CHAPTER

Matrix-Vector Multiplication

Anarchy, anarchy! Show me a greater evil!
This is why cities tumble and the great houses rain down,
This is what scatters armies!

Sophocles, *Antigone*

8.1 INTRODUCTION

Matrix-vector multiplication is embedded in algorithms solving a wide variety of problems. For example, many iterative algorithms for solving systems of linear equations rely upon matrix-vector multiplication. The conjugate gradient method, which we will examine in Chapter 12, is such an algorithm.

Another practical use of matrix-vector multiplication is in the implementation of neural networks. Neural networks are used in such diverse applications as handwriting recognition, petroleum exploration, airline seating allocation, and credit card fraud detection [114]. The most straightforward way to determine the output values of a k-level neural network from its input values is to perform $k - 1$ matrix-vector multiplications. Moreover, training neural networks is typically done using the backpropagation algorithm, which also has matrix-vector multiplication at its core [98].

In this chapter we design, analyze, implement, and benchmark three MPI programs to multiply a dense square matrix by a vector. Each design is based upon a different distribution of the matrix and vector elements among the MPI processes. Altering the data decomposition changes the communication pattern among the processes, meaning different MPI functions are needed to route the data elements. Hence each of the three programs is significantly different from the other two.

In the course of developing these three programs we introduce four powerful MPI communication functions:

- `MPI_Allgatherv`, an all-gather function in which different processes may contribute different numbers of elements
- `MPI_Scatterv`, a scatter operation in which different processes may end up with different numbers of elements
- `MPI_Gatherv`, a gather operation in which the number of elements collected from different processes may vary
- `MPI_Alltoall`, an all-to-all exchange of data elements among processes

We also introduce five MPI functions that support grid-oriented communicators:

- `MPI_Dims_create`, which provides dimensions for a balanced Cartesian grid of processes
- `MPI_Cart_create`, which creates a communicator where the processes have a Cartesian topology
- `MPI_Cart_coords`, which returns the coordinates of a specified process within a Cartesian grid
- `MPI_Cart_rank`, which returns the rank of the process at specified coordinates in a Cartesian grid
- `MPI_Comm_split`, which partitions the processes of an existing communicator into one or more subgroups

8.2 SEQUENTIAL ALGORITHM

The sequential algorithm for multiplying a matrix by a vector appears in Figure 8.1. Matrix-vector multiplication is simply a series of inner product (or dot product) computations, as illustrated in Figure 8.2. Since computing an inner

Matrix-Vector Multiplication:

Input: $a[0..m-1, 0..n-1]$ — matrix with dimensions $m \times n$
$b[0..n-1]$ — vector with dimensions $n \times 1$

Output: $c[0..m-1]$ — vector with dimensions $m \times 1$

```
for i ← 0 to m − 1
  c[i] ← 0
  for j ← 0 to n − 1
    c[i] ← c[i] + a[i, j] × b[j]
  endfor
endfor
```

Figure 8.1 Sequential matrix-vector multiplication algorithm.

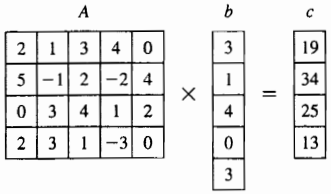

Figure 8.2 Matrix-vector multiplication can be viewed as a series of inner product (dot product) operations. For example, $c_1 = 5 \times 3 + (-1) \times 1 + 2 \times 4 + (-2) \times 0 + 4 \times 3 = 34$.

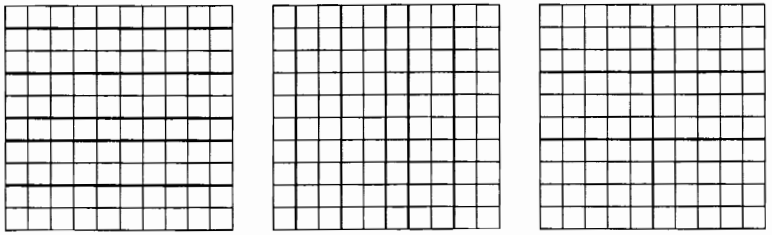

Figure 8.3 Three ways to decompose a two-dimensional matrix. In these examples a 10 × 10 matrix is decomposed among six processes. (a) Rowwise block-striped decomposition. (b) Columnwise block-striped decomposition. (c) Checkerboard block decomposition (processes are organized into a virtual 3 × 2 grid).

product of two n-element vectors requires n multiplication and $n - 1$ additions, it has complexity $\Theta(n)$. Matrix-vector multiplication performs m inner products; hence its complexity is $\Theta(mn)$. When the matrix is square, the algorithm's complexity is $\Theta(n^2)$.

8.3 DATA DECOMPOSITION OPTIONS

We use the domain decomposition strategy to develop our parallel algorithms. There are a variety of ways to partition, agglomerate, and map the matrix and vector elements. Each data decomposition results in a different parallel algorithm.

There are three straightforward ways to decompose an $m \times n$ matrix A: rowwise block striping, columnwise block striping, and the checkerboard block decomposition (Figure 8.3). We have already seen the rowwise block-striped decomposition; it is how we divided the matrix elements among the processes

in our parallel implementation of Floyd's algorithm in Chapter 6. In this decomposition each of the p processes is responsible for a contiguous group of either $\lfloor m/p \rfloor$ or $\lceil m/p \rceil$ rows of the matrix.

A **columnwise block-striped decomposition** is analogous, except that the matrix is divided into groups of columns. Each of the p processes is responsible for a contiguous group of either $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$ columns of the matrix.

In a **checkerboard block decomposition** the processes form a virtual grid, and the matrix is divided into two-dimensional blocks aligning with that grid. Assume the p processors form a grid with r rows and c columns. Each process is responsible for a block of the matrix containing at most $\lceil m/r \rceil$ rows and $\lceil n/c \rceil$ columns.

There are two natural ways to distribute vectors b and c. The vector elements may be **replicated,** meaning all the vector elements are copied on all of the tasks, or the vector elements may be divided among some or all of the tasks. In a **block decomposition** of an n-element vector, each of the p processes is responsible for a contiguous group of either $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$ vector elements.

Why is it acceptable for a task to store vectors b and c in their entirety, but not matrix A? To simplify our argument, let's assume $m = n$. Vectors b and c contain only n elements, the same number of elements as in a single row or column of A. A task storing a row or column of A and single elements of b and c is responsible for $\Theta(n)$ elements. A task storing a row or column of A and all elements of b and c is responsible for $\Theta(n)$ elements. Hence whether the vectors are replicated or distributed in blocks among the tasks, the storage requirements are in the same complexity class.

With three ways to decompose the matrix and two ways to distribute the vector, six possible combinations result. In this chapter we consider three of the six combinations: a rowwise block-striped decomposition of the matrix and replicated vectors; a columnwise block-striped decomposition of the matrix and block-decomposed vectors; and a checkerboard block decomposition of the matrix and vectors block decomposed among the processes in the first column of the process grid.

8.4 ROWWISE BLOCK-STRIPED DECOMPOSITION

8.4.1 Design and Analysis

In this section we develop a parallel matrix-vector multiplication algorithm based on a domain decomposition that associates a primitive task with each row of the matrix A. Vectors b and c are replicated among the primitive tasks. A high-level view of the algorithm resulting from this domain decomposition appears in Figure 8.4. To compute an inner product, a primitive task needs a row and a column vector. Task i has row i of A and a copy of b, so it has all the data it needs to perform the inner product. After the inner product computation, task i

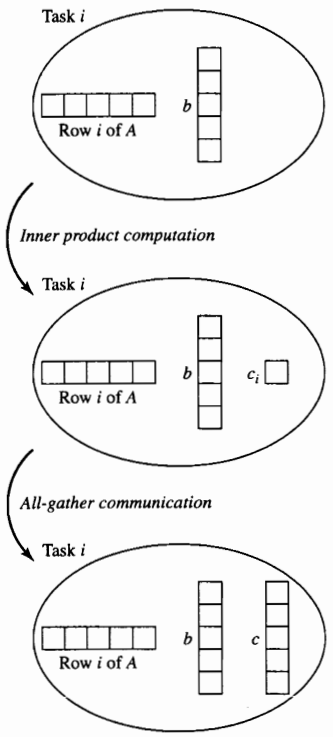

Figure 8.4 In our chosen domain decomposition, each primitive task has a row of the matrix and a copy of the vector. An inner product computation creates an element of the result vector c. An all-gather communication is needed to replicate vector c.

has element i of vector c. However, vectors are supposed to be replicated. An all-gather step communicates each task's element of c to all the other tasks, and the algorithm terminates.

In dense matrix-vector multiplication the number of computational steps needed to perform each inner product is identical. Hence our mapping strategy decision tree suggests we agglomerate primitive tasks associated with contiguous groups of rows and assign each of these combined tasks to a single process, creating a rowwise block-striped partitioning (shown in Figure 8.3a).

As we saw in Figure 8.4, at the end of the inner product computation, each primitive task computes a single element of the result vector. If the matrix decomposition is rowwise block striped, then each process (corresponding to a group of agglomerated tasks) will have a block of elements of the result vector.

When $m = n$, sequential matrix-vector multiplication has time complexity $\Theta(n^2)$. Let's determine the complexity of the parallel algorithm. Each process multiplies its portion of matrix A by vector b. No process is responsible for more than $\lceil n/p \rceil$ rows. Hence the complexity of the multiplication portion of the parallel algorithm is $\Theta(n^2/p)$.

In Chapter 3 we showed that in an efficient all-gather communication each process sends $\lceil \log p \rceil$ messages; the total number of elements passed is $n(p-1)/p$, when p is a power of 2. Hence the communication complexity of the parallel algorithm is $\Theta(\log p + n)$.

Combining the computational portion of the algorithm with the final all-gather communication step, the overall complexity of our parallel matrix-vector multiplication algorithm is $\Theta(n^2/p + n + \log p)$.

Now let's determine the isoefficiency of our parallel algorithm. The time complexity of the sequential algorithm is $\Theta(n^2)$. The only overhead in the parallel algorithm is performing the all-gather operation. When n is reasonably large, message transmission time in the all-gather operation is greater than the message latency. For this reason we simplify the communication complexity to $\Theta(n)$. Hence the isoefficiency function for the parallel matrix-vector multiplication algorithm based on a rowwise block-striped decomposition of the matrix is

$$n^2 \geq Cpn \Rightarrow n \geq Cp$$

When the problem size is n, the matrix has n^2 elements. Hence the memory utilization function $M(n) = n^2$. Computing the scalability function of our parallel algorithm:

$$M(Cp)/p = C^2p^2/p = C^2p$$

To maintain constant efficiency, memory utilization per processor must grow linearly with the number of processors. The algorithm is not highly scalable.

8.4.2 Replicating a Block-Mapped Vector

After each process performs its portion of the matrix-vector product, it has produced a block of result vector c. We must transform this block-mapped vector into a replicated vector, as shown in Figure 8.5.

Let's think about what we must to do to accomplish the transformation. First, each process needs to allocate memory to accommodate the entire vector, rather than just a piece of it. The amount of memory to be allocated depends on the type of the elements stored in the vector: characters, integers, floating-point numbers, or double-precision floating-point numbers, for example. Second, the processes must concatenate their pieces of the vector into a complete vector and

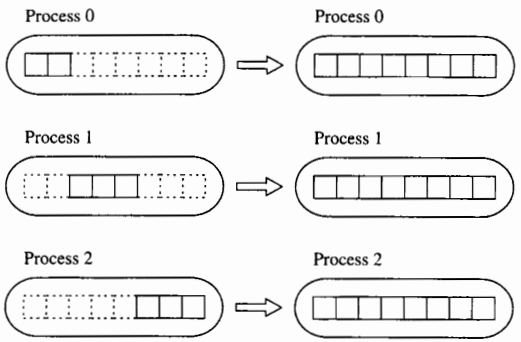

Figure 8.5 Transforming a block-distributed vector into a replicated vector. The elements of a block-distributed vector are distributed among the processes. Each element is stored exactly once. In contrast, when a vector is replicated, every process has every element.

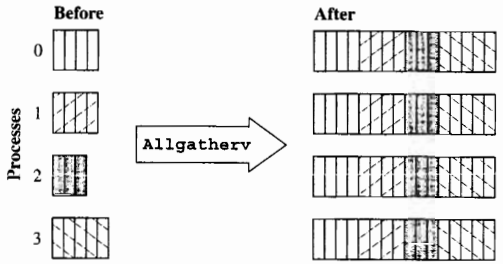

Figure 8.6 Function `MPI_Allgatherv` enables every process in a communicator to construct a concatenation of data items gathered from all of the processes in the communicator. If the same number of items is gathered from each process, the simpler function `MPI_Allgather` may be used.

share the results of the concatenation. Fortunately, a function that can perform the concatenation is in the MPI library.

8.4.3 Function `MPI_Allgatherv`

An **all-gather communication** concatenates blocks of a vector distributed among a group of processes and copies the resulting whole vector to all the processes. The correct MPI function is `MPI_Allgatherv` (illustrated in Figure 8.6).

If the same number of items is gathered from every process, the simpler function `MPI_Allgather` is appropriate. However, in a block decomposition of a vector, the number of elements per process is a constant only if the number of elements is a multiple of the number of processes. Since we cannot be assured of that, we will stick with `MPI_Allgatherv`.

Here is the function header:

```
int MPI_Allgatherv (void* send_buffer, int send_cnt,
   MPI_Datatype send_type, void* receive_buffer,
   int* receive_cnt, int* receive_disp,
   MPI_Datatype receive_type, MPI_Comm communicator)
```

Every parameter except the fourth is an input parameter:

`send_buffer`: the starting address of the data this process is contributing to the "all gather."

`send_cnt`: the number of data items this process is contributing.

`send_type`: the types of data items this process is contributing.

`receive_cnt`: an array indicating the number of data items to be received from each process (including itself).

`receive_disp`: an array indicating for each process the first index in the receive buffer where that process's items should be put.

`receive_type`: the type of the received elements.

`communicator`: the communicator in which this collective communication is occurring.

The fourth parameter, `receive_buffer`, is the address of the beginning of the buffer used to store the gathered elements.

Figure 8.7 illustrates function `MPI_Allgatherv` in action. Each process sets the scalar `send_cnt` to the number of elements in its block. Array `receive_cnt` contains the number of elements contributed by each process; it always has the same values on each process. In this example every process is concatenating elements in the same process order, so the values in array `receive_disp` are identical.

As we have seen, `MPI_Allgatherv` needs to be passed two arrays, each with one element per process. The first array indicates how many elements each process is contributing. The second array indicates the starting positions of these elements in the final, concatenated array.

We often encounter block mappings and in-order concatenation of array elements. We can write a function to build the two arrays needed for this common situation. Our function, called `create_mixed_xfer_arrays`, appears in Appendix B.

With these utilities in place, we can create a function to transform a block-distributed vector into a replicated vector, the last step in the matrix-vector multiplication algorithm. Function `replicate_block_vector` appears in Appendix B.

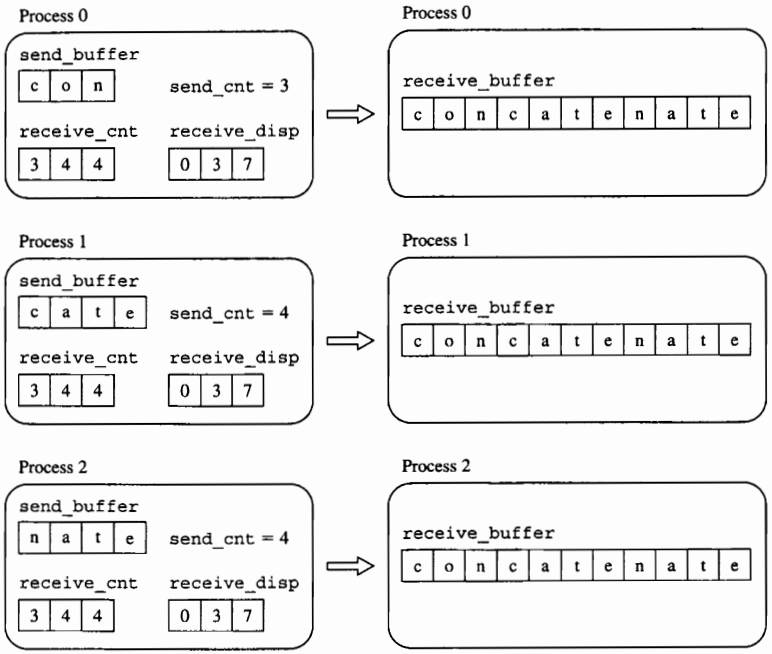

Figure 8.7 Example of how processes initialize variables `send_cnt`, `receive_cnt`, and `receive_disp` when performing a straightforward concatenation using function `MPI_Allgatherv`.

8.4.4 Replicated Vector Input/Output

We need a function to read a replicated vector from an input file. We are assuming the file was created with calls to `fwrite` and should be read with calls to `fread`. The file begins with an integer n representing the number of elements in the vector, followed by the n vector elements.

Process $p-1$ tries to open the data file for reading. If it can open the file, it reads n and broadcasts it to the other processes. If it cannot open the file, it broadcasts a zero to the other processes. If that should happen, all the processes terminate execution. Otherwise, every process allocates memory to store the vector. Process $p-1$ reads the vector and broadcasts it to the other processes. The source code for function `read_replicated_vector` appears in Appendix B.

From a parallel programming point of view, printing a replicated vector is simple. Typically we want a single process to do all the printing, to ensure that messages to standard output don't get scrambled. Since every process has a copy of the vector, all we have to do is ensure that only a single process executes the calls to `printf`, and we're set. The source code for function `print_replicated_vector` is in Appendix B.

8.4.5 Documenting the Parallel Program

With the support functions in place, we can now write a parallel program to perform matrix-vector multiplication. Take another look at Figure 8.4, which summarizes the principal steps of our implementation. The complete C program appears in Figure 8.8.

We begin with the standard include files. We also include the header file `MyMPI.h` for the utility functions we have developed.

We want to be able to change the matrix and vector types with a minimum amount of program editing. In the body of the program we will use `dtype` to indicate the data type of the matrix and vector elements, and we will use `mpitype` as the type designator needed for MPI function calls. At the beginning of the program we use a typedef and a macro definition to establish values for `dtype` and `mpitype`.

After MPI initializations, the processes read and print matrix A. (We developed these functions in Chapter 6.) We also read and print vector b.

Each process allocates its portion of the result vector c and performs its share of the inner products.

At this point every process has a block of c. We convert c to a replicated vector, print it, and end program execution.

8.4.6 Benchmarking

Now let's develop an expression for the expected execution time of the parallel program on a commodity cluster. Let χ represent the time needed to compute a single iteration of the loop performing the inner product. We can determine χ by dividing the execution time of the sequential algorithm by n^2. The expected time for the computational portion of the parallel program is $\chi n\lceil n/p\rceil$.

The all-gather reduction requires each process to send $\lceil \log p\rceil$ messages. Each message has a latency λ. The total number of vector elements transmitted during the all-gather is $n(2^{\lceil \log p\rceil}-1)/2^{\lceil \log p\rceil}$. Each vector element is a double-precision floating-point number occupying 8 bytes. Hence the expected execution time for the all-gather step is $\lambda\lceil \log p\rceil + 8n\left((2^{\lceil \log p\rceil}-1)/2^{\lceil \log p\rceil}\right)/\beta$.

Benchmarking on a commodity cluster of 450 MHz Pentium II processors connected by fast Ethernet reveals that $\chi = 63.4$ nsec, $\lambda = 250$ μsec, and $\beta = 10^6$ byte/sec.

Table 8.1 compares the actual and predicted execution times of our matrix-vector multiplication program solving a problem of size 1,000 on 1, 2, ..., 8 and 16 processors. The actual times reported in the table represent the average execution time over 100 runs of the parallel program. We determine the megaflops rate by dividing the total number of floating-point operations ($2n^2$) by the execution time, and then dividing by a million. The speedup of this program is illustrated in Figure 8.20 at the end of the chapter.

```
/*
 *   Matrix-vector multiplication, Version 1
 */

#include <stdio.h>
#include <mpi.h>
#include "MyMPI.h"

/* Change these two definitions when the matrix and vector
   element types change */

typedef double dtype;
#define mpitype MPI_DOUBLE

int main (int argc, char *argv[]) {
   dtype **a;          /* First factor, a matrix */
   dtype  *b;          /* Second factor, a vector */
   dtype  *c_block;    /* Partial product vector */
   dtype  *c;          /* Replicated product vector */
   dtype  *storage;    /* Matrix elements stored here */
   int     i, j;       /* Loop indices */
   int     id;         /* Process ID number */
   int     m;          /* Rows in matrix */
   int     n;          /* Columns in matrix */
   int     nprime;     /* Elements in vector */
   int     p;          /* Number of processes */
   int     rows;       /* Number of rows on this process */

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &id);
   MPI_Comm_size (MPI_COMM_WORLD, &p);

   read_row_striped_matrix (argv[1], (void *) &a,
      (void *) &storage, mpitype, &m, &n, MPI_COMM_WORLD);
   rows = BLOCK_SIZE(id,p,m);
   print_row_striped_matrix ((void **) a, mpitype, m, n,
      MPI_COMM_WORLD);

   read_replicated_vector (argv[2], (void *) &b, mpitype,
      &nprime, MPI_COMM_WORLD);
   print_replicated_vector (b, mpitype, nprime,
      MPI_COMM_WORLD);

   c_block = (dtype *) malloc (rows * sizeof(dtype));
   c = (dtype *) malloc (n * sizeof(dtype));

   for (i = 0; i < rows; i++) {
      c_block[i] = 0.0;
      for (j = 0; j < n; j++)
         c_block[i] += a[i][j] * b[j];
   }

   replicate_block_vector (c_block, n, (void *) c, mpitype,
      MPI_COMM_WORLD);

   print_replicated_vector (c, mpitype, n, MPI_COMM_WORLD);
   MPI_Finalize();
   return 0;
}
```

Figure 8.8 Version 1 of parallel matrix-vector multiplication.

Table 8.1 Predicted versus actual performance of rowwise striped matrix-vector multiplication program multiplying a 1,000 × 1,000 matrix by a 1,000-element vector.

Processors	Predicted time	Actual time	Speedup	Megaflops
1	0.0634	0.0634	1.00	31.6
2	0.0324	0.0327	1.94	61.2
3	0.0223	0.0227	2.79	88.1
4	0.0170	0.0178	3.56	112.4
5	0.0141	0.0152	4.16	131.6
6	0.0120	0.0133	4.76	150.4
7	0.0105	0.0122	5.19	163.9
8	0.0094	0.0111	5.70	180.2
16	0.0057	0.0072	8.79	277.8

The parallel computer is a commodity cluster of 450 MHz Pentium IIs. Each processor has a fast Ethernet connection to a shared switch.

8.5 COLUMNWISE BLOCK-STRIPED DECOMPOSITION

8.5.1 Design and Analysis

In this section we will design another parallel matrix-vector multiplication algorithm, assuming that each primitive task i has column i of A and element i of vectors b and c. The structure of the resulting parallel algorithm is shown in Figure 8.9.

The computation begins with each task i multiplying its column of A by b_i, resulting in a vector of partial results. At the end of the computation task i needs only a single element of the result vector: c_i. What we need is an **all-to-all communication:** every partial result element j on task i must be transferred to task j. At this point every task i has the n partial results it needs to add in order to produce c_i.

Because every primitive task has identical computation and communication requirements, agglomerating them into larger tasks with the same number of columns (plus or minus one) ensures we have balanced the workload. Hence we will agglomerate the primitive tasks into p metatasks and map one metatask to each process.

In the previous section we assigned to each process a block of rows of A, which we called a rowwise block-striped decomposition. Now we will use a columnwise block-striped decomposition, agglomerating contiguous groups of columns of A, as shown in Figure 8.3b.

Let's determine the complexity of this parallel algorithm, assuming $m = n$. Each process multiplies its portion of matrix A by its block of vector b. No process is responsible for more than $\lceil n/p \rceil$ columns of A or elements of b. Hence the initial multiplication phase has time complexity $\Theta(n(n/p)) = \Theta(n^2/p)$. After the all-gather step, each processor sums the partial vectors collected from the other processors. There are p partial vectors, each of length at most $\lceil n/p \rceil$. The time

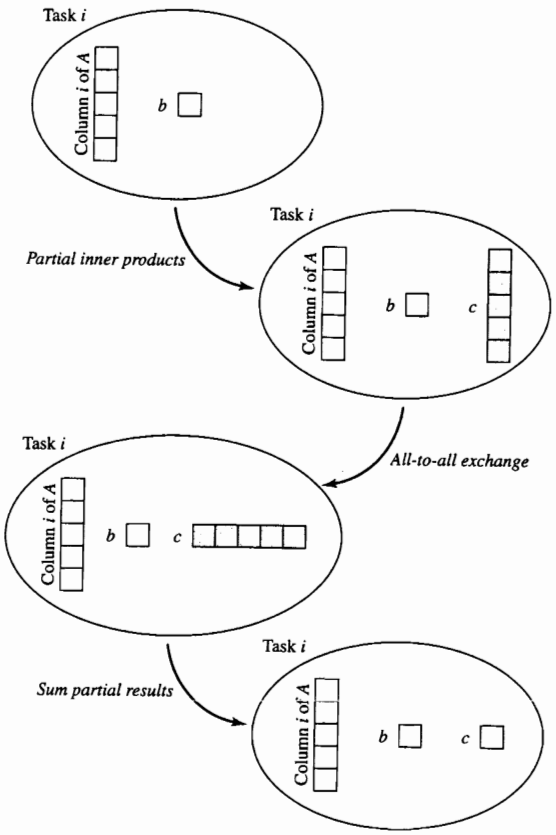

Figure 8.9 In this parallel matrix-vector multiplication algorithm each task has a column of the matrix and an element of the vector. An all-to-all communication moves the appropriate partial results to the tasks that will add them up.

complexity of this step is $\Theta(n)$. Therefore, the overall computational complexity of the parallel algorithm is $\Theta(n^2/p)$.

An all-to-all exchange can be performed in $\lceil \log p \rceil$ steps, using a hypercube communication pattern. During each step every process sends $n/2$ values to its partner and receives $n/2$ values from its partner. The total number of elements sent and received in the all-to-all exchange is $n\lceil \log p \rceil$. Hence the communication complexity of this implementation of all-gather is $\Theta(n \log p)$.

Another way to perform an all-to-all exchange is for each process to send a message to each of the other $p - 1$ processes. Every message contains just those

elements the destination process is supposed to receive from the source. In this implementation the total number of messages is $p - 1$, but the total number of elements passed by each process is less than or equal to n. The communication complexity of this algorithm is $\Theta(p + n)$.

When we combine the computational portion of the algorithm with the final all-gather communication step, the overall complexity of our parallel matrix-vector multiplication algorithm is either $\Theta(n^2/p + n \log p)$ or $\Theta(n^2/p + n + p)$, depending upon which way the all-to-all exchange is implemented.

Now let's determine the isoefficiency of the parallel algorithm. The time complexity of the sequential algorithm is $\Theta(n^2)$. The parallel overhead is limited to the all-to-all exchange operation. When n is reasonably large, the time for the all-to-all exchange is dominated by message transmission time rather than message latency. Using the second approach to implementing all-to-all exchange, we have $\Theta(n)$ complexity for this step, which is performed by all processes.

Hence the isoefficiency function for the parallel matrix-vector multiplication algorithm based on a columnwise block-striped decomposition of the matrix is

$$n^2 \geq Cpn \Rightarrow n \geq Cp$$

This is the same isoefficiency function we derived for the parallel algorithm based on a rowwise block-striped decomposition of the matrix. The parallel algorithm is not highly scalable, because in order to maintain a constant efficiency, memory used per processor must increase linearly with the number of processors.

8.5.2 Reading a Columnwise Block-Striped Matrix

Let's develop a function to read from a file a matrix stored in row-major order and distribute it among the processes in columnwise block-striped fashion. When a row-major matrix with multiple rows has a columnwise block-striped decomposition among multiple processes, the matrix elements controlled by a process are not stored as a contiguous group in the file. In fact, each row of the matrix must be scattered among all of the processes.

We will maintain our tradition of making a single process responsible for I/O. See Figure 8.10. In the first step, one process reads a row of the matrix into a temporary buffer. In step 2 that process scatters the elements of the buffer among all of the processes. The algorithm repeats these steps for the remaining rows of the matrix. The code for function `read_col_striped_matrix` appears in Appendix B.

Function `read_col_striped_matrix` makes use of MPI library routine `MPI_Scatterv` to distribute rows among the processes. Let's take a closer look at this function.

8.5.3 Function `MPI_Scatterv`

The MPI function `MPI_Scatterv` (Figure 8.11) enables a single root process to distribute a contiguous group of elements to all of the processes in a communicator, including itself.

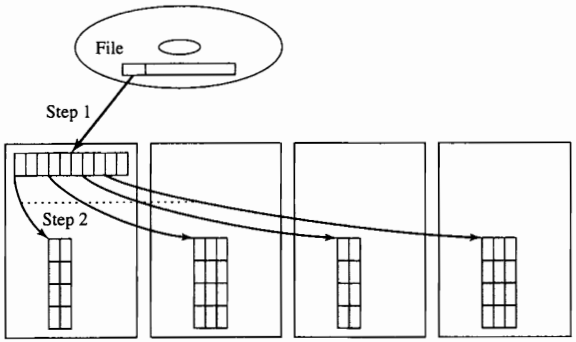

Figure 8.10 In a columnwise block-striped decomposition, each row of the matrix is distributed among the processors. One process inputs a row of the matrix (step 1) and then scatters its elements (step 2).

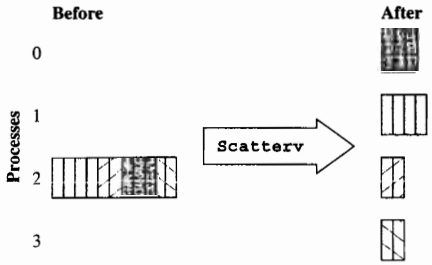

Figure 8.11 The collective communication function `MPI_Scatterv` allows a single MPI process to divide a contiguous group of data items and distribute unique portions to the rest of the processes in the communicator. If the same number of data items is distributed to every process, the simpler function `MPI_Scatter` is appropriate.

Here is the header of function `MPI_Scatterv`:

```
MPI_Scatterv (void *send_buffer, int* send_cnt,
   int* send_disp, MPI_Datatype send_type,
   void *recv_buffer, int recv_cnt,
   MPI_Datatype recv_type, int root, MPI_COMM communicator)
```


The function has nine parameters. All but the fifth are input parameters:

`send_buffer`: pointer to the buffer containing the elements to be scattered.

`send_cnt`: element i is the number of contiguous elements in `send_buffer` going to process i.

`send_disp`: element i is the offset in `send_buffer` of the first element going to process i.

`send_type`: the type of the elements in `send_buffer`.

`recv_buffer`: pointer to the buffer containing this process's portion of the elements received.

`recv_cnt`: the number of elements this process will receive.

`recv_type`: the type of the elements in `recv_buffer`.

`root`: the ID of the process with the data to scatter.

`communicator`: the communicator in which the scatter is occurring.

`MPI_Scatterv` is a collective communication function—all of the processes in a communicator participate in its execution. The function requires that each process has previously initialized two arrays: one that indicates the number of elements the root process should send to each of the other processes, and one that indicates the displacement of this block of elements in the array being scattered. In this case we want to scatter the blocks in process order: process 0 gets the first block, process 1 gets the second block, and so on. While we developed the function `create_mixed_xfer_arrays` in the context of a gather operation, we can use it in this context, too. The number of elements per process and the displacements are identical.

8.5.4 Printing a Columnwise Block-Striped Matrix

Now it's time to design a function to print a columnwise block-striped matrix. To ensure that values are printed in the correct order, we want only a single process to print all the values. In order to print a single row, a single process must gather together the elements of that row from the entire set of processes. Hence the data flow for this function is opposite that of function `read_col_striped_matrix`. The code for function `print_col_striped_matrix` appears in Appendix B.

Function `print_col_striped_matrix` makes use of MPI function `MPI_Gatherv` to collect row elements onto process 0, which then prints the row. The following subsection documents function `MPI_Gatherv`.

8.5.5 Function `MPI_Gatherv`

The MPI collective communication function `MPI_Gatherv` (Figure 8.12) performs this data-gathering function.

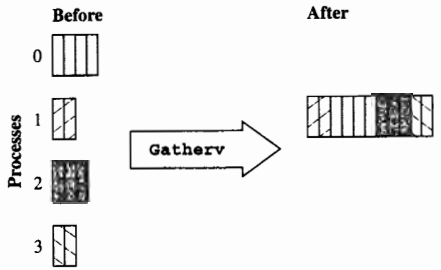

Figure 8.12 Function `MPI_Gatherv` allows a single MPI process to gather together data elements stored on all processes in a communicator. If every process is contributing the same number of data elements, the simpler function `MPI_Gather` is appropriate.

Function `MPI_Gatherv` has this header:

```
MPI_Gatherv (void* send_buffer, int send_cnt,
   MPI_Datatype send_type, void *recv_buffer,
   int* recv_cnt, int* recv_disp, MPI_Datatype recv_type,
   int root, MPI_Comm communicator)
```

The function has nine formal parameters. All of them are input parameters to the function, except the fifth.

`send_buffer`: points to location of first element this process is contributing to the gather.

`send_cnt`: number of elements this process is contributing to the gather.

`send_type`: type of elements stored in `send_buffer`.

`recv_buffer`: points to buffer on root process where gathered elements are to be stored.

`recv_cnt`: element i of this array is the number of elements being gathered from process i.

`recv_disp`: element i of this array is the displacement from the first element of `recv_buffer` where the first element gathered from process i is to be stored.

`recv_type`: type of the array `recv_buffer`.

`root`: process getting the gathered elements.

`communicator`: communicator in which the gather is occurring.

8.5.6 Distributing Partial Results

A series of m inner product operations create the m-element result vector c:

$$c[0] = a[0,0]b[0] + a[0,1]b[1] + \cdots + a[0][n-1]b[n-1]$$

$$c[1] = a[1,0]b[0] + a[1,1]b[1] + \cdots + a[1][n-1]b[n-1]$$

$$\cdots$$

$$c[m-1] = a[m-1,0]b[0] + a[m-1,1]b[1] + \cdots + a[m-1][n-1]b[n-1]$$

In our domain decomposition, we associate with each primitive task i column i of matrix A and element i of vector b. Multiplying each element of the column by b_i yields $a[0,i]b[i], a[1,i]b[i], a[2,i]b[i], \ldots, a[n-1,i]b[i]$. Product $a[0,i]b[i]$ is the ith term of the inner product for $c[0]$, while $a[1,i]b[i]$ is the ith term of the inner product for $c[1]$, and so on. In other words, the n multiplications performed by task i yield n terms that are not supposed to be added as an inner product. Instead, term j is part of the inner product forming $c[j]$, for all j, $0 \leq j < n$.

After performing n multiplications, each task needs to distribute $n-1$ result terms it doesn't need to the other processors and collect $n-1$ result terms it does need from them. This is an example of an **all-to-all exchange.** After the exchange, primitive task i adds the n elements now in its possession ($a[i,0]b[0]$ produced by task 0, $a[i,1]b[1]$ produced by task 1, etc.) to produce $c[i]$.

In our implementation of matrix-vector multiplication, we have assigned a block of columns of A and a block of elements of b to each task. The principle, however, is the same. Each task will exchange blocks it doesn't need for blocks it does need. Each task i receives `BLOCK_SIZE(i,p,n)` elements from each other task. After the exchange, it has p subarrays of size `BLOCK_SIZE(i,p,n)`. It adds these subarrays to form its block of c.

8.5.7 Function `MPI_Alltoallv`

Function `MPI_Alltoallv` enables every process to exchange values with every other process in a communicator (see Figure 8.13). The function has this header:

```
int MPI_Alltoallv (void *send_buffer, int *send_count,
            int *send_displacement, MPI_Datatype send_type,
            void *recv_buffer, int *recv_count,
            int *recv_displacement, MPI_Datatype recv_type,
            MPI_Comm communicator)
```

`send_buffer` is the starting address of the array of elements to be exchanged.

`send_count` is an array; element i indicates the number of elements destined for process i.

`send_displacement` is an array; element i indicates the starting point in `send_buffer` of the elements destined for process i.

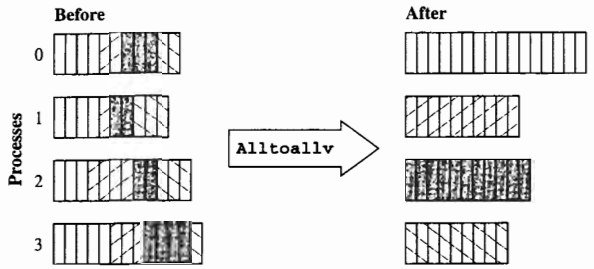

Figure 8.13 Function `MPI_Alltoallv` allows every MPI process to gather data items from all the processes in the communicator. The simpler function `MPI_Alltoall` should be used in the case where all of the groups of data items being transferred from one process to another have the same number of elements.

`send_type` is the type of the elements in `send_buffer`.

`recv_buffer` is the starting address of the buffer used to collect incoming elements (as well as the elements the process is sending to itself).

`recv_count` is an array; element i is the number of elements to receive from process i.

`recv_displacement` is an array; element i is the starting point in `recv_buffer` of the elements received from process i.

`recv_type` is the type the elements should be converted to before they are put in `recv_buffer`.

`communicator` indicates the set of processes participating in the all-to-all exchange.

8.5.8 Documenting the Parallel Program

We now have a firm foundation on which to build our second parallel matrix-vector multiplication program. The source for the program appears in Figure 8.14.

After the usual MPI initializations, we call function `read_col_striped_matrix` to input the contents of the data file containing a matrix and distribute it among the processes. We then print the matrix.

Similarly, we read vector b and print it.

Each process allocates memory to store `c_part_out`, the "outgoing" partial result vector. Most of these elements will end up on other processes. Each process also allocated memory for `c_part_in`, the "incoming" pieces of the other processes' partial result vectors.

Next is the actual computation. Each process multiplies its portion of the matrix (having dimensions `n` x `local_els`) by its portion of the vector (having length `local_else`), resulting in a partial result vector of length `n`.


```
/*
 *   Matrix-vector multiplication, Version 2
 */

#include <stdio.h>
#include <mpi.h>
#include "MyMPI.h"

/* Change these two definitions when the matrix and vector
   element types change */

typedef double dtype;
#define mpitype MPI_DOUBLE

int main (int argc, char *argv[]) {
   dtype **a;              /* The first factor, a matrix */
   dtype  *b;              /* The second factor, a vector */
   dtype  *c;              /* The product, a vector */
   dtype  *c_part_out;     /* Partial sums, sent */
   dtype  *c_part_in;      /* Partial sums, received */
   int    *cnt_out;        /* Elements sent to each proc */
   int    *cnt_in;         /* Elements received per proc */
   int    *disp_out;       /* Indices of sent elements */
   int    *disp_in;        /* Indices of received elements */
   int     i, j;           /* Loop indices */
   int     id;             /* Process ID number */
   int     local_els;      /* Cols of 'a' and elements of 'b'
                              held by this process */
   int     m;              /* Rows in the matrix */
   int     n;              /* Columns in the matrix */
   int     nprime;         /* Size of the vector */
   int     p;              /* Number of processes */
   dtype  *storage;        /* This process's portion of 'a' */

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &id);
   MPI_Comm_size (MPI_COMM_WORLD, &p);

   read_col_striped_matrix (argv[1], (void ***) &a,
      (void **) &storage, mpitype, &m, &n, MPI_COMM_WORLD);
   print_col_striped_matrix ((void **) a, mpitype, m, n,
      MPI_COMM_WORLD);
   read_block_vector (argv[2], (void **) &b, mpitype,
      &nprime, MPI_COMM_WORLD);
   print_block_vector ((void *) b, mpitype, nprime,
      MPI_COMM_WORLD);

   /* Each process multiplies its columns of 'a' and vector
      'b', resulting in a partial sum of product 'c'. */

   c_part_out = (dtype *) my_malloc (id, n * sizeof(dtype));
   local_els = BLOCK_SIZE(id,p,n);

   for (i = 0; i < n; i++) {
      c_part_out[i] = 0.0;
```

Figure 8.14 Second parallel matrix-vector multiplication program.

```
      for (j = 0; j < local_els; j++)
         c_part_out[i] += a[i][j] * b[j];
   }

   create_mixed_xfer_arrays (id, p, n, &cnt_out, &disp_out);
   create_uniform_xfer_arrays (id, p, n, &cnt_in, &disp_in);
   c_part_in =
      (dtype*) my_malloc (id, p*local_els*sizeof(dtype));
   MPI_Alltoallv (c_part_out, cnt_out, disp_out, mpitype,
      c_part_in, cnt_in, disp_in, mpitype, MPI_COMM_WORLD);

   c = (dtype*) my_malloc (id, local_els * sizeof(dtype));
   for (i = 0; i < local_els; i++) {
      c[i] = 0.0;
      for (j = 0; j < p; j++)
         c[i] += c_part_in[i + j*local_els];
   }
   print_block_vector ((void *) c, mpitype, n, MPI_COMM_WORLD);
   MPI_Finalize();
   return 0;
}
```

Figure 8.14 (contd.) Second parallel matrix-vector multiplication program.

The outgoing pieces of `c_part_out` have different sizes. A call to `create_mixed_xfer_arrays` sets up the counts and displacements for these pieces. In contrast, the incoming pieces of `c_part_in` all have the same size. Calling `create_uniform_xfer_arrays` correctly initializes the counts and displacements for these pieces. The MPI function `MPI_Alltoallv` performs the all-to-all communication, routing each piece to its destination.

Now each process has `n` chunks of length `local_els`. Adding these together yields its portion of the result vector `c`.

8.5.9 Benchmarking

Now let's develop an expression for the expected execution time of the parallel program on a commodity cluster. As before, χ is the time needed to compute a single iteration of the loop performing the inner product. The expected time for the computational portion of the parallel program is $\chi n\lceil n/p\rceil$.

The algorithm performs an all-to-all exchange of partially computed portions of the vector c. There are two common ways to perform an all-to-all exchange. The first way is for each process to send $\lceil \log p\rceil$ messages of length $n/2$. This requires that each process send $\lceil \log p\rceil$ messages and transmit a total of $\lceil \log p\rceil n/2$ data elements.

The second way is for each process to send directly to each of the other processes the elements destined for that process. This requires that each process send $p-1$ messages and transmit a total of about $n(p-1)/p$ data elements.

For a large n, the message transmission time dominates the message latency, and the second approach is superior. Assuming each message has latency λ, the time needed to transmit a single byte is $1/\beta$, and the time needed to perform an all-gather of double-precision floating-point variables is $(p-1)(\lambda + 8n/(p\beta))$.

Table 8.2 Comparison of predicted versus actual performance of our second matrix-vector multiplication program on a commodity cluster of 450 MHz Pentium IIs.

Processors	Predicted time	Actual time	Speedup	Megaflops
1	0.0634	0.0638	1.00	31.4
2	0.0324	0.0329	1.92	60.8
3	0.0222	0.0226	2.80	88.5
4	0.0172	0.0175	3.62	114.3
5	0.0143	0.0145	4.37	137.9
6	0.0125	0.0126	5.02	158.7
7	0.0113	0.0112	5.65	178.6
8	0.0104	0.0100	6.33	200.0
16	0.0085	0.0076	8.33	263.2

Benchmarking on a commodity cluster of 450 MHz Pentium II processors connected by fast Ethernet reveals that $\chi = 63.4$ nsec, $\lambda = 250$ μsec, and $\beta = 10^6$ byte/sec.

Table 8.2 compares the actual and predicted execution times of our matrix-vector multiplication program solving a problem of size 1,000 on $1, 2, \ldots, 8$ and 16 processors. The actual times reported in the table represent the average execution time over 100 runs of the parallel program. The speedup of this program is illustrated in Figure 8.20 at the end of the chapter.

8.6 CHECKERBOARD BLOCK DECOMPOSITION

8.6.1 Design and Analysis

In this domain decomposition we associate a primitive task with each element of the matrix. The task responsible for $a_{i,j}$ multiplies it by b_j, yielding $d_{i,j}$. Each element c_i of the result vector is $\sum_{j=0}^{n-1} d_{i,j}$. In other words, for each row i, we add all the $d_{i,j}$ terms to produce element i of vector c, as shown in Figure 8.15.

We agglomerate primitive tasks into rectangular blocks and associate a task with each block (as shown in Figure 8.3c). Since all the blocks have about the same size, the work required within each block is about the same, so we will set the block sizes so that we can map one task to each process. We can think of the processes as forming a two-dimensional grid. Vector b is distributed by blocks among the tasks in the first column of the task grid (Figure 8.16).

Now we can lay out the three principal steps of the parallel algorithm and the communications patterns necessary to accomplish these steps (see Figure 8.17). The task associated with matrix block $A_{i,j}$ performs a matrix-vector multiplication of this block with subvector b_j. Our first step, then, is to redistribute vector b so that each task has the correct portion of b. (We'll figure out how to do this