $\lg(\beta/P), \lg(\beta/P) + 1, \ldots, \lg \beta - 1$. Thus, the processor number of the ith element is $\left\lfloor \frac{i \bmod \beta}{\beta/P} \right\rfloor = \left\lfloor \frac{(i \bmod \beta)P}{\beta} \right\rfloor$.

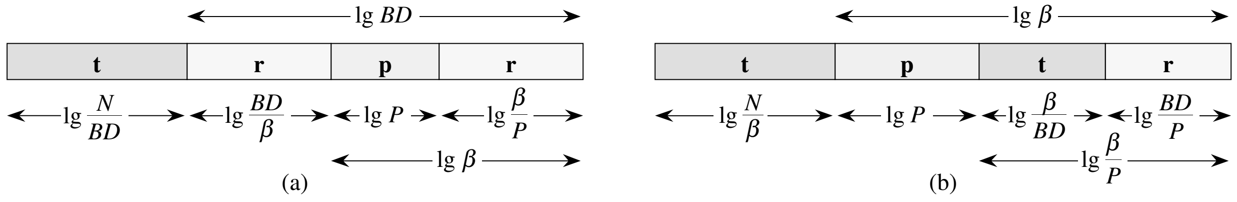


Figure 4.4: How element indices map to track numbers (bits labeled "t"), processor numbers ("p"), and row numbers within tracks ("r"). Least significant bits are on the right. **(a)** The scheme for $\beta \leq BD$. **(b)** The scheme for $\beta \geq BD$.

- The row within the track is given by the $\lg(BD/P)$ bits formed by concatenating the $\lg(\beta/P)$ bits $0, 1, \ldots, \lg(\beta/P) - 1$ and the $\lg(BD/\beta)$ bits $\lg \beta, \lg \beta + 1, \ldots, \lg BD - 1$. Thus, the row number of the ith element is $(i \bmod (\beta/P)) + \left\lfloor \frac{i \bmod BD}{\beta} \right\rfloor (\beta/P)$.

Figure 4.4(b) shows the scheme for the case in which $\beta \geq BD$:

- The track number is given by the $\lg(N/BD)$ bits formed by concatenating the $\lg(\beta/BD)$ bits $\lg(BD/P), \lg(BD/P) + 1, \ldots, \lg(\beta/P) - 1$ and the $\lg(N/\beta)$ bits $\lg \beta, \lg \beta + 1, \ldots, \lg N - 1$. Thus, the track number of the ith element is $\left\lfloor \frac{i \bmod (\beta/P)}{BD/P} \right\rfloor + \lfloor i/\beta \rfloor (\beta/BD) = \left\lfloor \frac{(i \bmod (\beta/P))P}{BD} \right\rfloor + \lfloor i/\beta \rfloor (\beta/BD)$.
- The processor number is the same as for $\beta \leq BD$. It is given by the $\lg P$ bits $\lg(\beta/P), \lg(\beta/P) + 1, \ldots, \lg \beta - 1$. Thus, the processor number of the ith element is $\left\lfloor \frac{i \bmod \beta}{\beta/P} \right\rfloor = \left\lfloor \frac{(i \bmod \beta)P}{\beta} \right\rfloor$.
- The row within the track is given by the $\lg(BD/P)$ bits $0, 1, \ldots, \lg(BD/P) - 1$. Thus, the row number of the ith element is $i \bmod (BD/P)$.

As one might expect, these two cases are equivalent for the Vitter-Shriver layout. That is, when $\beta = BD$, the least significant $\lg(BD/P)$ bits give the row within the track, the next $\lg P$ bits give the processor number, and the most significant $\lg(N/BD)$ give the track number. Moreover, we can view the least significant $\lg BD$ bits in a slightly different way. The least significant $\lg B$ bits give the offset of each element within its disk block, and the next $\lg D$ bits identify the disk containing the element. Because of this simple partition of bits among offset, disk number, and track number, the Vitter-Shriver layout is assumed by the parallel-disk algorithms of Chapters 2 and 3, Vitter and Shriver [VS90a, VS90b], and Nodine and Vitter [NV90, NV91, NV92].

4.2 Data-parallel operations

This short section presents an overview of the three major classes of data-parallel operations: elementwise operations, permuting operations, and scans. (See Blelloch [Ble90] for more on these classes of operations.) Sections 4.3–4.6 study the effect of band size on each of these operation classes.

Elementwise operations

In an *elementwise operation*, we apply a function to one or more *source vectors*, or *operands*, to compute a *target vector*, or *result*. All vectors involved are of equal length, and the value of each element of the result depends only on the corresponding elements in the operands. A simple example is elementwise addition: $Z \leftarrow X + Y$. We set Z_i to the sum $X_i + Y_i$ for each index $i = 0, 1, \ldots, N-1$. At first glance it might seem that the elementwise operations should be independent of the band size. As Section 4.6 shows, however, some elementwise operations take vectors with different record sizes. Such operations might affect the choice of band size.

Permuting operations

A *permuting operation* moves some or all of the elements from a source vector into a target vector according to a given mapping from the source-vector indices to the target-vector indices. Section 4.3 studies the effect of band size on permuting operations and concludes that band sizes less than or equal to the track size are best.

The mapping and the form in which it is specified may vary. In a *general permutation*, the mapping is a vector A of indices in the target vector. For source vector X and target vector Y, we set $Y_{A_i} \leftarrow X_i$ for all indices i such that $0 \le A_i \le N-1$.

There are many classes of *special permutations*. Chapters 2 and 3 showed that many of them can be specified more compactly and performed faster than general permutations. Section 4.3 studies three classes of special permutations: BPC permutations, mesh permutations, and torus permutations.

Scans

A *scan operation*, also known as a parallel-prefix operation, takes a source vector and yields a result vector for which each element is the "sum" of all the prior elements[1] of the source vector. Here, "sum" refers to any associative operation, which we denote by $\oplus$. Typical operations are addition, multiplication, logical-and, inclusive-or, exclusive-or, minimum, and maximum. The source and target vectors are of equal length. For example, here is a source vector X and a vector Y that is the result of scanning X with addition:

i	0	1	2	3	4	5	6	7	8	9
X_i	5	7	−3	4	−9	−2	2	0	−1	6
Y_i	0	5	12	9	13	4	2	4	4	3

Element Y_0 receives the identity value for the operation, which is 0 for addition.

Parallel machines often provide hardware [BK82, LF80] to perform scan operations with one element per processor. We call such an operation a *physical scan*.

Section 4.4 presents a method for performing scans that works well for all band sizes. Section 4.5 analyzes this method to determine the optimal band size for scans and also the optimal size of the I/O buffer used during scanning.

Related to scans are *reduce operations*, which apply an associative operator $\oplus$ to an operand vector, returning a single "sum" of all elements of the operand. Section 4.4 briefly studies reduce operations.

[1]This type of scan operation, which includes only prior elements, is often called an *exclusive* scan, as opposed to an *inclusive* scan, which includes the element itself and all prior ones.

Segmented operations

Scans, reduces, and permuting operations can treat vectors as *segmented*, so that they are understood to be smaller vectors concatenated together. Segmented operations are typically implemented by performing an equivalent unsegmented operation. (See Blelloch [Ble90] for more information on the uses and implementations of segmented vector operations.) As such, they have no effect on vector-layout issues, and we shall not consider them in this chapter.

4.3 Permuting operations and banded layout

In this section, we examine the effect of band size on permuting operations. We look at random permutations, monotonic routes, BPC permutations, and mesh and torus permutations. We shall see that random permutations and BPC permutations have no optimal band size, and that mesh and torus permutations have upper bounds of $5N/BD$ parallel I/Os when $\beta \leq BD$ and $9N/BD$ parallel I/Os when $\beta > BD$. Because of mesh and torus permutations, we prefer band sizes no greater than the track size.

Random permutations

In a random permutation on N elements, all $N!$ target orderings are equally likely. Each element of the source vector is equally likely to end up in each each position of the target vector. For a random permutation, the band size clearly does not matter.

Monotonic routes

Section 3.3 discussed monotonic routes under the machine model of Section 1.3, in which the band size is a track or less. Here we consider all band sizes.

If the band size is less than or equal to half the RAM size—$\beta \leq M/2$—we can perform a monotonic route in only one pass over the source and target vectors. We partition RAM into a "source half" and a "target half," and we read or write $M/2$ records at a time from the source and target vectors as necessary. Since no arrows cross, groups of $M/2$ records, or *half memoryloads*, are read and written in order. We read each half memoryload of the source vector once, and we read and write each half memoryload of the target vector once. (We have to read the target vector to avoid overwriting data in positions that are not routed to.) When $\beta \leq M/2$, therefore, we can perform a monotonic route with a source vector of length N_s and a target vector of length N_t using at most $N_s/BD + 2N_t/BD$ parallel I/Os.

If the band size exceeds half the RAM size, then we cannot perform monotonic routes in just one pass over the source and target vectors. We shall see later in this section that for mesh and torus permutations we prefer band sizes less than the track size, so for monotonic routes we need not concern ourselves with band sizes the size of RAM or greater. For permuting, we don't want band sizes that large.

BPC permutations

In a *bit-permute/complement*, or *BPC permutation*, we form the target index of source element X_i by permuting the bits of the $(\lg N)$-bit binary representation of the index i according to a fixed permutation $\pi : \{0, 1, \ldots, \lg N - 1\} \xrightarrow{\text{1-1}} \{0, 1, \ldots, \lg N - 1\}$. We then complement a fixed subset of the bits. Chapter 2 presents asymptotically optimal algorithms (with small constant factors) for BPC permutations on parallel disk systems.

The BPC algorithm assumes that the vector is stored in Vitter-Shriver layout, and the analysis depends on it. When the band size β is not equal to BD, Section 2.9 shows how to compute another bit permutation $\widehat{\pi} : \{0, 1, \ldots, \lg N - 1\} \xrightarrow{\text{1-1}} \{0, 1, \ldots, \lg N - 1\}$ such that the bit permutation actually performed by the BPC algorithm is $\widehat{\pi}^{-1} \circ \pi \circ \widehat{\pi}$. (The algorithm can perform the bit permutation π if the vector is laid out with the Vitter-Shriver scheme.) The bit permutation $\widehat{\pi}$ depends on β and BD.

It turns out that no one band size is always best for BPC permutations. We show this property by exhibiting a bit permutation $\widehat{\pi}$ and two bit permutations π_1 and π_2 such that the BPC algorithm uses fewer disk I/Os for $\widehat{\pi}^{-1} \circ \pi_1 \circ \widehat{\pi}$ than for π_1 but it uses more disk I/Os for $\widehat{\pi}^{-1} \circ \pi_2 \circ \widehat{\pi}$ than for π_2. For these bit permutations, we let $N = 2^{10}$, $B = 2^6$, $D = 2$, $P = 2^2$, $M = 2^8$, and $\beta = 2^3$. Applying the scheme of Section 2.9, we have the following bit permutation $\widehat{\pi}$:[2]

j	0	1	2	3	4	5	6	7	8	9
$\widehat{\pi}(j)$	0	3	4	5	6	1	2	7	8	9

The bit permutation π_1 is the bit-reversal permutation defined in Section 2.1. We then have the following bit permutations:

j	0	1	2	3	4	5	6	7	8	9
$\pi_1(j)$	9	8	7	6	5	4	3	2	1	0
$(\widehat{\pi}^{-1} \circ \pi_1 \circ \widehat{\pi})(j)$	9	4	3	2	1	8	7	6	5	0

In this example, $\rho(\pi_1) = 4$ and $\rho(\widehat{\pi}^{-1} \circ \pi_1 \circ \widehat{\pi}) = 2$. By Theorem 2.8 (page 44), the upper bound for the BPC algorithm on π_1 is 5 passes and for $\widehat{\pi}^{-1} \circ \pi_1 \circ \widehat{\pi}$ it is only 3 passes.

The bit permutation π_2 defines the transpose of a 256×4 matrix, so that we have the following bit permutations:

j	0	1	2	3	4	5	6	7	8	9
$\pi_2(j)$	8	9	0	1	2	3	4	5	6	7
$(\widehat{\pi}^{-1} \circ \pi_2 \circ \widehat{\pi})(j)$	8	5	6	1	2	9	0	3	4	7

Here, $\rho(\pi_2) = 2$ and $\rho(\widehat{\pi}^{-1} \circ \pi_2 \circ \widehat{\pi}) = 3$. By Theorem 2.8, the upper bound for the BPC algorithm on π_2 is 3 passes but for $\widehat{\pi}^{-1} \circ \pi_2 \circ \widehat{\pi}$ it is 5 passes.

Thus, we cannot show that any one band size is always best for BPC permutations.

Mesh and torus permutations

The final class of permutations we consider are mesh and torus permutations on grids each of whose dimensions are powers of 2. Section 3.3 defined these permutations. We shall show that regardless of the number of grid dimensions, we can perform mesh and torus permutations efficiently. The performance is good no matter what the band size, but it is best for band sizes

[2]The bit permutation $\widehat{\pi}$ here is the mapping $\pi_{\beta<BD}$ in equation (2.37) (page 79).

no greater than the track size, i.e., for $\beta \leq BD$.

We will show that in a mesh or torus permutation, for each track in the source vector, the elements of that track map to a small constant number of tracks in the target vector. This property is independent of the number of grid dimensions. The following lemma shows why this property is useful.

Lemma 4.1 *If the elements of each source-vector track map to at most k target tracks for a given permutation and $(k+1)BD \leq M$, then the permutation can be performed with at most $(2k+1)N/BD$ parallel I/Os.*

Proof: We perform the permutation as follows. Read into RAM a source-vector track and the k target-vector tracks it maps to, using $k+1$ parallel reads. We can do so if we have room in RAM for these $k+1$ tracks, i.e., as long as $(k+1)BD \leq M$. Move the source-vector elements into the appopriate locations in the track images of the target vector. Then write out the k target-vector tracks. We perform at most $2k+1$ parallel I/Os for this source-vector track. Repeat this process for each of the N/BD source-vector tracks, for a total of $(2k+1)N/BD$ parallel I/Os. ■

We shall assume in the remainder of this section that each grid dimension is a power of 2 and that a vector representing a grid is stored in row-major order with 0-origin indexing in each dimension. Indices vary most rapidly in dimension d and least rapidly in dimension 1. For a 3-dimensional mesh, for example, the element in grid position (p_1, p_2, p_3) appears in index $m_2 m_3 p_1 + m_3 p_2 + p_3$ of the vector. Thus, we can partition the bits of each element's index as shown in Figure 4.5: the least significant $\lg m_d$ bits give the element's position in dimension d, the next most significant $\lg m_{d-1}$ give the position in dimension $d-1$, and so on, up to the most significant $\lg m_1$ bits, which give the position in dimension 1.

Performing a mesh or torus permutation entails adding an offset o_i to the original position p_i in each dimension i. We can place each offset o_i into a $(\lg m_i)$-bit field of a $(\lg N)$-bit offset "word" that we add to each source position to compute the corresponding target position. Figure 4.5 shows that we treat the $\lg m_i$ bits of each dimension i independently.

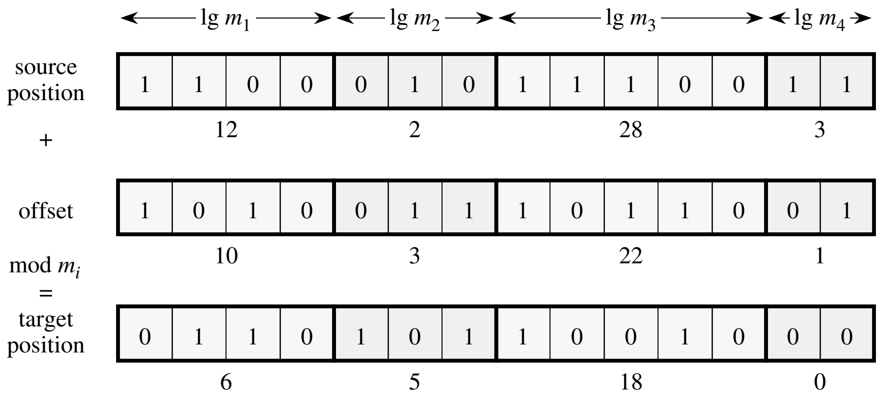


Figure 4.5: Partitioning the bits of an element's index to determine its grid position. The least significant bits are on the right. This example shows $d = 4$ dimensions, with $m_1 = 16$, $m_2 = 8$, $m_3 = 32$, and $m_4 = 4$. For each dimension i, a group of $\lg m_i$ bits determines the position in dimension i. Adding the offset $(10, 3, 22, 1)$ to the source position $(12, 2, 28, 3)$ in a torus permutation yields the target position $(6, 5, 18, 0)$. The bits of each dimension are treated independently.

We will use this way of partitioning index bits to prove that each source-vector track maps to few target-vector tracks. Before we do so, however, we need the following lemma, which will help us determine how adding an offset to a source position affects the bits corresponding to the track number in the resulting target position.

Lemma 4.2 *Let r and s be any positive integers. Let a be any integer such that $0 \le a < 2^s$, and let x' and y' be integers such that $0 \le x' \le y' < 2^r$. Define $x = a2^r + x'$ and $y = a2^r + y'$. Consider any integer c such that $0 \le c < 2^{r+s}$. Then either*

$$\begin{aligned} \left\lfloor \frac{x+c}{2^r} \right\rfloor \bmod 2^s &= \left\lfloor \frac{y+c}{2^r} \right\rfloor \bmod 2^s \ , \textit{ or} \\ \left\lfloor \frac{x+c}{2^r} \right\rfloor \bmod 2^s &= \left(\left\lfloor \frac{y+c}{2^r} \right\rfloor - 1 \right) \bmod 2^s \ . \end{aligned}$$

Proof: Let $c = e2^r + c'$, where $0 \le e < 2^s$ and $0 \le c' < 2^r$. Then

$$\begin{aligned} \left\lfloor \frac{x+c}{2^r} \right\rfloor \bmod 2^s &= \left\lfloor \frac{(a2^r + x') + (e2^r + c')}{2^r} \right\rfloor \bmod 2^s \\ &= \left(a + e + \left\lfloor \frac{x' + c'}{2^r} \right\rfloor \right) \bmod 2^s \end{aligned}$$

and

$$\begin{aligned}\left\lfloor\frac{y+c}{2^r}\right\rfloor \bmod 2^s &= \left\lfloor\frac{(a2^r+y')+(e2^r+c')}{2^r}\right\rfloor \bmod 2^s \\ &= \left(a+e+\left\lfloor\frac{y'+c'}{2^r}\right\rfloor\right) \bmod 2^s\ .\end{aligned}$$

Because $0 \le x' \le y' < 2^r$, we have that $0 \le y'-x' < 2^r$, which in turn implies that either

$$\left\lfloor\frac{x'+c'}{2^r}\right\rfloor = \left\lfloor\frac{y'+c'}{2^r}\right\rfloor \quad \text{or} \quad \left\lfloor\frac{x'+c'}{2^r}\right\rfloor = \left\lfloor\frac{y'+c'}{2^r}\right\rfloor - 1\ .$$

Thus, we have either

$$\begin{aligned}\left\lfloor\frac{x+c}{2^r}\right\rfloor \bmod 2^s &= \left(a+e+\left\lfloor\frac{x'+c'}{2^r}\right\rfloor\right) \bmod 2^s \\ &= \left(a+e+\left\lfloor\frac{y'+c'}{2^r}\right\rfloor\right) \bmod 2^s \\ &= \left\lfloor\frac{y+c}{2^r}\right\rfloor \bmod 2^s\end{aligned}$$

or

$$\begin{aligned}\left\lfloor\frac{x+c}{2^r}\right\rfloor \bmod 2^s &= \left(a+e+\left\lfloor\frac{x'+c'}{2^r}\right\rfloor\right) \bmod 2^s \\ &= \left(a+e+\left\lfloor\frac{y'+c'}{2^r}\right\rfloor-1\right) \bmod 2^s \\ &= \left(\left\lfloor\frac{y+c}{2^r}\right\rfloor-1\right) \bmod 2^s\ ,\end{aligned}$$

which completes the proof. ■

Lemma 4.2 has the following interpretation. We treat x and y as $(r+s)$-bit integers whose most significant s bits are equal and whose least significant r bits may be unequal. Without loss of generality, we assume that $x \le y$. We add to both x and y the $(r+s)$-bit integer c, and we examine the most significant s bits of the results, viewed as s-bit integers. Then either these values are equal for x and y, or the value for y is one greater than the value for x.

We are now ready to prove that each source-vector track maps to few target-vector tracks.

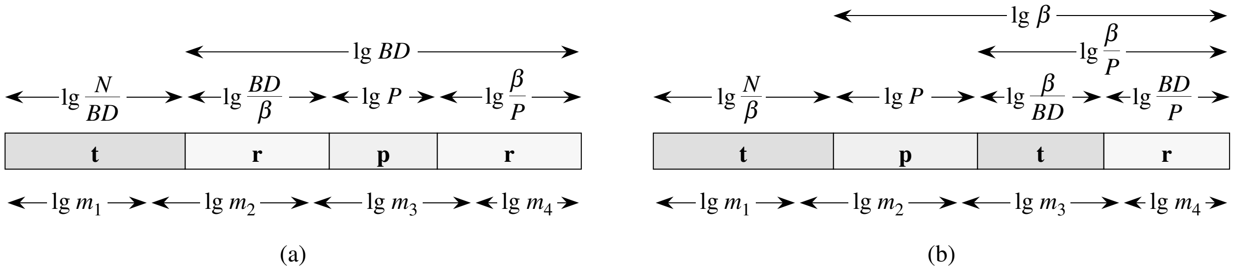


Figure 4.6: Cases in the proof of Lemma 4.3. **(a)** When $\beta \leq BD$, the track number is in bits $\lg(N/BD)$ through $\lg N-1$. By Lemma 4.2, no matter what offset o we add to the source index, the resulting track number is one of at most 2 values. **(b)** When $\beta > BD$, the track number is in bit positions $\lg(BD/P)$ through $\lg(\beta/P)-1$ and $\lg\beta$ through $\lg N-1$. By Lemma 4.2, no matter what offset o we add to the source index, the resulting value in each of these two fields is one of at most 2 values. The resulting track number is therefore one of at most 4 values.

Lemma 4.3 *Consider any d-dimensional mesh or torus permutation on a vector laid out with a banded layout with band size β, where each dimension is a power of* 2.

1. *If $\beta \leq BD$, then the elements of each source-vector track map to at most* 2 *target-vector tracks.*
2. *If $\beta > BD$, then the elements of each source-vector track map to at most* 4 *target-vector tracks.*

Proof: The idea is to partition the bits of the index according to the schemes of Figures 4.4 and 4.5.

We first consider the case for $\beta \leq BD$. As Figures 4.4(a) and 4.6(a) show, only the most significant $\lg(N/BD)$ bits give the track number. Consider two source indices x and y that are in the same source track, and without loss of generality, let $x \leq y$. Since x and y are in the same source track, the most significant $\lg(N/BD)$ bits of x and y are equal; let us say that they give the binary representation of the integer a. In a mesh or torus permutation, we add the same offset, say o, to both x and y. We now apply Lemma 4.2, with $r = \lg BD$, $s = \lg(N/BD)$, and $c = o$. By Lemma 4.2, if we examine the track-number bits of $x + o$ and $y + o$, either they are equal or the track number for $y + o$ is 1 greater than the track number for $x + o$. Because we chose x and y arbitrarily within the same track, we see that the target track numbers for any two source addresses within the same track differ by at most 1. Therefore, the elements of each source-vector track map to at most 2 target-vector tracks.

The case for $\beta > BD$ is slightly more complicated. As Figures 4.4(b) and 4.6(b) show, the track-number field is split among two sets of bit positions: $\lg(BD/P)$ through $\lg(\beta/P) - 1$ and $\lg \beta$ through $\lg N - 1$. As we are about to see, the constant 2 for the $\beta \leq BD$ case becomes a 4 in the $\beta > BD$ case because the track-number field is split.

As before, we consider two source indices x and y that are in the same source track, where without loss of generality we have $x \leq y$. Because x and y are in the same source track, the $\lg(\beta/BD)$ bits in positions $\lg(BD/P)$ through $\lg(\beta/P) - 1$ of x and y are equal; let us say that they give the binary representation of the integer a. Again we add the offset o to both x and y. To see the effect of this addition on the target-address bits in positions $\lg(BD/P)$ through $\lg(\beta/P) - 1$, we apply Lemma 4.2, with $r = \lg(BD/P)$, $s = \lg(\beta/BD)$, and $c = o \bmod (\beta/P)$. Again, we see that the target-address bits in positions $\lg(BD/P)$ through $\lg(\beta/P) - 1$ are either equal or differ by at most 1. We can apply the same argument to the remaining track-number bits in positions $\lg \beta$ through $\lg N - 1$ (here, $r = \lg \beta$, $s = \lg(N/\beta)$, $c = o$, and a is the binary representation of source index bits $\lg \beta$ through $\lg N - 1$) to conclude that the target-address bits in positions $\lg \beta$ through $\lg N - 1$ are either equal or differ by at most 1. The target track mapped to by a given source index is either the same as the source target track or it may be 1 greater in one or both track-number fields. Thus, each source index within a track maps to one of 4 possible target tracks. ■

The key to the proof of Lemma 4.3 is that the track-number bits in Figures 4.4 and 4.6 fall into just one (if $\beta \leq BD$) or two (if $\beta > BD$) fields of bits. Each field of track bits doubles the number of target tracks that the elements of each source track can map to.

If the grid dimensions and band sizes match up just right, we can even lower the constants 2 and 4 in Lemma 4.3. In the $\beta \leq BD$ case, for example, suppose that the line between bit positions $\lg(N/BD)$ and $\lg(N/BD) - 1$ is also the line between two dimensions. (That is, suppose that $m_i m_{i+1} \cdots m_d = BD$ for some dimension i.) Then the bits below position $\lg(N/BD)$ have no effect on the bits in positions $\lg(N/BD)$ through $\lg(N/BD) - 1$, and so any two source indices in the same source-vector track map to exactly the same target-vector track. In this case, we can reduce the constant 2 to just 1. Similarly, in the $\beta > BD$ case, if a line between dimensions matches up with either the line between positions $\lg(BD/P)$ and $\lg(BD/P)-1$ or the line between positions $\lg\beta$ and $\lg\beta-1$, then we can reduce the constant 4 to just 2 or, if the right side of both track-number fields match up with lines between dimensions, just 1.

Finally, we put the above lemmas together to conclude that small band sizes are better for mesh and torus permutations.

Theorem 4.4 *Let N be a power of 2, and consider any d-dimensional mesh or torus permutation on an N-element vector laid out with a banded layout with band size β.*

1. *If $\beta \leq BD$ and $3BD \leq M$, then we can perform the permutation with at most $5N/BD$ parallel I/Os.*
2. *If $\beta > BD$ and $5BD \leq M$, then we can perform the permutation with at most $9N/BD$ parallel I/Os.*

Proof: The proof is a simple application of Lemmas 4.1 and 4.3. If $\beta \leq BD$, we apply Lemma 4.1 with $k = 2$. If $\beta > BD$, we apply Lemma 4.1 with $k = 4$. ■

Thus, we can perform mesh and torus permutations efficiently regardless of the band size, but band sizes less than or equal to the track size are more efficient.

4.4 Scans and reduces with banded layout

This section presents an algorithm to perform scan operations on vectors with banded layout. It also derives a formula for the time to perform the scan. Section 4.5 analyzes this formula to determine the optimal band and I/O buffer sizes. This section also briefly looks at reduce operations, showing that the band size has no effect on their performance.

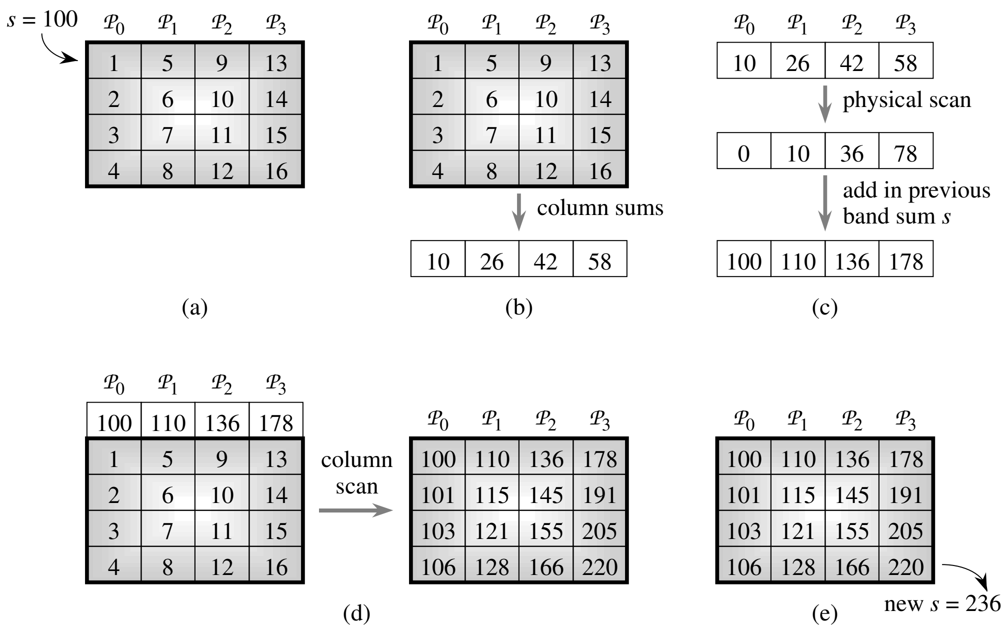


Figure 4.7: Processing a band when scanning. **(a)** The sum s of all prior bands is 100. **(b)** Reduce down the columns. **(c)** Perform a physical scan on the column sums and add s into the scan of the column sums. **(d)** Prepend the row produced in (c) to the band as an initial row, and then scan down the columns. **(e)** Compute a new sum s of bands for use by the next band.

The scan algorithm

We perform a scan operation on a banded vector band by band, making two passes over each band. The method we present is efficient for general band sizes, and it also yields efficient scan algorithms for the extreme cases of row-major and column-major layout. We assume that the parallel machine provides hardware to perform a physical scan operation in which each of the P processors contains one element.

Figure 4.7 shows how we operate on an individual band. For each band except the first, we are given the sum s of all elements in prior bands, as shown in Figure 4.7(a). We do the following:

1. Figure 4.7(b):

   Reduce down the columns of the band. That is, step row-by-row through the values in each processor to produce the P sums of the values in each processor.

2. Figure 4.7(c):

   Perform a physical scan on the column sums and, except for the first band, add to this scan result the sum s from the prior bands.

3. Figure 4.7(d):

   Prepend this set of P values to the band as an initial row and then scan down the columns.

4. Figure 4.7(e):

   For each band except the last, add the result in the last position of the band to the element that started there to compute a new sum s to pass to the next band.

Machine parameters

The formula we shall derive for the time to perform a scan operation will depend on three performance parameters for the machine:

1. A denotes the time to perform an arithmetic operation in parallel for all P processors.

2. S denotes the time to perform a physical scan operation across all P processors.

3. IO denotes the time to perform one parallel disk I/O operation.

For most machines, $A \ll S \ll IO$. Typical estimates for these parameters might be

$$\begin{aligned} A &\approx 40 \text{ ns} , \\ S &\approx 4\ \mu\text{s} , \\ IO &\approx 20 \text{ ms} , \end{aligned}$$

so that $S \approx 100A$ and $IO \approx 5000S$. Because the cost of performing disk I/O is so high, it is important to minimize the number of disk I/O operations.

Processing costs

To derive a formula for the scan time, we start by counting only the time for arithmetic and physical scans. Later, we shall include disk I/O time, which is more difficult to account for.

1. Because each band has β/P rows, each column sum requires $\beta/P-1$ arithmetic operations. These operations are performed in parallel across the P processors. Multiplied by N/β bands, the column reduces total $(N/P - N/\beta)A$ arithmetic time.

2. There is one physical scan per band, and one arithmetic operation per band (except for the first) to add the scan result to the sum s from prior bands. Multiplied by N/β bands, these operations take $(N/\beta - 1)A + (N/\beta)S$ time.

3. Each scan down the columns takes $\beta/P - 1$ arithmetic operations per band. It is not β/P operations because we can form the first row of the band by copying the prepended row without performing any arithmetic. Multiplied by N/β bands, the column scans total $(N/P - N/\beta)A$ arithmetic time.

4. We perform one arithmetic operation per band (except for the last) to create the new sum s, totaling $(N/\beta - 1)A$ time.

Adding up these costs, we get a total processing time of

$$T_{\text{processing}}(\beta) = 2\left(\frac{N}{P} - 1\right)A + \frac{N}{\beta}S$$

for arithmetic and physical scan operations. In the absence of I/O costs, processing time strictly decreases as the band size β increases. When all data fits in RAM, therefore, column-major layout yields the fastest scan time since it uses the maximum band size.

Disk I/O costs

I/O costs generally dominate the cost of the scan computation, especially for large vectors.

Before we can analyze the I/O costs, we need to examine how we organize RAM during the scan operation. A disk read needs an area of RAM to read into, and a disk write needs an area of RAM to write from. We call this area the *I/O buffer*. It is large enough to hold F records, where F is a parameter whose value we will choose later to optimize performance. There are some restrictions on the I/O buffer size F. We require that $F \geq BD$, so that the I/O buffer is at least one track in size, and that $F \leq M$, so that the I/O buffer is no greater than the RAM size. These F records comprise F/BD track frames. Although the BD record locations in each track frame are consecutive, the F/BD track frames need not be.

As one can see from the above description of the scan algorithm, once data has been read into the I/O buffer, we can perform the column reduces and column scans entirely within the buffer. In other words, except for the physical scan and adding in the previous band sums (which involve only P records, independent of all other parameters), we can perform all the work of the scan within the I/O buffer. Therefore, we assume that the only portion of the vector that is in RAM at any one time is in the I/O buffer. If we need to hold more of the vector, we do so by increasing the size F of the buffer. This assumption may be a pessimistic one, since there might be tracks of the vector being scanned residing in the $M - F$ records of RAM that are not allocated to the I/O buffer. A system could save the cost of some disk accesses by using the tracks already in RAM. To keep the analysis in Section 4.5 simple, however, we ignore such sections of the vector. Instead, we read each track into RAM each time it is needed and write each track out to disk once its scan values are computed.

If we are also running a demand paging system, allocation of the F-record I/O buffer may itself incur some I/O costs. These occur from two types of tracks in the RAM space allocated to the buffer:

1. Tracks in the buffer that have been changed since they were last brought into RAM. These tracks must be written out to disk before we read in tracks of the vector to be scanned.

2. Tracks in the buffer that are not needed for the scan but will be needed in a later operation. Losing these tracks from RAM incurs a cost later on.

We call either of these types of tracks *penalty tracks*.

Unfortunately, we have no way to determine until run time how many penalty tracks there are. Generally speaking, the larger the I/O buffer, the more penalty tracks, because there are that many more chances for a given track image to be in the RAM space allocated to the I/O buffer. We model the number of penalty tracks by a function $\phi(F)$, with the understanding that any such function provides only an approximation of reality. No matter what function $\phi(F)$

we use, any given scan operation may have more than $\phi(F)$ or less than $\phi(F)$ penalty tracks. We leave the exact form of $\phi(F)$ unspecified until we analyze the cost of the scan operation in Section 4.5. We will assume, however, that $\phi(F)$ is monotonically increasing in F. Whatever function $\phi(F)$ we use, allocating the I/O buffer incurs a cost of one disk access per penalty track. Accordingly, we set

$$T_{\text{buffer}}(F) = \phi(F)\ IO\ .$$

We are now ready to compute the I/O cost of a scan operation. The steps of the scan algorithm for each band incur the following I/O costs:

1. We read each track of each band into RAM once to perform the column reduces. There are β/BD tracks per band and N/β bands, for a total I/O time of $(N/BD)\ IO$. If $\beta > F$, we read the tracks of each band from back to front; we shall see in a moment why we do so. This order does not affect the computation of the column sums. If $\beta \leq F$, it does not matter in what order we read each band's tracks.

2. The physical scan and addition of the prior band sums s require no additional I/O.

3. The number of I/Os per band for the scans down the columns depends on the relative sizes of β and F. If $\beta \leq F$, then the entire band is already in RAM, and so no further reads are required. If $\beta > F$, then the first F/BD tracks of the band are in RAM because we read the tracks from back to front during the column reduces. The remaining $\beta/BD - F/BD$ tracks are no longer in RAM and must be read again. In either case, all β/BD tracks are written out. The I/O time for the scans, summed over all N/β bands is thus $(N/BD)\ IO$ if $\beta \leq F$ and

$$\begin{aligned} \left(\frac{2\beta}{BD} - \frac{F}{BD}\right)\frac{N}{\beta}\ IO &= \left(2 - \frac{F}{\beta}\right)\frac{N}{BD}\ IO \\ &> \frac{N}{BD}\ IO \end{aligned}$$

   if $\beta > F$.

4. Creating the new sum s requires no additional I/O.

Adding up these costs, we get a total I/O time of

$$T_{\mathrm{I/O}}(\beta, F) = \begin{cases} 2 \dfrac{N}{BD}\, IO & \text{if } \beta \leq F \ , \\ \left(3 - \dfrac{F}{\beta}\right) \dfrac{N}{BD}\, IO & \text{if } \beta > F \ , \end{cases}$$

where $P \leq \beta \leq N$ and $BD \leq F \leq M$.

We make some observations at this point. First, and perhaps most important, the value in the first case is never greater than the value in the second case, since $\beta > F$ implies that $3 - \beta/F > 2$. Second, when $\beta = F$, the two cases of this formula result in the same value, so we can say that the second case holds for $\beta \geq F$, not just $\beta > F$. Third, when $\beta \leq F$, the function $T_{\mathrm{I/O}}(\beta, F)$ does not depend on the band size β. Fourth, when $\beta > F$, the value of $T_{\mathrm{I/O}}(\beta, F)$ strictly increases with β and it strictly decreases with F.

Total scan costs

The total scan time is the sum of the individual costs:

$$T_{\mathrm{scan}}(\beta, F) = T_{\mathrm{processing}}(\beta) + T_{\mathrm{I/O}}(\beta, F) + T_{\mathrm{buffer}}(F) \ . \tag{4.1}$$

This cost function reflects the fact that we cannot begin processing a buffer of input values until it has been read into RAM, and we cannot begin writing out the scan results until they have been computed. Thus, we sum the costs $T_{\mathrm{processing}}(\beta)$ and $T_{\mathrm{I/O}}(\beta, F)$ rather than, say, taking the maximum of the two costs, which would model overlapping I/O and computation.

Section 4.5 shows how to choose optimal values for the band size β and the I/O buffer size F under cost function (4.1).

Reduce operations

The band size has no effect on the performance of reduce operations with banded layout. We can perform any reduce operation using an I/O buffer only one track in size.

When the operator $\oplus$ is commutative as well as associative, we can sum the elements in any order. Therefore, we can read them in any order, and so we can read the vector track by track. The band size does not matter.

If the operator $\oplus$ is not commutative, we have two cases. In either case, we compute the sum band by band, having added in the sum s of all previous bands.

In the first case, the band size is less than or equal to the track size. Reading the vector track by track still allows us to compute the sums band by band, since each band fits within a track.

In the second case, the band size exceeds the track size. To compute each band sum, we maintain P running sums, one for each column of the band. We keep these sums from track to track within the band, and we update them for each new track within the band. When we reach the end of a band, we sum the P column sums together to compute the new sum s of all bands and then initialize them to the identity for $\oplus$ before starting on the next band.

Thus we see that regardless of the band size and whether or not the operator $\oplus$ is commutative, we can perform any reduce operation in one read pass with the minimum I/O buffer size.

4.5 Choosing optimal band and I/O buffer sizes for scans

In this section, we analyze the cost function, equation (4.1), for the scan operation to determine the optimal band size β and I/O buffer size F. We shall study two different scenarios for the number of penalty tracks. In one, we assume that there are no penalty tracks; we shall conclude that optimal scan performance occurs when $\beta = F = M$, that is, when the band size and I/O buffer size are both equal to the RAM size. In the other scenario, we assume that the number of penalty tracks is proportional to the I/O buffer size; we shall conclude that optimal scan performance occurs when $\beta = F$. The exact values to which we should set β and F in this case depend on several of the other parameters. If a fraction $0 \le \alpha \le 1$ of the tracks in the I/O buffer are penalty tracks, we should set β and F to a power of 2 that is near $\sqrt{\frac{NBD}{\alpha}\frac{S}{IO}}$ and is between BD and M.

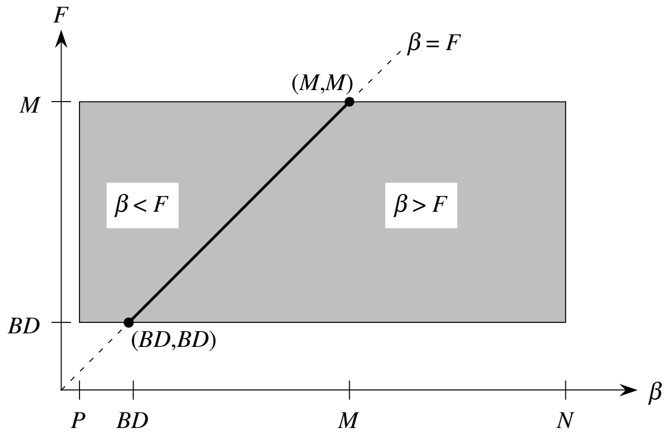


Figure 4.8: The domain of the band size β and the I/O buffer size F for the total scan cost $T_{\text{scan}}(\beta, F)$. The line $\beta = F$ divides the two cases of the cost function.

Analysis for no penalty tracks

Under the assumption that there are no penalty tracks, we have $\phi(F) = 0$ for all I/O buffer sizes F. The following theorem shows that under the reasonable assumption that the time to perform a disk access exceeds the time to perform a physical scan, both the band size and I/O buffer size should equal the RAM size in the absence of penalty tracks.

Theorem 4.5 *If there are no penalty tracks and $IO > S$, then $T_{\text{scan}}(\beta, F)$ is minimized when $\beta = F = M$.*

Proof: When $\phi(F) = 0$, we have

$$T_{\text{scan}}(\beta, F) = \begin{cases} 2\left(\dfrac{N}{P} - 1\right) A + \dfrac{N}{\beta} S + 2\dfrac{N}{BD}\, IO & \text{if } \beta \le F \ , \\[2ex] 2\left(\dfrac{N}{P} - 1\right) A + \dfrac{N}{\beta} S + \left(3 - \dfrac{F}{\beta}\right) \dfrac{N}{BD}\, IO & \text{if } \beta \ge F \ . \end{cases}$$

Figure 4.8 shows the domain of the function $T_{\text{scan}}(\beta, F)$, including the line $\beta = F$, which divides the two cases.

We determine the optimal values of β and F by applying standard calculus techniques. We treat the two regions in Figure 4.8, for $\beta \le F$ and for $\beta \ge F$, separately. If the interior of either one contains a local minimum, then both partial derivatives of T_{scan}—with respect to β and to F—must equal 0 at that point. We claim that neither region's interior contains a local minimum, because $IO > S$ implies that $\partial T_{\text{scan}}(\beta, F)/\partial\beta < 0$ for all β in the domain. We have

$$\frac{\partial T_{\text{scan}}(\beta, F)}{\partial\beta} = \begin{cases} -\dfrac{NS}{\beta^2} & \text{if } \beta \le F \ , \\ -\dfrac{N}{\beta^2}\left(\dfrac{F}{BD}\ IO - S\right) & \text{if } \beta \ge F \ . \end{cases} \tag{4.2}$$

Because $\beta \ge P > 0$, this derivative is negative whenever $\beta \le F$. Moreover, because $IO > S$ and $F \ge BD$, the coefficient $((F/BD)\ IO - S)$ is positive, and so the derivative (4.2) is negative whenever $\beta \ge F$ as well.

Having ruled out the interior of either region for the minimum value, we now examine the region boundaries. We first look at the boundary for $\beta \le F$. In this case, $T_{\text{scan}}(\beta, F)$ is a decreasing function of β and does not depend on F. Therefore, the cost is minimized by choosing the rightmost possible value of β on the boundary, namely $\beta = F = M$.

Now we look at the boundary for $\beta \ge F$. Because $\partial T_{\text{scan}}(\beta, F)/\partial\beta < 0$, the minimum cannot be on either of the horizontal boundaries of the region except for the corners. Also, we have that $\partial T_{\text{scan}}(\beta, F)/\partial F = -(N/\beta BD)\ IO < 0$, and so the minimum cannot be on the right boundary except for the corners. That leaves only the corners and the boundary marked by $BD \le \beta = F \le M$. We examine the corners and this boundary one by one.

- $T_{\text{scan}}(M, M) = 2(N/P - 1)A + (N/M)S + 2(N/BD)\ IO$. This value turns out to be the minimum.

- $T_{\text{scan}}(N, M) = 2(N/P - 1)A + S + (3 - M/N)(N/BD)\ IO$. Simple manipulation shows that this value is less than $T_{\text{scan}}(M, M)$ if and only if $IO/S < BD/M$. Since $BD \le M$ and $IO > S$, we conclude that $T_{\text{scan}}(N, M) > T_{\text{scan}}(M, M)$.

- $T_{\text{scan}}(N, BD) = 2(N/P - 1)A + S + (3N/BD - 1)\ IO$. Simple manipulation shows that this value is less than $T_{\text{scan}}(M, M)$ if and only if $IO/S < (BD/M)((N - M)/(N - BD))$. Because $BD \le M$, both of the fractions on the right side are at most 1, so we conclude that $T_{\text{scan}}(N, BD) > T_{\text{scan}}(M, M)$.

- $T_{\text{scan}}(BD, BD) = 2(N/P-1)A + (N/BD)S + 2(N/BD)\ IO$. Since $BD \le M$, this value is never less than $T_{\text{scan}}(M, M)$.
- On the boundary marked by $BD \le \beta = F \le M$, we have $T_{\text{scan}}(\beta, F) = 2(N/P-1)A + (N/\beta)S + 2(N/BD)\ IO$, which is greater than $T_{\text{scan}}(M, M)$ everywhere except for $\beta = M$.

We conclude that the function $T_{\text{scan}}(\beta, F)$ is minimized at the point $\beta = F = M$. ■

It should come as no surprise that when there are no penalty tracks, we want large band and I/O buffer sizes. Because the number of physical scans decreases as the band size increases, we want a large band size. Because there is no cost to having a large I/O buffer size, we want a large one in order to accommodate a large band size. Theorem 4.5 serves to formalize these notions, and it also proves that increasing the band size beyond the buffer size incurs I/O costs that we do not recoup by reducing the number of physical scans.

Analysis for penalty tracks proportional to I/O buffer size

Now we look at a scenario in which the number of penalty tracks is proportional to the I/O buffer size. We set

$$\phi(F) = \frac{\alpha F}{BD} \ ,$$

where α is a fraction in the range $0 \le \alpha \le 1$. This function models the situation in which each track frame originally in the RAM space occupied by the I/O buffer is a penalty track with probability α.

The following theorem shows that in this case, we always want the band size to equal the I/O buffer size, but the exact value we want depends on several parameters.

Theorem 4.6 *If $IO > S$ and the number of penalty tracks is given by $\phi(F) = \alpha F/BD$, where $0 \le \alpha \le 1$, then $T_{\text{scan}}(\beta, F)$ is minimized only if $\beta = F$. Furthermore, if we define*

$$t^* = \sqrt{\frac{NBD}{\alpha}\frac{S}{IO}} \ ,$$

then the following hold:

1. *If $BD \le t^* \le M$, then $T_{\text{scan}}(\beta, F)$ is minimized by choosing β and F to be t^* if it is a power of 2 and otherwise choosing β and F to be one of the two closest powers of 2 to t^*.*

2. *If $t^* \le BD$, then $T_{\text{scan}}(\beta, F)$ is minimized by choosing $\beta = F = BD$.*

3. *If $t^* \ge M$, then $T_{\text{scan}}(\beta, F)$ is minimized by choosing $\beta = F = M$.*

Proof: When $\phi(F) = \alpha F/BD$, we have

$$T_{\text{scan}}(\beta, F) = \begin{cases} 2\left(\dfrac{N}{P} - 1\right)A + \dfrac{N}{\beta}S + 2\dfrac{N}{BD}\,IO + \dfrac{\alpha F}{BD}\,IO & \text{if } \beta \le F\ , \\[2ex] 2\left(\dfrac{N}{P} - 1\right)A + \dfrac{N}{\beta}S + \left(3 - \dfrac{F}{\beta}\right)\dfrac{N}{BD}\,IO + \dfrac{\alpha F}{BD}\,IO & \text{if } \beta \ge F\ . \end{cases}$$

As in the proof of Theorem 4.5, Figure 4.8 shows the domain of the function $T_{\text{scan}}(\beta, F)$, including the line $\beta = F$, which divides the two cases.

We again apply standard calculus techniques, treating the two regions separately. Because $\phi(F)$ does not depend on β, the partial derivative $\partial T_{\text{scan}}(\beta, F)/\partial\beta$ is again given by equation (4.2). As in Theorem 4.5, we rule out the interior of both regions for the minimum value because this derivative is negative for all positive values of β.

Again, we check the boundaries and corners of the two regions, starting with the region $\beta \le F$. As in Theorem 4.5, $\partial T_{\text{scan}}(\beta, F)/\partial\beta < 0$, and so the minimum cannot be on either of the horizontal boundaries of the region except for the corners. For the left boundary, we have $\partial T_{\text{scan}}(\beta, F)/\partial F = (\alpha/BD)\,IO$, which equals 0 only if $\alpha = 0$; in this case, $T_{\text{scan}}(\beta, F)$ does not depend on F in the region $\beta \le F$, and because $T_{\text{scan}}(\beta, F)$ is a decreasing function of β, the minimum cannot be on the left boundary. We are left with only the corners and the boundary marked by $BD \le \beta = F \le M$, which we examine one by one.

- $T_{\text{scan}}(BD, BD) = 2(N/P - 1)A + (N/BD)S + 2(N/BD)\,IO + \alpha\,IO$.

- $T_{\text{scan}}(M, M) = 2(N/P - 1)A + (N/M)S + 2(N/BD)\,IO + (\alpha M/BD)\,IO$. Simple manipulation shows that this value is less than $T_{\text{scan}}(BD, BD)$ if and only if $IO/S < N/\alpha M$.

- $T_{\text{scan}}(P, BD) = 2(N/P - 1)A + (N/P)S + 2(N/BD)\ IO + \alpha\ IO$. Because $BD \geq P$, this value is never less than $T_{\text{scan}}(BD, BD)$.

- $T_{\text{scan}}(P, M) = 2(N/P - 1)A + (N/P)S + 2(N/BD)\ IO + (\alpha M/BD)\ IO$. Because $M \geq P$, this value is never less than $T_{\text{scan}}(BD, BD)$.

- On the boundary marked by $BD \leq \beta = F \leq M$, we parameterize the function and simplify it as

$$T_{\text{scan}}(t) = \frac{a}{t} + bt + c \ ,$$

where $a = NS$, $b = (\alpha/BD)\ IO$, and $c = 2(N/P - 1)A + 2(N/BD)\ IO$. We take the derivative with respect to t:

$$\frac{d\,T_{\text{scan}}(t)}{dt} = -\frac{a}{t^2} + b \ .$$

If there is a minimum along this boundary, it occurs where $d\,T_{\text{scan}}(t)/dt = 0$, or at

$$t^* = \sqrt{\frac{a}{b}} \ . \tag{4.3}$$

This extreme point is in fact a local minimum of $T_{\text{scan}}(t)$, since its second derivative is

$$\frac{d^2\,T_{\text{scan}}(t)}{dt^2} = \frac{a}{t^3} \ ,$$

which is positive for all positive values of t. Observe that at the point t^*, the terms a/t and bt of $T_{\text{scan}}(t)$ both equal $\sqrt{ab}$, so they balance each other out. The value

$$t^* = \sqrt{\frac{NBD}{\alpha}\,\frac{S}{IO}} \ ,$$

in the theorem statement comes from substituting $a = NS$ and $b = (\alpha/BD)\ IO$ in equation (4.3).

If $t^* < BD$ or $t^* > M$, then we cannot choose $\beta = F = t^*$. Similarly, if t^* is not a power of 2, we cannot choose $\beta = F = t^*$. Observe, however, that because $T_{\text{scan}}(t)$ has its only extreme point at t^*, it decreases for $t < t^*$ and increases for $t > t^*$. If $t^* < BD$, therefore, we should choose $\beta = F = BD$. If $t^* > M$, we should choose $\beta = F = M$. If $BD \le t^* \le M$ but t^* is not a power of 2, we choose one of the powers of 2 immediately above or below t^*, whichever yields a smaller value of $T_{\text{scan}}(t)$. This completes the analysis of the boundaries and corners of the region $\beta \le F$.

Now we analyze the boundaries and corners of $\beta \ge F$. We have already analyzed the boundary and corners for which $\beta = F$, so we only have to show that the other three boundaries and two corners have higher costs. As before, we rule out the horizontal boundaries because $\partial T_{\text{scan}}(\beta, F)/\partial\beta < 0$. For the right boundary, we observe that

$$\frac{\partial T_{\text{scan}}(\beta, F)}{\partial F} = \left(\frac{\alpha}{BD} - \frac{N}{\beta BD}\right) IO \ ,$$

which equals 0 if and only if $\alpha = 1$ and $\beta = N$. In this case, however, we have $T_{\text{scan}}(\beta, F) = 2(N/P - 1)A + S + 3(N/BD)\ IO$. Simple manipulation shows that this value is less than $T_{\text{scan}}(BD, BD)$ if and only if $IO/S < (N - BD)/N$, which is not true since $IO > S$ and $(N - BD)/N < 1$. Thus, we rule out the right boundary. All that remain are the two rightmost corner points.

- $T_{\text{scan}}(N, BD) = 2(N/P - 1)A + S + 3(N/BD)\ IO + (\alpha - 1)\ IO$. Simple manipulation shows that this value is less than $T_{\text{scan}}(BD, BD)$ if and only if $IO/S < 1$, which is not true.

- $T_{\text{scan}}(N, M) = 2(N/P-1)A+S+3(N/BD)\ IO+(\alpha-1)(M/BD)\ IO$. Simple manipulation shows that this value is less than $T_{\text{scan}}(M, M)$ if and only if $IO/S < BD/M$, which is not true since $BD \le M$.

By exhaustive analysis, therefore, we have proven the theorem. ■

Where does the strange formula for t^* in the statement of Theorem 4.6 come from? It was a subtle point in the middle of the proof, but t^* is the value for β and F for which the cost of the physical scans equals the cost of buffer allocation.

Note that Theorem 4.5 is a special case of Theorem 4.6 in which $\alpha = 0$. We have $t^* = \infty > M$, and so $T_{\text{scan}}(\beta, F)$ is minimized by choosing $\beta = F = M$.

4.6 Band sizes for elementwise operations

Elementwise operations are often the most frequent operations in a data-parallel program. When all record sizes are the same, elementwise operations have no effect on our choice of band size. This section looks at elementwise operations when record sizes differ, arguing in favor of small band sizes.

Logical vs. physical sizes

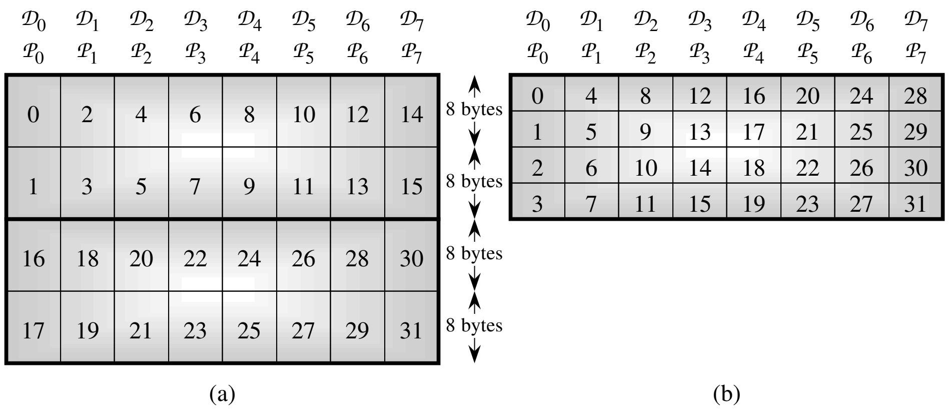
So far, we have been purposely vague as to what constitutes a record, but now let us discuss records in more detail. A record consists of some number of bytes, and different vectors may have elements of different record sizes. In the VM-DP system, for example, the VCODE interpreter supports two record sizes: 4-byte integers and 8-byte double-precision floats.

Sizes of disk blocks and RAM are actually measured in bytes, not records. When we say that RAM holds M records, we are dealing in an abstraction. If the capacity of RAM is M 8-byte floats, it is also $2M$ 4-byte integers. Similarly, a track that holds BD 8-byte floats is also capable of holding $2BD$ 4-byte integers. We call the size of a memory unit measured in bytes its *physical size*, and we call its size in terms of records of a given size its *logical size*.

Mapping elements to processors

Should a system with multiple record sizes also have multiple band sizes? The analyses of the previous sections suggest that we should choose band sizes based on the logical track size BD or the logical RAM size M. These logical sizes depend on their physical sizes and the record size. If we choose the band size to equal the physical track size, for example, we would choose one band size for 8-byte floats and a band size twice as large for 4-byte integers, since twice as many 4-byte records as 8-byte records fit into the same physical track size.

The band size of a vector determines how its elements are mapped to processors. If the band size is β, element X_i maps to processor $\mathcal{P}_{\lfloor (i \bmod \beta) P/\beta \rfloor}$. If the band size is determined by the record size, the mapping of elements to processors depends on the record size.



$\mathcal{D}_0$ $\mathcal{D}_1$ $\mathcal{D}_2$ $\mathcal{D}_3$ $\mathcal{D}_4$ $\mathcal{D}_5$ $\mathcal{D}_6$ $\mathcal{D}_7$
$\mathcal{P}_0$ $\mathcal{P}_1$ $\mathcal{P}_2$ $\mathcal{P}_3$ $\mathcal{P}_4$ $\mathcal{P}_5$ $\mathcal{P}_6$ $\mathcal{P}_7$
0 2 4 6 8 10 12 14
1 3 5 7 9 11 13 15
16 18 20 22 24 26 28 30
17 19 21 23 25 27 29 31
8 bytes
8 bytes
8 bytes
8 bytes
(a)
0 4 8 12 16 20 24 28
1 5 9 13 17 21 25 29
2 6 10 14 18 22 26 30
3 7 11 15 19 23 27 31
(b)

Figure 4.9: When differing record sizes cause band sizes to differ, elements with equal indices do not always map to the same processor. **(a)** The mapping of a 32-element vector of 8-byte values with $B_8 = 2$ records per block and $D = P = 8$. The band size is $\beta_8 = 16$. **(b)** The mapping of a 32-element vector of 4-byte values with $B_4 = 4$ records per block and band size $\beta_4 = 32$.

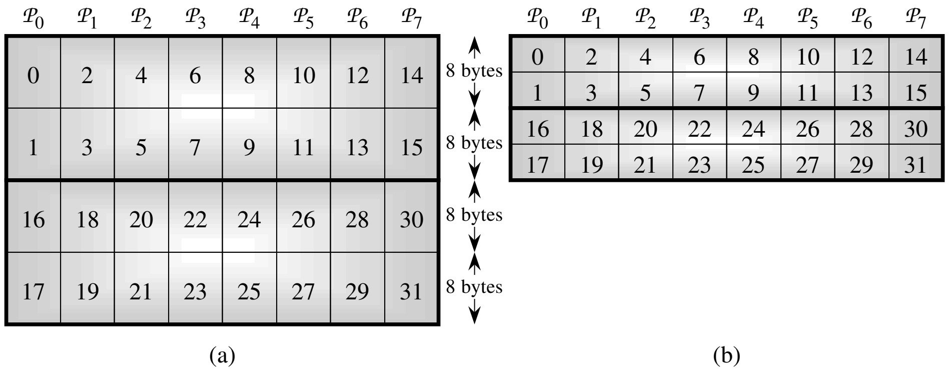


Figure 4.10: How the VM-DP system maps vectors to $P = 8$ processors with a physical track size of 128 bytes. **(a)** A 32-element vector of 8-byte floats. The band size is $\beta = 128/8 = 16$. **(b)** A 32-element vector of 4-byte integers, also mapped with band size $\beta = 16$. Corresponding elements of either type of vector map to the same processor.

The problem with this scheme is that we would like the mapping of element indices to processors to be independent of record size. To see why, consider an elementwise operation with vectors of different record sizes, such as that of converting a vector of 4-byte integers to a vector of 8-byte double-precision values. In an elementwise operation, for each index i, the operand elements with index i and the result element with index i must map to the same processor. This requirement is not always met when the mapping of elements to processors depends on the record size.

As an example, Figure 4.9 shows how two 32-element vectors of 4- and 8-byte elements map to processors when we base band sizes on physical sizes. Here, we have $D = P = 8$, and each physical block holds 16 bytes. For 8-byte elements the logical block size is $B_8 = 2$ elements per block, and for 4-byte elements the logical block size is $B_4 = 4$ elements per block. If we choose the band size to be the number of records that fill a physical track, we get different band sizes for the two element sizes: $\beta_8 = B_8 D = 16$ and $\beta_4 = B_4 D = 32$. The figure clearly shows that many indices do not map to the same processor for the two vectors. The first such index is 2, which maps to processor $\mathcal{P}_0$ in one vector and to $\mathcal{P}_1$ in the other.

Using the smallest band size for all vectors

Figure 4.10 shows the solution that we adopted in the VM-DP system. We use the smallest band size for all vectors. That is, we consider a vector whose elements have the largest record size over all vectors that we will use. We determine β such that if this vector has a band size of β, then each of its bands occupies one physical track. We use this band size β for all vectors, regardless of their record size. All bands are less than or equal to a physical track, and the mapping of elements to processors is independent of each vector's record size.

4.7 Conclusions

We have seen that the band size used to lay out a vector on a parallel disk system in a data-parallel machine affects the performance of several data-parallel operations. There is no one best band size overall. For permuting, we prefer band sizes that are a track or less. For scans, the optimal band size can be anywhere from a track to the size of RAM. For reduces, the band size does not matter. For elementwise operations, the band size can depend on the record size and the physical sizes of the system.

How then should one choose band sizes in an actual data-parallel system? In the VM-DP system, we chose the band size to be the number of 8-byte floats that fit in a physical track, these being the largest record type. The other record type, 4-byte integers, has the same band size; hence each band of 4-byte integers occupies half a physical track. We chose one track for the band size because it yields the best performance for permuting and overall good (if not optimal) performance for scans. Moreover, it made the programming task easier to have each parallel I/O always read or write at least one entire band.

It is important to put this work into perspective. This chapter has been mostly about saving constant factors. We can perform mesh and torus permutations in $\Theta(N/BD)$ parallel I/Os regardless of the band size, but the constant factor is better for small band sizes. In a similar vein, the dominant performance difference for scans between good and poor choices of band and buffer sizes is less than N/BD I/Os. Although differences due to constant factors can add up to a lot of time for huge vectors, it is likely to be more fruitful for us to channel our efforts into algorithms for operations in which we can realize asymptotic savings. The best examples of such algorithms are those for permuting, as in Chapters 2 and 3.

Chapter 5

Paging and Addressing in the VM-DP System

Chapters 2 and 3 looked at algorithms to perform data-parallel operations in which the virtual-memory system choreographed the patterns of disk accesses. This chapter focuses on a different aspect of a virtual-memory system, the one that perhaps most often comes to mind: the demand-paging system.

Specifically, this chapter presents the design of the demand-paging portion of the VM-DP system. We implemented three different paging schemes and collected empirical data on their I/O performance. One of the schemes was straight LRU (least-recently used) paging, in which all tracks are treated equally. The other two schemes treat tracks differently according to the sizes of vectors they contain. Our empirical tests yielded two somewhat surprising results:

- The observed I/O counts for our test suite were roughly equal under all three schemes. Moreover, as problem sizes increased, performance differences among the three schemes decreased.
- Overall, the best scheme of the three was the simplest one: straight LRU paging. The other two schemes, which attempt to account for vast differences in vector sizes, were not quite as good overall.

This chapter also discusses some of the addressing and implementation issues that arose in the VM-DP system.

This chapter represents joint work with Lars Bader.

Measuring sizes in bytes

In this chapter, we need to make a slight modification to the machine model that we defined in Section 1.3 and used in Chapters 1–4. Rather than measure block and RAM sizes in records, this chapter measures them in bytes. In this chapter, M denotes the number of bytes of RAM, and B denotes the number of bytes per disk block. Accordingly, the track size BD is also measured in bytes.

We use bytes rather than records because VCODE, the stack-based, intermediate, data-parallel language that is interpreted by VM-DP supports two different record sizes: 4-byte integers and 8-byte floats. Normalizing all sizes to bytes makes our descriptions independent of record sizes.

When does VM-DP use demand paging?

All operations that access vectors use VM-DP's demand-paging system. Operations other than the permuting operations use it for all their vector accesses. These operations include all elementwise operations, scans, and reduces. In addition, VCODE includes six different operations that permute according to target addresses. These permuting operations go through the paging system at least part of the time; they disable it only during external radix sort[1] (see Section 3.1) and the BPC detection and permutation code (see Sectoin 3.4). Demand paging is enabled during the `pack` operation, which is a type of monotonic route (see Section 3.3).

Outline of this chapter

Since the demand-paging system is used so heavily by VM-DP, its I/O behavior is crucial to overall system performance. To minimize the number of disk accesses, we explored three design alternatives. Section 5.1 presents these three designs and their I/O performance on our test suite. The remaining sections of this chapter focus on details of VM-DP that relate to its paging system. Section 5.2 examines how vectors are mapped to pages and the effect on address-space

[1]Actually, the demand-paging system is enabled during the pass within external radix sort that creates a census of how many records will end up in each bucket in each pass.

allocation. Section 5.3 discusses further implementation issues in the VM-DP paging system.

5.1 Performance of paging schemes in VM-DP

A paging-system designer has many options. This section presents the three paging schemes we implemented for VM-DP and summarizes their performance on our test suite. One scheme is essentially straight LRU paging, and the other two partition RAM according to vector sizes. As we shall see, there is not much difference in I/O performance among the three schemes, and the scheme that had the best I/O performance overall was also the simplest: straight LRU paging.

As we mentioned in Section 1.5, VM-DP is designed for amounts of data in excess of the RAM size, but we also want performance to be good when all the data fits in RAM. In other words, computations that do not really need VM should not pay a high price for its presence in the system.

Paging and LRU replacement

The VM-DP system uses a paging system similar to the approach of using both segmentation and paging, as described by Denning [Den70] and used in the Multics system [BCD72, DD68]. In the VM-DP system, vectors serve as the equivalent of segments in a system such as Multics. Just as Multics addresses consist of a segment number and an index within the segment, we can address individual vector elements by a vector address and an index within the vector. Because vectors, like Multics segments, can take on a range of sizes from very small to larger than RAM, it is infeasible to require that each segment is either entirely in RAM or entirely out of RAM, as is the case in a pure segmented virtual-memory system. Instead, we divide the address space into fixed-size *pages*, we partition the RAM into *page frames* of the same size, and we swap pages in and out of page frames as necessary.

We chose a track of BD bytes as our page size. This size is the minimum feasible size, since it represents the smallest unit of parallel I/O. We decided to use a size no larger because, as we shall see, smaller page sizes lead to less wasted space and more flexibility in the execution of the page-replacement policy. Because any given vector may be smaller than, the same size as, or larger than a page, any given page may contain several vectors, exactly one vector, or just a portion of a vector.

Having decided that we were going to use paging, the next question was what page-replacement strategy to use when performing demand paging. In *demand paging*, we remove a page from RAM only when we need to find a free page frame and all page frames are in use. A *page fault* occurs when we need to access a page that is not currently in RAM.

Least-recently used, or LRU, replacement is a provably good strategy. In *LRU replacement*, we maintain the order in which page frames in RAM have been accessed for either reading or writing, and we always replace the frame whose most recent access (for either reading or writing) was the longest ago.

Sleator and Tarjan [ST85] proved that LRU is "competitive" with any *on-line* page-replacement policy, that is, any policy that foregoes knowledge about future accesses. They showed that in the worst case, no on-line replacement policy can produce even within a constant factor of the number of page faults produced by an optimal off-line policy (i.e., one that uses foreknowledge about the sequence of page accesses). Moreover, they also proved that overall, LRU is as good as any other on-line policy. There are particular page-access sequences for which another on-line policy may produce fewer page faults than LRU, but even in the worst case, LRU comes within a constant factor of the number of page faults made by an optimal off-line policy with a constant factor smaller RAM.

We use a true LRU policy rather than an approximation such as a "use bit" [PH90, p. 436]. The overhead of using true LRU is spread over the many elements involved in a single access.

Small and large vectors

VM-DP classifies vectors according to size when allocating them to tracks. A *small vector* requires one track (BD bytes) or less, and a *large vector* requires more than one track. Each track is in one of three states: completely unallocated, containing small vectors, or containing part of a large vector. A track that contains one or more small vectors contains no part of any large vector, and a track that contains part of a large vector contains only that part of the large vector. In other words, no track contains both large and small vectors. This distinction can help our paging scheme and, as we shall see in Section 5.2, it also helps in vector allocation.

VM-DP paging schemes

We now present the three paging schemes that we investigated in VM-DP. Each scheme partitions RAM into one or two portions. Tracks are classified according to the sizes of vectors they contain. Each portion of RAM contains page frames for one class of tracks. Page replacement within each portion is performed on a strict LRU basis.

- In the *one-space scheme*, all of RAM is one portion, containing M/BD page frames, and there is only one classification of track. In other words, the one-space scheme is straight LRU paging. It is the simplest of the three schemes and, according to our empirical measurements, it has the best I/O performance overall.

- The *two-space scheme* partitions RAM into two equal-sized portions. The *small-vector space* contains $M/2BD$ page frames and is dedicated only to tracks containing small vectors. The *large-vector space* also contains $M/2BD$ page frames, and it is dedicated only to tracks containing parts of large vectors. Because no track contains both large and small vectors, each allocated track is eligible to belong to exactly one of these spaces.

- In the *ω-RAM scheme*, we partition RAM into two unequal portions. The *ω-RAM space* contains five page frames and is dedicated only to tracks containing parts of vectors that are at least $M/4$ bytes in size. The *O-RAM space* contains the remaining $M/BD-5$ page frames and is dedicated only to small-vector tracks or to large-vector tracks containing parts of vectors that are less than $M/4$ bytes in size.[2]

[2]The names "ω-RAM" and "O-RAM" were chosen by analogy to asymptotic notation. A function $g(n)$ is $\omega(f(n))$ if $0 \leq f(n) < g(n)$ for some constant $c > 0$ and sufficiently large n. A function $g(n)$ is $O(f(n))$ if $0 \leq g(n) \leq cf(n)$ for some constant $c > 0$ and sufficiently large n. In our discussions, we called a vector "ω-RAM" if it is more than a constant—1/4 as it turned out—of the RAM size and "O-RAM" if its size is at most this constant fraction of RAM.

There are other schemes that we could have investigated, but we chose the above schemes because they are fairly simple and also for the following reasons.

The one-space scheme was an obvious candidate because it is straight LRU.

The ω-RAM scheme was an outgrowth of the following observation. Suppose we use straight LRU paging and we perform a simple elementwise operation, say addition of two vectors, on very large vectors, say over a third of the RAM size each. By the time we have the last track of each of the two source vectors and the one result vector in RAM, the first track of each has been thrown out of RAM. If we access these vectors again—and we are likely to access the result vector soon, if not also one of the operands—each track will produce a fault. Not only does straight LRU paging fail to improve the I/O behavior of accessing large vectors, but any small vectors that were in RAM prior to the first elementwise operation have been thrown out as well. Since LRU paging does not help in accessing very large vectors, by separating page frames for these vectors from those for smaller vectors and also limiting the number of page frames for very large vectors, we prevent operations on very large vectors from causing smaller vectors to be ejected from RAM for no gain. We chose to use five page frames for very large vectors because one CVL function that VM-DP supports uses five vectors and no CVL function uses more. We chose the size $M/4$ as the smallest ω-RAM vector because most CVL functions use at most four vectors, and so if all of them are paged in, they exhaust all of RAM.

The rationale behind the two-space scheme is similar to that of the ω-RAM scheme, but with a view toward preventing *intermediate vectors*—larger than a track but at most $M/4$ bytes—from interfering with small vectors. That is, if a computation processes small vectors, then intermediate vectors, and then small vectors, the processing of intermediate vectors should not cause all the small vectors to be thrown out of RAM.

Each partition in each scheme uses a straight LRU replacement policy for the reasons we mentioned above. An LRU policy does not really improve the I/O performance of the ω-RAM space in the ω-RAM scheme, but neither does it hurt and it was easily implemented because all the other partitions we implemented already used it.

Test suite

Our test suite consisted of five fairly small programs written in NESL, which were provided to us by Blelloch's research group at Carnegie Mellon. Descriptions of three of the programs—`convex-hull`, `primes`, and `qsort`—appear in the NESL documentation [Ble92].

- `convex-hull` is an implementation of the recursive quickhull algorithm (see [Ble90] and [PS85, pp. 106–108]) for computing the convex hull of a set of points in two dimensions. The goal is to find two polygonal chains whose concatenation forms the convex hull of the given set of points. The quickhull algorithm divides the input set into two sets of points, recursively finds a polygonal chain for each set, and then merges the chains together.

- `linefit` is a NESL implementation of linear regression (see Press et. al [PFTV89, pp. 553–558]) for fitting points in two dimensions to a straight line. It is straight-line code with no recursion.

- `order` is a recursive function that finds the median or any order statistic of a given set of values. It is a deterministic version of the algorithm in [CLR90, pp. 187–189]. Each recursive invocation removes several values that cannot be the desired order statistic, and so the length of the vector in each recursive call strictly decreases over time.

- `primes` uses the sieve of Eratosthenes to find all the prime numbers less than n. It recursively finds all primes less than $\sqrt{n}$ and then uses these prime numbers to sieve out candidate primes greater than or equal to $\sqrt{n}$.

- `qsort` is an implementation of quicksort (see [Ble90] and [CLR90, Chapter 8]). To sort a set of numbers, it chooses one of them as a "pivot," partitions the numbers into those less than, equal to, and greater than the pivot, and recurses on these sets.

Empirical results

Our empirical tests of the three paging schemes on the five test programs were VM-DP simulations with $P = 16$ processors, $D = 4$ disks, $B = 128$ bytes per block, and $M = 16384$ bytes

N	one-space	two-space	ω-RAM	worst/best
32	41	306	82	7.463
128	667	971	780	1.456
512	3930	5694	4752	1.449
2048	21140	22453	22390	1.062
8192	89260	90104	89886	1.009

Table 5.1: I/O counts for `convex` running under the three paging schemes, along with the ratio for the worst scheme to the best scheme for each input size.

N	one-space	two-space	ω-RAM	worst/best
128	6	6	6	1.000
512	73	188	99	2.575
2048	1183	1217	1217	1.029
8192	4836	4817	4817	1.004
32768	19236	19217	19217	1.001

Table 5.2: I/O counts for `linefit` running under the three paging schemes.

of RAM in all. These parameters may seem small, but space and time limitations prevented us from simulating with parameters significantly larger. Moreover, these parameters exercised the paging system to a greater degree than larger parameters would have. The simulations we ran with other parameter values yielded essentially the same results in comparing paging schemes as those we obtained with the above parameters.

- Table 5.1 shows the results for `convex`. The one-space scheme performed the fewest I/Os for each problem size, but for $N = 2048$ points, the best and worst schemes differed by only 6.2 percent, and for $N = 8192$ points, the best and worst schemes differed by only 0.9 percent.

- Table 5.2 shows the results for `linefit`. The one-space scheme performed the fewest I/Os for smaller problem sizes, but the two-space and ω-RAM schemes tied for the fewest I/Os for larger problem sizes. For $N = 8192$ points, the best and worst schemes differed by only 0.4 percent, and for $N = 32768$ points, the best and worst schemes differed by only

N	one-space	two-space	ω-RAM	worst/best
32	4	4	4	1.000
128	6	6	6	1.000
512	3	20	3	6.667
2048	288	520	398	1.806
8192	2174	2492	2397	1.147
32768	10663	10676	10616	1.006

Table 5.3: I/O counts for `order` running under the three paging schemes. We do not know why the I/O counts for $N = 512$ should be less than those for $N = 128$ for the one-space and ω-RAM schemes.

N	one-space	two-space	ω-RAM	worst/best
128	8	8	8	1.000
512	217	285	228	1.313
2048	1762	1796	1839	1.044
8192	7299	7475	7530	1.032
32768	30683	30686	30737	1.002

Table 5.4: I/O counts for `primes` running under the three paging schemes. This is the only program in the test suite for which the two-space scheme ever outperformed the ω-RAM scheme.

0.1 percent.

- Table 5.3 shows the results for `order`. The one-space scheme performed the fewest I/Os for all but the largest problem size we ran, in which ω-RAM was the best. For $N = 8192$ numbers, the best and worst schemes differed by 14.7 percent, but for $N = 32768$ numbers, the best and worst schemes differed by only 0.6 percent.

- Table 5.4 shows the results for `primes`. The one-space scheme performed the fewest I/Os for all problem sizes. This program was the only one in our test suite for which the ω-RAM scheme was worse than the two-space scheme. For $N = 8192$ numbers, the best and worst schemes differed by 3.2 percent, and for $N = 32768$ numbers, the best and worst schemes differed by only 0.2 percent.

N	one-space	two-space	ω-RAM	worst/best
32	26	89	26	3.423
128	139	316	208	2.273
512	10808	11756	11169	1.088
2048	56206	66372	65968	1.181
8192	374489	383915	379426	1.025

Table 5.5: I/O counts for `qsort` running under the three paging schemes.

- Table 5.5 shows the results for `qsort`. The one-space scheme performed the fewest I/Os for all problem sizes. For $N = 2048$ numbers, the best and worst schemes differed by 18.1 percent, but for $N = 8192$ numbers, the best and worst schemes differed by 2.5 percent.

In general, especially for larger vector sizes, the differences among the three schemes were small, often under 1 percent. The one-space scheme, which is the simplest to implement, performed the fewest I/Os for all problem sizes we considered for `convex`, `primes`, and `qsort`. Moreover, for those problem sizes of `linefit` and `order` for which the one-space scheme was not the best, it was within 1.1 percent of the best scheme. According to our empirical study, one-space is the scheme of choice among these three.

5.2 Vector allocation

The classification of vectors into large and small sizes affects how the VM-DP system manages its vector address space. This section discusses how vectors are mapped to pages and the effect of the mapping on address-space allocation. Large vectors and small vectors are mapped according to different sets of rules.

The address space

The restriction that the initial element of each vector maps to the RAM of processor $\mathcal{P}_0$ affects how we view the address space of vectors. When a vector or portion thereof is in RAM, we can

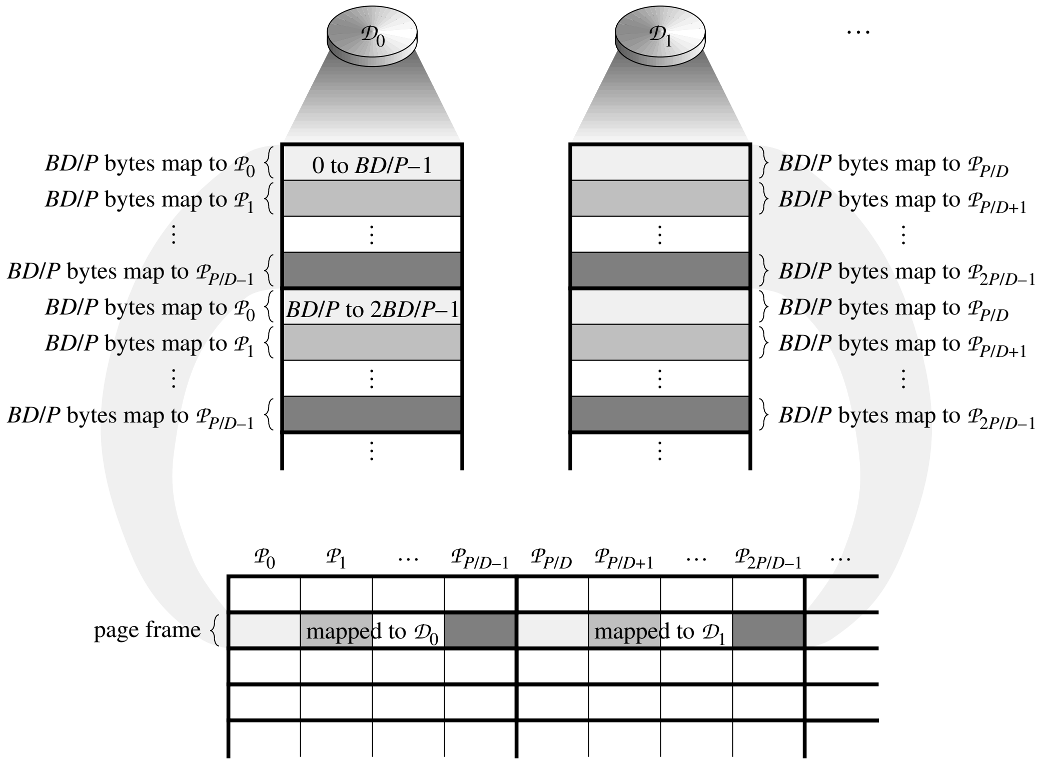

$\mathcal{D}_0$
$\mathcal{D}_1$
…
BD/P bytes map to $\mathcal{P}_0$
0 to $BD/P-1$
BD/P bytes map to $\mathcal{P}_1$
BD/P bytes map to $\mathcal{P}_{P/D-1}$
BD/P to $2BD/P-1$
BD/P bytes map to $\mathcal{P}_{P/D}$
BD/P bytes map to $\mathcal{P}_{P/D+1}$
BD/P bytes map to $\mathcal{P}_{2P/D-1}$
$\mathcal{P}_0$ $\mathcal{P}_1$ … $\mathcal{P}_{P/D-1}$ $\mathcal{P}_{P/D}$ $\mathcal{P}_{P/D+1}$ … $\mathcal{P}_{2P/D-1}$ …
page frame
mapped to $\mathcal{D}_0$
mapped to $\mathcal{D}_1$

Figure 5.1: The relationship among disks, blocks, and RAM. Each block on disk $\mathcal{D}_i$ is divided into P/D units of BD/P bytes each which map to the portion of a page frame in processors $\mathcal{P}_{iP/D}, \mathcal{P}_{iP/D+1}, \ldots, \mathcal{P}_{(i+1)P/D-1}$. Equal shading indicates blocks on disk (top) and in RAM (bottom) that map to each other. Blocks, both on disk and in RAM, have heavy borders around them. Addresses in the first BD/P bytes of each block on $\mathcal{D}_0$ identify vector addresses.

identify it by an address in the RAM of processor $\mathcal{P}_0$. The machine model associates the first BD/P bytes of each block on disk $\mathcal{D}_0$ with processor $\mathcal{P}_0$. The vector address space, therefore, is byte offsets into the first BD/P bytes of each block of disk $\mathcal{D}_0$. Because each page is a track and each track is a stripe across the parallel disk system, there are BD/P addresses per page.

Figure 5.1 shows the relationship among disks, blocks, and RAM. Each track, and hence each page, is comprised of D blocks of B bytes each. Each block on disk $\mathcal{D}_i$ is divided into P/D units of BD/P bytes each which map to the portion of a page frame in processors $\mathcal{P}_{iP/D}$, $\mathcal{P}_{iP/D+1}, \ldots, \mathcal{P}_{(i+1)P/D-1}$. The addresses in the first BD/P bytes of each block on disk $\mathcal{D}_0$ identify vector addresses.

Because the VM-DP system supports the VCODE model, there are only two types of vectors:

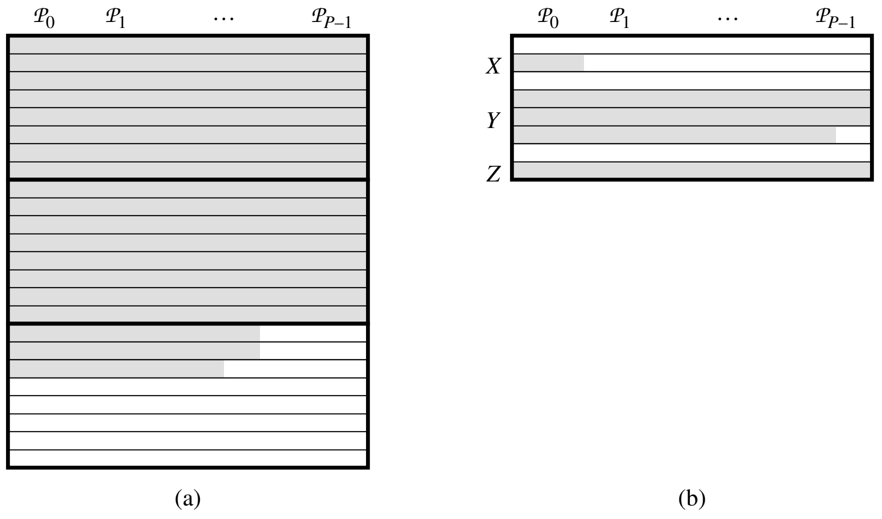


Figure 5.2: How large and small vectors are mapped to the address space. Shaded positions contain elements, and white positions are empty. **(a)** A large vector starts at the beginning of a track, occupies a set of contiguous tracks, and uses the minimum VPR in its last track. The remainder of the last track is unused. Each track is drawn with a heavy border. **(b)** A small-vector track. Each small vector is aligned to start in processor $\mathcal{P}_0$, has the minimum VPR, and may not cross a track boundary. In this example, vectors X and Z have VPRs of 1, and vector Y has a VPR of 3.

vectors of 4-byte integers and vectors of 8-byte (double precision) floats. All vector addresses are therefore multiples of 4.

Mapping of vectors to the address space

Figure 5.2(a) shows how large vectors are mapped. The address of any large vector is the address of the beginning of a track. In other words, large vectors always start at the beginning of a page. The vector occupies a contiguous set of tracks for as many tracks as it requires. All tracks except the possibly last one are full. The last track contains the remaining elements in a *load-balanced* fashion, by which we mean the following. For an assignment of elements to processors, we define the *virtual processor ratio*, or *VPR*, as the maximum number of elements assigned to any processor. If there are k elements in the last track, the VPR of the last track is $\lceil k/P \rceil$, which is the minimum possible. In particular, we map the first $\lceil k/P \rceil$ elements to processor $\mathcal{P}_0$, the next $\lceil k/P \rceil$ elements to processor $\mathcal{P}_1$, and so on, until we run out of elements. Even if only partially full, the last track of a large vector contains only the elements of this vector; no other vector uses this portion of the address space. A track that contains part of a large vector therefore contains no small vectors. In the worst case—a vector that occupies one track plus one more element—this scheme wastes less than half of the address space, and on average it wastes much less.

Small vectors are mapped to avoid wasting space yet make accessing them simple. If a track contains small vectors, it contains only small vectors and no portions of large vectors. As Figure 5.2(b) shows, small vectors are placed into small vector tracks with three restrictions. First, each vector begins in processor $\mathcal{P}_0$. Second, each vector is laid out with the minimum VPR, just like the last track of a large vector. That is, we reserve $\lceil N/P \rceil$ elements per processor for an N-element vector. Third, small vectors cannot cross track boundaries. When a small-vector track is read into RAM, we know that we have its small vectors read into RAM in their entirety.

Vector allocation and deallocation

The only part of the NESL/VCODE/CVL package that we had to modify other than the CVL functions was the VCODE module that performs vector allocation and deallocation. As in the VCODE system provided by Blelloch's group, this code runs on the front-end machine attached to the data-parallel machine, and there it maintains a data structure of all allocated and unallocated sections of the address space.

The VM-DP allocation scheme takes advantage of the quantization of vector sizes. They are quantized in two ways.

For small vectors, because they must start in processor $\mathcal{P}_0$ and their vector addresses are multiples of 4, many vectors may occupy the same amount of address space. In Figure 5.2(b), for example, the vectors X and Z each occupy only one row across the processors. Although X is a vector of only one 4-byte integer and Z is a vector of P 4-byte integers, they take the same amount of address space. In general, two small vectors with N_1 and N_2 elements and R_1 and R_2 bytes per element, respectively, occupy the same amount of address space if $R_1 \lceil N_1/P \rceil = R_2 \lceil N_2/P \rceil$. The quantization effect of dividing by P and taking the ceiling can cause vectors whose lengths differ by up to $P-1$ have the same address-space size.

For large vectors, because the last track is not shared with any other vector, we allocate them as if the last track were full. In other words, we round up large-vector sizes to the next full track. The BD-byte size of a track can be quite large, resulting in a considerable quantization effect.

Quantization helps because the VM-DP vector-allocation code uses a policy of "first exact fit but settle for first fit." To allocate a new vector, we search through all unallocated sections of the address space, looking for an exact fit of the quantized size. For example, an unallocated section of three consecutive tracks is an exact fit for any vector that needs more than two but no more than three tracks. As we search through the unallocated sections, we keep the address of the first section that fit at all; if there is no exact fit, we use this first fit. This search takes place in the front-end machine, causing no disk accesses for vectors.

Why is this strategy good? It is guaranteed to use an exact fit if one exists, thus reducing fragmentation of the address space upon allocation. The quantization increases the likelihood of an exact fit occurring. And if there is no exact fit, this strategy has the advantage of first fit, which Knuth [Knu68, pp. 435–451] finds to be a good allocation method.

Vector deallocation is performed as simply as possible. When we deallocate a vector, we merge it with any of its unallocated neighboring sections of the address space to make a maximal unallocated section. The current implementation does not attempt to compact the remaining allocated vectors in the address space or otherwise relocate them. Even if an allocation request finds no unallocated section large enough, no compaction occurs. Instead, we allocate new space on the disk; disk space is viewed by the current implementation as a limitless resource.

5.3 Implementation issues in the VM-DP paging system

This section presents some of the implementation details of the VM-DP paging system. It is by no means a complete account of the implementation but rather touches on some of the techniques we use to access vectors quickly and save disk I/Os.

Data structures

The VM-DP paging system uses two primary data structures: a page-frame table and a hash table. The page-frame table has one entry for each of the M/BD page frames in RAM. Among the information contained in each entry is the following:

- The vector address of the beginning of the page in the frame if a page is present, or a special value if the frame is empty.
- A flag indicating whether the page has been changed in RAM since it was last read.
- Pointers to the next and previous page frame in a doubly linked list of frames in most recently used (MRU) order. There is one such list in the one-space scheme, and the two-space and ω-RAM schemes each use two disjoint lists, one for each portion of RAM. The tail of each list is its LRU page.

The hash table contains an entry for each large-vector track in RAM and for each small-vector track whether or not it is in RAM. Among the information contained in each hash-table entry is the following:

- The vector address of the beginning of the track, which is also the hash key.
- The number of the page frame containing the track if it is in RAM, or a special value if the track is not in RAM.
- A count of the number of vectors currently allocated in the track.

The hash table permits fast accesses and quick determination of whether a page fault occurs. If the first address of a large-vector track is not present in the hash table, then there is a fault. For small-vector accesses, it is simple to compute the first address of the page holding a particular small vector. This address is present in the hash table, and the number of its page frame determines whether there is a fault.

The count field is not really needed for large-vector tracks, since at most one large vector can occupy a track. For small-vector tracks, however, we increment the value of this field when a vector is allocated on the track, and we decrement it when a vector is deallocated. When the count goes down to zero, the track is empty.

Reducing disk accesses

In order to reduce the number of disk accesses, we do not immediately remove a track from the page-frame and hash tables when it becomes unused. Instead we move it to the tail of the MRU list so that it becomes a likely candidate to be swapped out upon the next page fault. If it turns out that a vector is soon to be allocated on this track, it is already in RAM and does not need to be read from disk.

An even more worthwhile optimization takes advantage of the property that large-vector tracks are not shared by multiple vectors. Suppose that we are writing a large vector from beginning to end, as occurs in elementwise and scan operations. Because each track of the large vector contains no other data and it is being written, there is no need to read it from disk first. Unfortunately, this optimization cannot be used for a small vector, since it may share its track with other small vectors. Also, the disk read must occur when only a single element of a vector is being written in order avoid corrupting the other elements in the track.

5.4 Conclusions

This chapter has presented the highlights of the design of the VM-DP paging system. In particular, we have seen that the one-space paging scheme is overall the simplest and best. We also examined how VM-DP maps vectors to pages and allocates vectors, along with some key implementation details.

The remainder of this section discusses future work in paging systems for virtual memory for data-parallel computing.

Are there other paging schemes that would yield significantly better performance than the one-space scheme? We have considered two variations on the ω-RAM scheme. In one, we place an upper limit the on number of page frames for each track classification, but there are no lower limits. Moreover, the upper limits are not in force whenever there are unused page frames. The upper limits sum to more than the number of page frames in RAM. Such a scheme would allow for greater flexibility yet protect some smaller vectors from being thrown out of RAM by larger ones. The other variation is a hybrid of the ω-RAM and two-space schemes, which is essentially a three-space scheme. We allocate a small number of page frames for vectors that are more than a constant times the RAM size, and then split (perhaps unevenly) the remaining frames between small and intermediate vectors. This approach may take the best from these two schemes, or it may take the worst.

To take the first variation a step further, can we improve I/O performance with a scheme that adapts according to the observed usage pattern?

We could have used other vector-allocation strategies as well. For example, rather than "first exact fit but settle for first fit," we could use "first exact fit on a page already in RAM, settle for first exact fit, then settle for first fit on a page already in RAM, and finally settle for first fit." Would such a strategy improve performance, or would it be more trouble than it was worth?

The VM-DP system does not attempt to relocate vectors upon allocation or deallocation. Perhaps it should. If so, when should we relocate vectors? Which vectors should be relocated? Where should they move to?

The number of empirical studies one can perform on paging systems seems limitless.

Chapter 6

Conclusions

Jim Salem feels a little better now. He sees the promise that he won't have to write explicit I/O calls for much longer. And he is relieved that he won't have to become expert in how to perform vector operations when data resides on a parallel disk system, for he sees that there will be VM systems for data-parallel computing that handle these tasks for him.

This thesis has demonstrated theoretical and practical methods for performing permutations, shown how to lay out vectors for the best performance, and presented an empirical study of paging systems. Many of the ideas in this thesis affected the design and implementation of the VM-DP system, and they ought to prove valuable to anyone implementing a VM system for data-parallel computing.

But Jim Salem is not yet completely satisfied.

There is much more to be done in order to make VM for data-parallel computing truly useful to the computing community at large. The conclusions sections of Chapters 2–5 posed open problems and suggested future work. Yet, there is one important area that this thesis has not addressed.

The future of VM for data-parallel computing lies in languages and compilers as much as in algorithms. Most of the application areas listed back in Section 1.1 were scientific. VM for data-parallel computing will likely yield the greatest benefit to the scientific community if it supports Fortran and other frequently used languages.[1]

A couple of examples show the deficiencies of an intermediate language such as VCODE or Paris, in which the instructions apply one operation to a fixed number of operand vectors.

[1]Jim Salem reports, "For finite element science work I'd probably want Fortran, for graphics I'd want C, for AI and prototyping I'd want Lisp."

First, consider the dot product of two N-element vectors X and Y:

$$X \cdot Y = \sum_{i=0}^{N-1} X_i \, Y_i \ .$$

In VCODE or Paris, this computation would require a temporary vector variable, and it would be expressed along the lines of

$$\begin{aligned} &\mathit{temp} \leftarrow X * Y && \text{(elementwise multiplication)} \\ &\mathit{result} \leftarrow +\text{-REDUCE}(\mathit{temp}) && \end{aligned}$$

The problem here is that we would end up writing each track of the temporary vector *temp* in the elementwise multiplication operation, only to read it back during the reduce operation. Ideally, we would like to run code that reads in the operand vectors track by track and maintains the running sum without the need to write or read tracks of *temp*. Thus, we would save $2N/BD$ of the total $4N/BD$ parallel I/Os.

Another example takes this notion a step further. Consider the elementwise addition of several vectors:

$$Z \leftarrow A + B + C + \cdots + W + X + Y \ .$$

Rather than writing and reading tracks of temporary variables for each addition operation, we would like to read in a track of A and a track of B, add them into a track's worth of sums in RAM, read in a track of C, add it into the sums in RAM, and so on, writing out a track of Z after adding in a track of Y; we then repeat this process for the next track of each vector, and so on. If there are k operand vectors, using full temporary vector variables incurs a cost of $3(k-1)N/BD$ parallel I/Os, whereas the above method of computing the sum track by track in RAM costs only $(k+1)N/BD$ parallel I/Os.

The point of these examples is that we need to define the right set of primitive operations and invoke them effectively at compile time in order to produce efficient data-parallel code in the presence of virtual memory. Chatterjee [Cha91] took a similar approach for shared-memory multiprocessing. For virtual memory in a data-parallel environment, the stakes are even higher. The price of inefficient code is the expense of disk accesses.

Bibliography

[ACS87] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Hierarchical memory with block transfer. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 204–216, October 1987.

[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.

[BC90] Guy E. Blelloch and Siddhartha Chatterjee. VCODE: A data-parallel intermediate language. In *Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation*, pages 471–480, October 1990.

[BCD72] A. Bensoussan, C. T. Clingen, and R. C. Daley. The Multics virtual memory: Concepts and design. *Communications of the ACM*, 15(5):308–318, May 1972.

[BCK$^+$92] Guy E. Blelloch, Siddartha Chatterjee, Fritz Knabe, Jay Sipelstein, and Marco Zagha. VCODE *Reference Manual (Version 1.3)*, February 1992.

[BCSZ91] Guy E. Blelloch, Siddartha Chatterjee, Jay Sipelstein, and Marco Zagha. *CVL: A C Vector Library*, November 1991.

[BK82] Richard B. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, C-31(3):260–264, March 1982.

[Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* The MIT Press, Cambridge, Massachusetts, 1990.

[Ble92] Guy E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie Mellon University, January 1992.

[BLM$^+$91] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 3–16, July 1991.

[BM76] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications.* American Elsevier, 1976.

[BME90] Jean-Phillipe Brunet, Jill P. Mesirov, and Alan Edelman. An optimal hypercube direct N-body solver on the Connection Machine. In *Proceedings of Supercomputing '90*, pages 748–752, November 1990.

[Bre69] Norman M. Brenner. Fast Fourier transform of externally stored data. *IEEE Transactions on Audio and Electroacoustics*, AU-17(2):128–132, June 1969.

[Can90] Charles R. Cantor. Orchestrating the human genome project. *Science*, 248:49–51, 1990.

[CC90] Moon Jung Chung and Yunmo Chung. Efficient parallel logic simulation techniques for the Connection Machine. In *Proceedings of Supercomputing '90*, pages 606–614, November 1990.

[CGK$^+$88] Peter Chen, Garth Gibson, Randy H. Katz, David A. Patterson, and Martin Schulze. Two papers on RAIDs. Technical Report UCB/CSD 88/479, Computer Science Division (EECS), University of California, Berkeley, December 1988.

[Cha91] Siddhartha Chatterjee. *Compiling Data-Parallel Programs for Efficient Execution on Shared-Memory Multiprocessors.* PhD thesis, School of Computer Science, Carnegie Mellon University, October 1991. Available as Technical Report CMU-CS-91-189.

[Che91] Robert Chervin. Climate modeling with parallel vector supercomputers. In *Proceedings of Supercomputing '91*, page 677, November 1991.

[Chr91] A. T. Chronopoulos. Towards efficient parallel implementation of the CG method applied to a class of block tridiagonal linear systems. In *Proceedings of Supercomputing '91*, pages 578–587, November 1991.

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

[Cor91] Daniel Corkill. Blackboard systems. *AI Expert*, pages 40–47, September 1991.

[Cor92] Thomas H. Cormen. Fast permuting in disk arrays. In *Proceedings of the 1992 Brown/MIT Conference on Advanced Research in VLSI and Parallel Systems*, pages 58–76, 1992. Conference version is an extended abstract; full paper to appear in *Journal of Parallel and Distributed Computing.*

[CP90] Peter M. Chen and David A. Patterson. Maximizing performance in a striped disk array. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 322–331, 1990.

[Dav92] Dan Davenport. Private communication, July 1992.

[DD68] Robert C. Daley and Jack B. Dennis. Virtual memory, processes, and sharing in MULTICS. *Communications of the ACM*, 11(5):306–312, May 1968.

[Dem88] Gary Demos. Generating movie-quality animated graphics with massively parallel computers. In *Proceedings of the Symposium on the Frontiers of Massively Parallel Computation*, pages 15–20, October 1988.

[Den70] Peter J. Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–189, September 1970.

[EHJ92] Alan Edelman, Steve Heller, and S. Lennart Johnsson. Index transformation algorithms in a linear algebra framework. Technical Report LBL-31841, Lawrence Berkeley Laboratory, 1992.

[EHMN90] Matthew Evett, James Hendler, Ambujashka Mahanti, and Dana Nau. PRA*: A memory-limited heuristic search procedure for the Connection Machine. In *Proceedings of the Symposium on the Frontiers of Massively Parallel Computation*, pages 145–149, October 1990.

[Flo72] Robert W. Floyd. Permuting information in idealized two-level storage. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, pages 105–109. Plenum Press, 1972.

[Fra76] Donald Fraser. Array permutation by index-digit permutation. *Journal of the ACM*, 23(2):298–309, April 1976.

[GHK$^+$89] Garth A. Gibson, Lisa Hellerstein, Richard M. Karp, Randy H. Katz, and David A. Patterson. Failure correction techniques for large disk arrays. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 123–132, April 1989.

[Gib92] Garth A. Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage.* The MIT Press, Cambridge, Massachusetts, 1992. To appear. Also Technical Report UCB/CSD 91/613, Computer Science Division (EECS), University of California, Berkeley, May 1991.

[GK82] Harold N. Gabow and Oded Kariv. Algorithms for edge coloring bipartite graphs and multigraphs. *SIAM Journal on Computing*, 11(1):117–129, February 1982.

[Has87] Brosl Hasslacher. Discrete fluids. *Los Alamos Science*, Special Issue(15):175–217, 1987.

[Her75] I. N. Herstein. *Topics in Algebra.* Xerox College Publishing, second edition, 1975.

[Hig92] Peter Highnam. Private communication, March 1992.

[JH91] S. Lennart Johnsson and Ching-Tien Ho. Generalized shuffle permutations on boolean cubes. Technical Report TR-04-91, Harvard University Center for Research in Computing Technology, February 1991.

[Kau88] Tuomo Kauranne. An introduction to parallel processing in meteorology. In G.-R. Hoffmann and D. K. Maretis, editors, *The Dawn of Massively Parallel Processing in Meteorology*, pages 3–20. Springer-Verlag, 1988.

[KNPF87] Michelle Y. Kim, Anil Nigam, George Paul, and Robert H. Flynn. Disk interleaving and very large fast Fourier transforms. *International Journal of Supercomputer Applications*, 1(3):75–96, 1987.

[Knu68] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison-Wesley, Reading, MA, 1968.

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

[Koc90] Peter Kochevar. Too many cooks don't spoil the broth: Light simulation on massively parallel computers. In *Proceedings of the Symposium on the Frontiers of Massively Parallel Computation*, pages 100–109, October 1990.

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

[LF80] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, October 1980.

[LK91] Edward K. Lee and Randy H. Katz. Performance consequences of parity placement in disk arrays. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 190–199, April 1991.

[Mac92] *Macworld*, September 1992.

[Man92] Nashat Mansour. Private communication, July 1992.

[MC69] A. C. McKellar and E. G. Coffman, Jr. Organizing matrices and matrix operations for paged memory systems. *Communications of the ACM*, 12(3):153–165, March 1969.

[MF92] Nashat Mansour and Geoffrey C. Fox. Parallel physical optimization algorithms for allocating data to multicomputer nodes. Unpublished manuscript, 1992.

[MK91] Ethan L. Miller and Randy H. Katz. Input/output behavior of supercomputing applications. In *Proceedings of Supercomputing '91*, pages 567–576, November 1991.

[MMF$^+$91] Carlos R. Mechoso, Chung-Chun Ma, John D. Farrara, Joseph A. Spahr, and Reagan W. Moore. Distribution of a climate model across high-speed networks. In *Proceedings of Supercomputing '91*, pages 253–260, November 1991.

[MS91] Jacek Myczkowski and Guy Steele. Seismic modeling at 14 gigaflops on the Connection Machine. In *Proceedings of Supercomputing '91*, pages 316–326, November 1991.

[MW91] Tom Myers and Elizabeth Williams. Mass storage requirements in the intelligence community. In *Proceedings of Supercomputing '91*, pages 878–889, November 1991.

[NGHG$^+$91] Hugh Nicholas, Grace Giras, Vasiliki Hartonas-Garmhausen, Michael Kopko, Christopher Maher, and Alexander Ropelewski. Distributing the comparison of DNA and protein sequences across heterogeneous supercomputers. In *Proceedings of Supercomputing '91*, pages 139–146, November 1991.

[NM87] Alan Norton and Evelyn Melton. A class of boolean linear transformations for conflict-free power-of-two stride access. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 247–254, August 1987.

[NS81] David Nassimi and Sartaj Sahni. A self-routing Benes network and parallel permutation algorithms. *IEEE Transactions on Computers*, C-30(5):332–340, May 1981.

[NS82] David Nassimi and Sartaj Sahni. Optimal BPC permutations on a cube connected SIMD computer. *IEEE Transactions on Computers*, C-31(4):338–341, April 1982.

[NV90] Mark H. Nodine and Jeffrey Scott Vitter. Greed sort: An optimal external sorting algorithm for multiple disks. Technical Report CS-90-04, Department of Computer Science, Brown University, February 1990.

[NV91] Mark H. Nodine and Jeffrey Scott Vitter. Large-scale sorting in parallel memories. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 29–39, July 1991.

[NV92] Mark H. Nodine and Jeffrey Scott Vitter. Optimal deterministic sorting on parallel disks. Technical Report CS-92-08, Department of Computer Science, Brown University, 1992.

[PF91] C. Pommerell and W. Fichtner. PILS: An iterative linear solver package for ill-conditioned systems. In *Proceedings of Supercomputing '91*, pages 588–599, November 1991.

[PFTV89] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in Pascal: The Art of Scientific Computing.* Cambridge University Press, 1989.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM International Conference on Management of Data (SIGMOD)*, pages 109–116, June 1988.

[PH90] David A. Patterson and John L. Hennessy. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[Pin82] Ron Yair Pinter. *The Impact of Layer Assignment Methods on Layout Algorithms for Integrated Circuits.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, August 1982. Available as Technical Report MIT/LCS/TR-291.

[PMG91] Steve Plimpton, Gary Mastin, and Dennis Ghiglia. Synthetic aperture radar image processing on parallel supercomputers. In *Proceedings of Supercomputing '91*, pages 446–452, November 1991.

[Pre92a] John K. Prentice. Private communication, July 1992.

[Pre92b] John K. Prentice. *A Quantum Mechanical Theory for the Scattering of Low Energy Atoms From Incommensurate Crystal Surface Layers.* PhD thesis, Department of Physics and Astronomy, University of New Mexico, July 1992.

[PS85] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.

[Sal92] Jim Salem. Private communication, January 1992.

[SBGM91] James A. Sethian, Jean-Phillipe Brunet, Adam Greenberg, and Jill P. Mesirov. Computing turbulent flow in complex geometries on a massively parallel processor. In *Proceedings of Supercomputing '91*, pages 230–241, November 1991.

[SDM91] R. D. Smith, J. K. Dukowicz, and R. C. Malone. Ocean modeling on the Connection Machine. In *Proceedings of Supercomputing '91*, page 679, November 1991.

[Sea92] Sandra H. Seale. Private communication, July 1992.

[Sin67] Richard C. Singleton. A method for computing the fast Fourier transform with auxiliary memory and limited high-speed storage. *IEEE Transactions on Audio and Electroacoustics*, AU-15(2):91–98, June 1967.

[Smi92] Stephen J. Smith. Private communication, July 1992.

[ST85] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, February 1985.

[Thi89] Thinking Machines Corporation, Cambridge, Massachusetts. *Paris Reference Manual*, February 1989.

[Til90] James C. Tilton. Porting an iterative parallel region growing algorithm to the MPP to the Maspar MP-1. In *Proceedings of the Symposium on the Frontiers of Massively Parallel Computation*, pages 170–173, October 1990.

[TMB91] Pablo Tamayo, Jill P. Mesirov, and Bruce M. Boghosian. Parallel approaches to short range molecular dynamics simulations. In *Proceedings of Supercomputing '91*, pages 462–470, November 1991.

[VS90a] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. Technical Report CS-90-21, Department of Computer Science, Brown University, September 1990.

[VS90b] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Optimal disk I/O with parallel block transfer. In *Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing*, pages 159–169, May 1990.

[Wat90] James D. Watson. The human genome project: Past, present, and future. *Science*, 248:44–49, 1990.

[Wil91] Dennis E. Willen. Exploration geophysics, parallel computing, and reality. In *Proceedings of Supercomputing '91*, page 540, November 1991.

[YZ91] L. C. Young and S. E. Zarantonello. High performance vector processing in reservoir simulation. In *Proceedings of Supercomputing '91*, pages 304–315, November 1991.