Dartmouth College

Dartmouth Digital Commons

Computer Science Technical Reports

Computer Science

1-1-1993

Vector Layout in Virtual-Memory Systems for Data-Parallel Computing

Thomas H. Cormen
Dartmouth College

Follow this and additional works at: https://digitalcommons.dartmouth.edu/cs_tr

Part of the Computer Sciences Commons

Vector Layout in Virtual-Memory Systems
for Data-Parallel Computing

Thomas H. Cormen
Department of Mathematics and Computer Science
Dartmouth College

Abstract

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. We present 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, and we analyze the effect of the band size on the major classes of data-parallel operations. We find that although the best band size varies among the operations, choosing fairly small band sizes—at most a track—works well in general.

1   Introduction

Some applications that are well-suited to data-parallel computing, such as three-dimensional finite-element problems like seismic modeling, must sometimes process more data than will fit in the primary memory of even the largest parallel computer. Consequently, the information may have to reside on a parallel disk system, and then application programmers may have to explicitly invoke the disk I/O from within the application.

With the advent of virtual memory in sequential machines many years ago, application programmers were freed from explicitly calling the disk I/O operations, letting the system perform the calls instead. (For a good historical and technical background, see Denning's survey article [Den70] on virtual memory.) But there has been little work done on virtual-memory systems for data-parallel computing.

Note that by "virtual memory" we mean "providing the illusion of a large physical memory." One must resist the temptation to confuse virtual memory with implementations of it, such as demand paging. Although the term "extended memory" might lead to less confusion, the term "virtual memory" is appropriate in the context of data-parallel computing because we wish to provide transparent mechanisms for data-parallel programs to access more data than will fit in primary memory.

The VM-DP project at MIT is a first cut at understanding some of the issues that arise in the design and implementation of virtual-memory systems for data-parallel computing [Cor92]. In the VM-DP system, source programs are written in Nesl, a strongly-typed, applicative, data-parallel language designed by Blelloch [Ble92]. The Nesl compiler produces Vcode, a stack-based
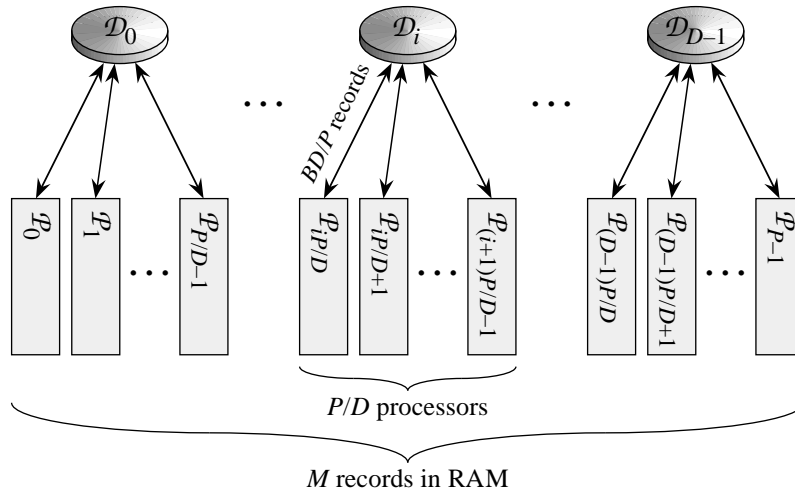
Figure 1: The virtual-memory data-parallel machine model. The P processors $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{P-1}$ can hold a total of M records in RAM, and an array of D disks $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{D-1}$ is connected to the processors. Records are transferred between disk $\mathcal{D}_i$ and the RAM of processors $\mathcal{P}_{iP/D}, \mathcal{P}_{iP/D+1}, \ldots, \mathcal{P}_{(i+1)P/D-1}$, with disk I/O occurring in blocks of B records per disk.

intermediate data-parallel language [BC90, BCK$^+$92]. The VCODE interpreter makes calls to CVL [BCSZ91], its interface to the machine. The VM-DP project is a complete implementation of CVL along with some slight modifications to the VCODE interpreter to perform a simulation of a data-parallel machine with virtual memory. The VM-DP code comprises about 7500 lines of C and runs under Unix on a workstation.

This paper investigates some of the issues of data layout that arise in a system such as VM-DP. In particular, we examine how the layout of vectors affects the performance of data-parallel operations.[1] By studying the effect of vector layout on performance, we can develop a guideline for how to lay out vectors. We start by presenting the model of a data-parallel machine with virtual memory assumed by the VM-DP system.

Machine model

We use the model of a data-parallel machine with a parallel disk system shown in Figure 1. 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 random-access memory (RAM), each processor holding up to M/P records in its RAM. We purposely leave the notion of what constitutes a record as unspecified, but for now, think of each record as containing one datum, say an integer. (We are concerned with I/O for data only, not code.) Connected to the processors is an array of D disks $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{D-1}$. We assume that $D \leq P$, so that each disk corresponds to one or more processors.[2] In particular, for $i = 0, 1, \ldots, D - 1$, records can be transferred between disk $\mathcal{D}_i$ and the RAM of the P/D processors $\mathcal{P}_{iP/D}, \mathcal{P}_{iP/D+1}, \ldots, \mathcal{P}_{(i+1)P/D-1}$. 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, BD/P such records for each processor connected to $\mathcal{D}_i$. Moreover, all blocks start on

---

[1]In a similar vein, McKellar and Coffman [MC69] studied how layout of matrices affects the performance of matrix operations in serial machines.

[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.

2

multiples of B records on each disk. We assume that the parameters P, B, and D are all powers of 2.

Most of the parallel I/O operations that we use in this paper are "striped."[3] 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*. Each striped I/O operation reads or writes a track of BD records. To ensure that there is sufficient RAM to accommodate a parallel I/O, we require that $BD \leq M$. Generally, when we perform striped I/O, we partition RAM into M/BD *track frames*. Each track frame is a set of BD consecutive locations in RAM which can be used, among other purposes, as an I/O buffer.

A *vector* is a one-dimensional array of records. We call each record in a vector an *element*, and we use N to denote the number of records, or *length*, of a vector. We index the elements of vector X as $X_0, X_1, \ldots, X_{N-1}$. This paper addresses how to establish a correspondence between the elements of a vector and locations in RAM and on disk. We require that for each vector X, element X_0 maps to the RAM of processor $\mathcal{P}_0$.

Data-parallel instructions operate on one or more vectors at a time. To execute a data-parallel instruction in a virtual-memory environment, vectors (especially large ones), which reside on the disk array, are brought into RAM for processing as needed, a page at a time. Some operations access pages through a demand paging system; others may control disk I/O on their own. In this paper, we use a track, containing BD records, as our page size.

We will focus on vectors that occupy several pages. 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.

The remainder of this paper is organized as follows. Section 2 presents how to lay out vectors in terms of a parameterized band size. Section 3 summarizes the three major classes of data-parallel operations considered in Sections 4–7, in which we determine the effect of the band size on the operation performance. We show how to optimize band sizes where possible. Section 4 discusses the effect of band size on the performance of several permutation classes. Section 5 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 6 analyzes the scan formula to derive the optimal band size. Section 7 looks at the effect of band size on elementwise operations. Finally, Section 8 reviews the results of the previous sections to conclude that small band sizes work well in general.

A note about the results

The results in this paper are based on reducing constant factors in upper bounds. Because disk accesses typically take on the order of a few milliseconds and virtual-memory systems for data-parallel computing might deal with several terabytes of data (or even more), it is important to squeeze out constant factors in I/O counts whenever possible.

It is not yet known whether the constants obtained in this paper are the best possible. The algorithms we consider in this paper are asymptotically optimal. That is, they are known to be optimal to within a constant factor. For most of problems we consider, the only exact lower bounds known are the obvious ones based on reading each input record and writing each output record once. All other known lower bounds are only asymptotic.

---

[3]Section 4 briefly considers BPC permutations, which [Cor93] performs using "independent" parallel I/Os: the accessed disk blocks may reside at any location as long as only one block per disk is accessed at a time.

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

Figure 2: 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.

2   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.

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 2 shows an example of banded layout for $P = 8$ and $\beta = 32$. Each row in Figure 2 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. 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. Overall, there are N/P rows, N/β bands, and β/P rows per band.

Particular banded layouts

The specific choice of the band size β determines the exact layout of a vector. Three particular choices yield familar vector layouts: row-major, column-major, and the layout style defined by Vitter and Shriver [VS90, VS92]. 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 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 3. 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

4

$\mathcal{P}_0$	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$	$\mathcal{P}_4$	$\mathcal{P}_5$	$\mathcal{P}_6$	$\mathcal{P}_7$	
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 3: 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	} band
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: Column-major layout of a vector, shown for $P = 8$ and $N = 64$ elements in the vector.

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 6, 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. 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 can fit in RAM because, as we shall see in Section 6, 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*[4] uses $\beta = BD$. That is, each band is exactly one track, and there are N/BD bands. Figure 2 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 processor number. We just saw examples of these mappings

---

[4]So called because it was proposed for the parallel-disk algorithms of Vitter and Shriver [VS90, VS92].
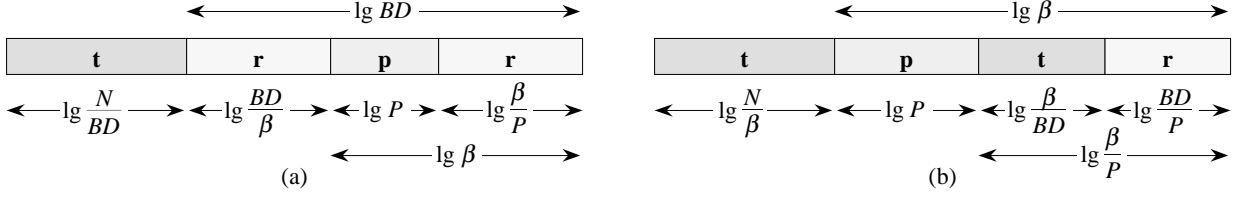
Figure 5: How element indices map to track numbers (bits labeled "t"), processor numbers ("p"), and row numbers within tracks ("r"). Least significant bits are on the right. **(a)** The scheme for $\beta \leq BD$. **(b)** The scheme for $\beta \geq BD$.

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 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 5(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 $\left\lfloor \frac{i \bmod \beta}{\beta/P} \right\rfloor = \left\lfloor \frac{(i \bmod \beta)P}{\beta} \right\rfloor$.

- 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 5(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 $\left\lfloor \frac{i \bmod (\beta/P)}{BD/P} \right\rfloor + \left\lfloor \frac{i}{\beta} \right\rfloor (\beta/BD) = \left\lfloor \frac{(i \bmod (\beta/P))P}{BD} \right\rfloor + \left\lfloor \frac{i}{\beta} \right\rfloor (\beta/BD)$.

- 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 Vitter and Shriver [VS90, VS92], Nodine and Vitter [NV90, NV91, NV92], Cormen [Cor92, Cor93], and Cormen and Wisniewski [CW93].

3   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 4–7 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 7 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 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*, which can be specified more compactly. Many of these classes can be performed faster than general permutations [Cor92, Cor93, CW93]. Section 4 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[5] 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

---

[5]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.

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 5 presents a method for performing scans that works well for all band sizes. Section 6 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 5 briefly studies reduce operations.

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 paper.

4 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

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

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

If we view each individual mapping in a monotonic route as one of the arrows in Figure 6, then no arrows cross.

8
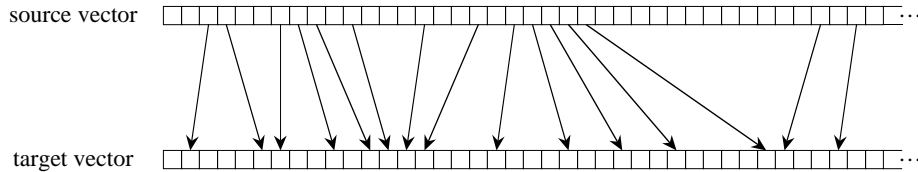
Figure 6: A monotonic route, with each element mapping indicated by an arrow. No arrows cross in a monotonic route. If the band size is no greater than half the RAM size, we can perform any monotonic route by making just one pass over the source and target vectors. Otherwise, we need to make more passes.

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 $\pi : \{0, 1, \ldots, \lg N - 1\} \overset{\text{1-1}}{\to} \{0, 1, \ldots, \lg N - 1\}$. We then complement a fixed subset of the bits. Cormen [Cor93] 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, [Cor92] shows how to compute another bit permutation $\widehat{\pi} : \{0, 1, \ldots, \lg N - 1\} \overset{\text{1-1}}{\to} \{0, 1, \ldots, \lg N - 1\}$ 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. We won't go into the details here, but for some bit permutations π and $\widehat{\pi}$, the algorithm performs π with fewer disk I/Os than for $\widehat{\pi}^{-1} \circ \pi \circ \widehat{\pi}$, and for other bit permutations π and $\widehat{\pi}$, the reverse is true.

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. We shall show that regardless of the number of grid dimensions, we

can perform these permutations efficiently. The performance is good no matter what the band size, but it is best for band sizes no greater than the track size, i.e., for $\beta \leq BD$.

Many applications use data that is organized into multidimensional grids, or meshes. A class of permutation commonly performed in d-dimensional grids 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 grid 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 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 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 7: 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"

10
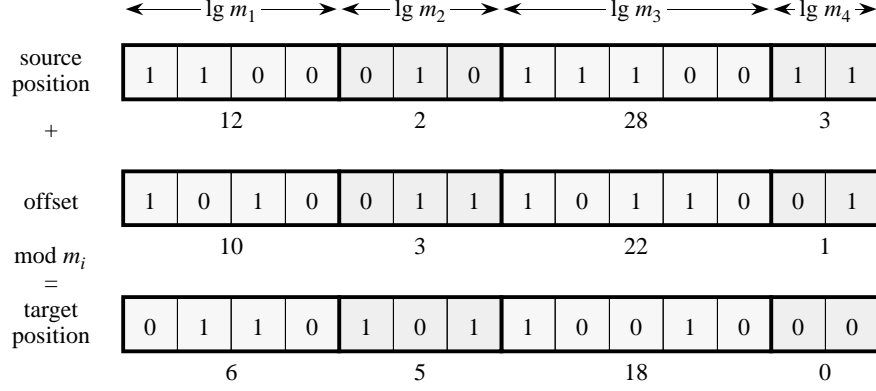
Figure 7: Partitioning the bits of an element's index to determine its grid position. The least significant bits are on the right. This example shows $d = 4$ dimensions, with $m_1 = 16$, $m_2 = 8$, $m_3 = 32$, and $m_4 = 4$. For each dimension i, a group of $\lg m_i$ bits determines the position in dimension 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.

that we add to each source position to compute the corresponding target position. Figure 7 shows that we treat the $\lg m_i$ bits of each dimension i 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 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*

$$\left\lfloor \frac{x+c}{2^r} \right\rfloor \bmod 2^s = \left\lfloor \frac{y+c}{2^r} \right\rfloor \bmod 2^s , \quad or$$

$$\left\lfloor \frac{x+c}{2^r} \right\rfloor \bmod 2^s = \left(\left\lfloor \frac{y+c}{2^r} \right\rfloor - 1 \right) \bmod 2^s .$$

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

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

11

Because $0 \le x' \le y' < 2^r$, we have that $0 \le y' - x' < 2^r$, which in turn implies that either

$$\left\lfloor \frac{x' + c'}{2^r} \right\rfloor = \left\lfloor \frac{y' + c'}{2^r} \right\rfloor \qquad \text{or} \qquad \left\lfloor \frac{x' + c'}{2^r} \right\rfloor = \left\lfloor \frac{y' + c'}{2^r} \right\rfloor - 1 \; .$$

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 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 $x + c$ and $y + c$, or the value for $y + c$ is one greater than the value for $x + c$.

We are now ready to prove that each source-vector track maps to few target-vector tracks.

Lemma 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 $\beta \le BD$, 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 5 and 7.

We first consider the case for $\beta \le BD$. As Figures 5(a) and 8(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 $x \le y$. 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 2, with $r = \lg BD$, $s = \lg(N/BD)$, and $c = o$. By Lemma 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

12
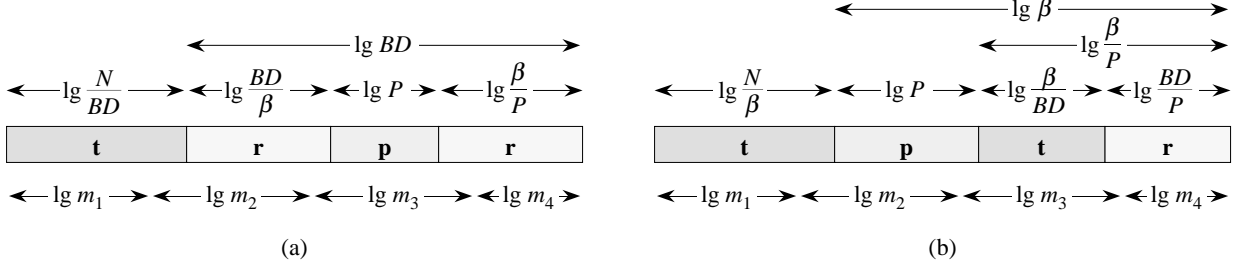
Figure 8: Cases in the proof of Lemma 3. **(a)** When $\beta \leq BD$, the track number is in bits $\lg(N/BD)$ through $\lg N - 1$. By Lemma 2, no matter what offset o we add to the source index, the resulting track number is one of at most 2 values. **(b)** When $\beta > BD$, the track number is in bit positions $\lg(BD/P)$ through $\lg(\beta/P) - 1$ and $\lg \beta$ through $\lg N - 1$. By Lemma 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.

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 5(b) and 8(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 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 3 is that the track-number bits in Figures 5 and 8 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 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 *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 1 and 3. If $\beta \leq BD$, we apply Lemma 1 with $k = 2$. If $\beta > BD$, we apply Lemma 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.

5   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 6 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.

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 9 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 9(a). We do the following:

1. Figure 9(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 9(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 9(d):
   Prepend this set of P values to the band as an initial row and then scan down the columns.

14

$s = 100$

P_0	P_1	P_2	P_3
1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

(a)

P_0	P_1	P_2	P_3
1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

column sums

10	26	42	58

(b)

P_0	P_1	P_2	P_3
10	26	42	58

physical scan

0	10	36	78

add in previous band sum s

100	110	136	178

(c)

P_0	P_1	P_2	P_3
100	110	136	178
1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

column scan →

P_0	P_1	P_2	P_3
100	110	136	178
101	115	145	191
103	121	155	205
106	128	166	220

(d)

P_0	P_1	P_2	P_3
100	110	136	178
101	115	145	191
103	121	155	205
106	128	166	220

new $s = 236$

(e)
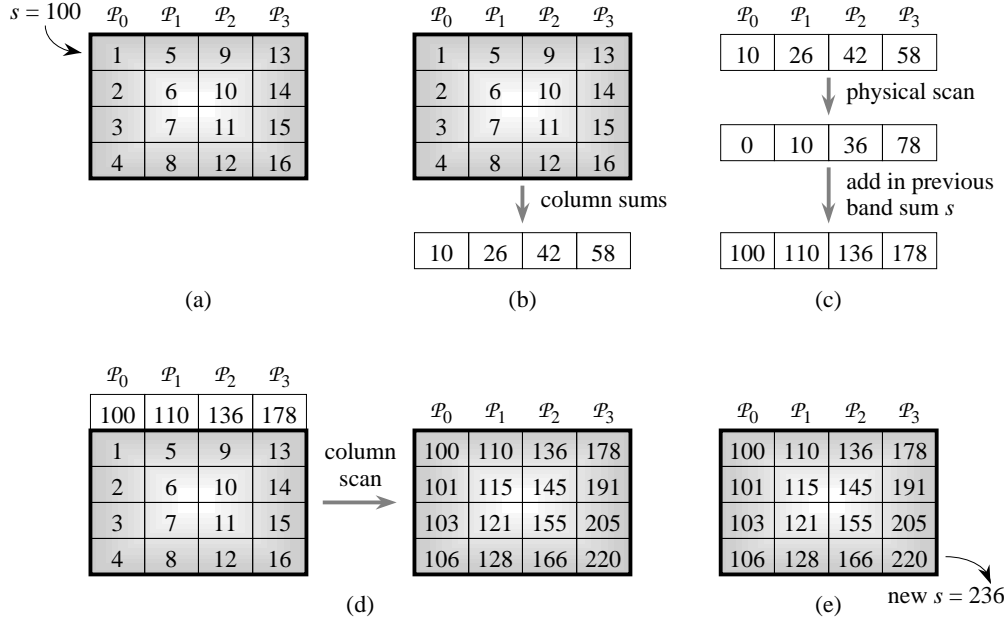
Figure 9: Processing a band when scanning. **(a)** The sum s of all prior bands is 100. **(b)** Reduce down the columns. **(c)** Perform a physical scan on the column sums and add s into the scan of the column sums. **(d)** Prepend the row produced in (c) to the band as an initial row, and then scan down the columns. **(e)** Compute a new sum s of bands for use by the next band.

4. Figure 9(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.

15

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 6 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 6. 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 $\beta \le F$, 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 $\beta \le F$, 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 $\beta \le F$, and if $\beta > F$ it is

$$
\begin{aligned}
\left(\frac{2\beta}{BD} - \frac{F}{BD} \right) \frac{N}{\beta} \; IO \;\; &= \;\; \left(2 - \frac{F}{\beta} \right) \frac{N}{BD} \; IO \\
&> \;\; \frac{N}{BD} \; IO \; .
\end{aligned}
$$

17

4. Creating the new sum s requires no additional I/O.

Adding up these costs, we get a total I/O time of

$$T_{\mathrm{I/O}}(\beta, F) = \begin{cases} 2 \dfrac{N}{BD} \, IO & \text{if } \beta \leq F \ , \\[3ex] \left(3 - \dfrac{F}{\beta}\right) \dfrac{N}{BD} \, IO & \text{if } \beta > F \ , \end{cases}$$

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:

$$T_{\mathrm{scan}}(\beta, F) = T_{\mathrm{processing}}(\beta) + T_{\mathrm{I/O}}(\beta, F) + T_{\mathrm{buffer}}(F) \ . \tag{1}$$

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 6 shows how to choose optimal values for the band size β and the I/O buffer size F under cost function (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.

6   Choosing optimal band and I/O buffer sizes for scans

In this section, we analyze the cost function, equation (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 $0 \leq \alpha \leq 1$ 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.

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 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

$$
T_{\text{scan}}(\beta, F) = \begin{cases} 2\left(\dfrac{N}{P} - 1\right) A + \dfrac{N}{\beta} S + 2\dfrac{N}{BD} IO & \text{if } \beta \leq F , \\[2ex] 2\left(\dfrac{N}{P} - 1\right) A + \dfrac{N}{\beta} S + \left(3 - \dfrac{F}{\beta}\right) \dfrac{N}{BD} IO & \text{if } \beta \geq F . \end{cases}
$$

Figure 10 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 10, for $\beta \leq F$ and for $\beta \geq F$, 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

$$
\frac{\partial T_{\text{scan}}(\beta, F)}{\partial\beta} = \begin{cases} -\dfrac{NS}{\beta^2} & \text{if } \beta \leq F , \\[2ex] -\dfrac{N}{\beta^2}\left(\dfrac{F}{BD} IO - S\right) & \text{if } \beta \geq F . \end{cases} \tag{2}
$$

Because $\beta \geq P > 0$, this derivative is negative whenever $\beta \leq F$. Moreover, because $IO > S$ and $F \geq BD$, the coefficient $((F/BD) IO - S)$ is positive, and so the derivative (2) is negative whenever $\beta \geq F$ 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 $\beta \leq F$. In this case, $T_{\text{scan}}(\beta, F)$ is a decreasing
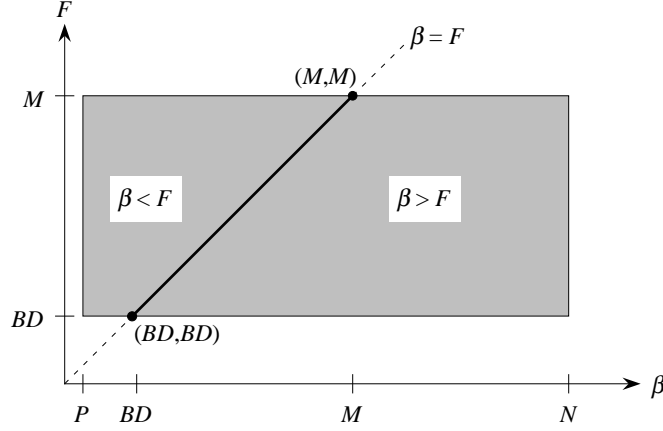
Figure 10: The domain of the band size β and the I/O buffer size F for the total scan cost $T_{\text{scan}}(\beta, F)$. The line $\beta = F$ divides the two cases of the cost function.

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 $\beta \geq F$. 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 $BD \leq \beta = F \leq M$. 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 $BD \leq M$ 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 $BD \leq M$, 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 $BD \leq M$, this value is never less than $T_{\text{scan}}(M, M)$.

- On the boundary marked by $BD \leq \beta = F \leq M$, 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 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 $0 \leq \alpha \leq 1$. 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 6 *If $IO > S$ and the number of penalty tracks is given by $\phi(F) = \alpha F/BD$, where $0 \leq \alpha \leq 1$, then $T_{\mathrm{scan}}(\beta, F)$ is minimized only if $\beta = F$. Furthermore, if we define*

$$t^* = \sqrt{\frac{NBD}{\alpha} \frac{S}{IO}} ,$$

then the following hold:

1. *If $BD \leq t^* \leq M$, then $T_{\mathrm{scan}}(\beta, F)$ 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 $t^* \leq BD$, then $T_{\mathrm{scan}}(\beta, F)$ is minimized by choosing $\beta = F = BD$.*

3. *If $t^* \geq M$, then $T_{\mathrm{scan}}(\beta, F)$ is minimized by choosing $\beta = F = M$.*

Proof: When $\phi(F) = \alpha F/BD$, we have

$$T_{\mathrm{scan}}(\beta, F) = \begin{cases} 2\left(\dfrac{N}{P} - 1\right) A + \dfrac{N}{\beta} S + 2\dfrac{N}{BD} IO + \dfrac{\alpha F}{BD} IO & \text{if } \beta \leq F , \\[3ex] 2\left(\dfrac{N}{P} - 1\right) A + \dfrac{N}{\beta} S + \left(3 - \dfrac{F}{\beta}\right) \dfrac{N}{BD} IO + \dfrac{\alpha F}{BD} IO & \text{if } \beta \geq F . \end{cases}$$

As in the proof of Theorem 5, Figure 10 shows the domain of the function $T_{\mathrm{scan}}(\beta, F)$, 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 $\partial T_{\mathrm{scan}}(\beta, F)/\partial \beta$ is again given by equation (2). As in Theorem 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 $\beta \leq F$. As in Theorem 5, $\partial T_{\mathrm{scan}}(\beta, F)/\partial \beta < 0$, 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 $\partial T_{\mathrm{scan}}(\beta, F)/\partial F = (\alpha/BD) IO$, which equals 0 only if $\alpha = 0$; in this case, $T_{\mathrm{scan}}(\beta, F)$ does not depend on F in the region $\beta \leq F$, and because $T_{\mathrm{scan}}(\beta, F)$ 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 $BD \leq \beta = F \leq M$, which we examine one by one.

- $T_{\mathrm{scan}}(BD, BD) = 2(N/P - 1)A + (N/BD)S + 2(N/BD)\ IO + \alpha\ IO$.

- $T_{\mathrm{scan}}(M, M) = 2(N/P-1)A + (N/M)S + 2(N/BD)\ IO + (\alpha M/BD)\ IO$. Simple manipulation shows that this value is less than $T_{\mathrm{scan}}(BD, BD)$ if and only if $IO/S < N/\alpha M$.

- $T_{\mathrm{scan}}(P, BD) = 2(N/P-1)A + (N/P)S + 2(N/BD)\ IO + \alpha\ IO$. Because $BD \geq P$, this value is never less than $T_{\mathrm{scan}}(BD, BD)$.

- $T_{\mathrm{scan}}(P, M) = 2(N/P - 1)A + (N/P)S + 2(N/BD)\ IO + (\alpha M/BD)\ IO$. Because $M \geq P$, this value is never less than $T_{\mathrm{scan}}(BD, BD)$.

- On the boundary marked by $BD \leq \beta = F \leq M$, we parameterize the function and simplify it as
$$T_{\mathrm{scan}}(t) = \frac{a}{t} + bt + c\ ,$$
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:
$$\frac{d\, T_{\mathrm{scan}}(t)}{dt} = -\frac{a}{t^2} + b\ .$$
If there is a minimum along this boundary, it occurs where $d\, T_{\mathrm{scan}}(t)/dt = 0$, or at
$$t^* = \sqrt{\frac{a}{b}}\ . \tag{3}$$
This extreme point is in fact a local minimum of $T_{\mathrm{scan}}(t)$, since its second derivative is
$$\frac{d^2\, T_{\mathrm{scan}}(t)}{dt^2} = \frac{a}{t^3}\ ,$$
which is positive for all positive values of t. Observe that at the point t^*, the terms a/t and bt of $T_{\mathrm{scan}}(t)$ 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 (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 $T_{\mathrm{scan}}(t)$ 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 $BD \leq t^* \leq M$ 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 $T_{\mathrm{scan}}(t)$. This completes the analysis of the boundaries and corners of the region $\beta \leq F$.

Now we analyze the boundaries and corners of $\beta \geq F$. 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 $\partial T_{\mathrm{scan}}(\beta, F)/\partial \beta < 0$. For the right boundary, we observe that
$$\frac{\partial T_{\mathrm{scan}}(\beta, F)}{\partial F} = \left(\frac{\alpha}{BD} - \frac{N}{\beta BD}\right) IO\ ,$$

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 $BD \leq M$.

By exhaustive analysis, therefore, we have proven the theorem. ∎

Where does the strange formula for t^* in the statement of Theorem 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 5 is a special case of Theorem 6 in which $\alpha = 0$. We have $t^* = \infty > M$, and so $T_{\text{scan}}(\beta, F)$ is minimized by choosing $\beta = F = M$.

7 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.
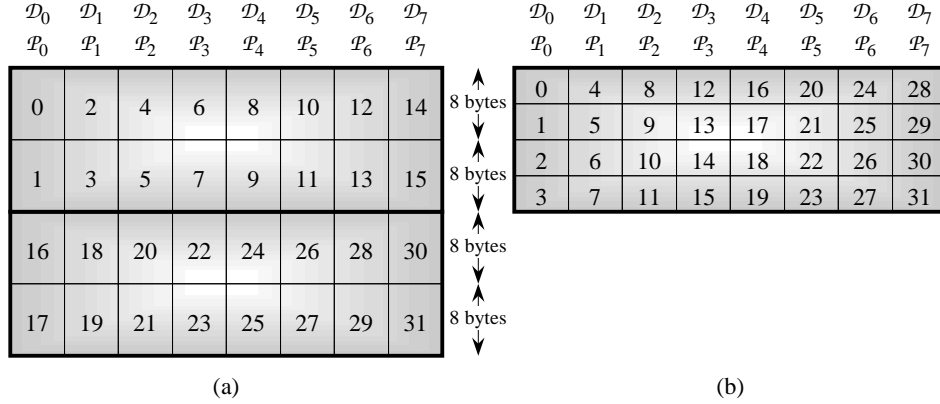
Logical vs. physical sizes

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.

$\mathcal{D}_0$ / $\mathcal{P}_0$	$\mathcal{D}_1$ / $\mathcal{P}_1$	$\mathcal{D}_2$ / $\mathcal{P}_2$	$\mathcal{D}_3$ / $\mathcal{P}_3$	$\mathcal{D}_4$ / $\mathcal{P}_4$	$\mathcal{D}_5$ / $\mathcal{P}_5$	$\mathcal{D}_6$ / $\mathcal{P}_6$	$\mathcal{D}_7$ / $\mathcal{P}_7$
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 per band segment)

(a)

$\mathcal{D}_0$ / $\mathcal{P}_0$	$\mathcal{D}_1$ / $\mathcal{P}_1$	$\mathcal{D}_2$ / $\mathcal{P}_2$	$\mathcal{D}_3$ / $\mathcal{P}_3$	$\mathcal{D}_4$ / $\mathcal{P}_4$	$\mathcal{D}_5$ / $\mathcal{P}_5$	$\mathcal{D}_6$ / $\mathcal{P}_6$	$\mathcal{D}_7$ / $\mathcal{P}_7$
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 11: 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 $\beta_4 = 32$.

$\mathcal{P}_0$	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$	$\mathcal{P}_4$	$\mathcal{P}_5$	$\mathcal{P}_6$	$\mathcal{P}_7$
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 per band segment)

(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	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

(b)
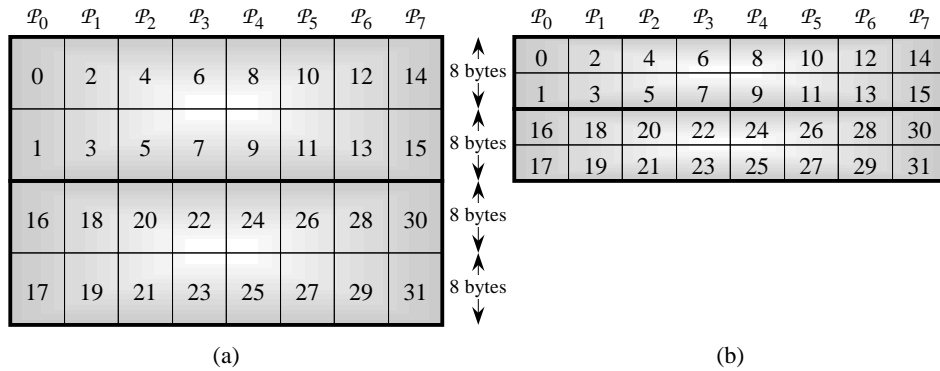
Figure 12: How the VM-DP system maps vectors to $P = 8$ processors with a physical track size of 128 bytes. **(a)** A 32-element vector of 8-byte floats. The band size is $\beta = 128/8 = 16$. **(b)** A 32-element vector of 4-byte integers, also mapped with band size $\beta = 16$. Corresponding elements of either type of vector map to the same processor.

The 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.

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 11 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 12 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.

8 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. Band sizes a track or smaller seem to be a good choice in general.

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 paper 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, for which many special permutations can be performed asymptotically faster than general permutations.

Acknowledgments

References

[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.

[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.

[Cor92]   Thomas H. Cormen. *Virtual Memory for Data-Parallel Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992. Available as Technical Report MIT/LCS/TR-559.

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

[CW93]   Thomas H. Cormen and Leonard F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. Technical Report PCS-TR93-193, Department of Mathematics and Computer Science, Dartmouth College, August 1993.

[Den70]   Peter J. Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–189, September 1970.

[LF80]   Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, October 1980.

[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.

[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.

[VS90]   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.

[VS92]   Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. Technical Report CS-92-04, Department of Computer Science, Brown University, August 1992. Revised version of Technical Report CS-90-21.