Chapter 8 Matrix-Vector Multiplication

"We can't solve problems by using the same kind of thinking we used when we created them." - Albert Einstein

8.1 Introduction

The purpose of this chapter is two-fold: on a practical level, it introduces many new MPI functions and concepts to expand your programming toolkit; on a theoretical level, it introduces three different ways of applying data decomposition to the exact same problem, showing how the differences in data decomposition lead to very different algorithms. In addition, the problem that we address is a very important and ubiquitous one, namely (left) multiplication of a vector by a matrix.

Multiplication of a vector by a matrix is a fundamental operation that arises whenever there are systems of linear equations to be solved. Such systems of equations arise in fields as diverse as computer graphics, economics, quantum mechanics, electronics, graph theory, probability theory, statistical analysis, artificial neural networks, and operations research. Because multiplying a vector by a matrix is of such widespread use, it has been thoroughly studied and parallelized. It is not an objective of this chapter to invent a new great parallel algorithm; it is to introduce and explain some important parallel programming and algorithm design techniques. In this chapter it is certainly the journey that matters, not the destination, and on the way we will introduce the following MPI functions:

MPI Function	**Purpose**
`MPI_Allgatherv`	Gathers data from all processes and delivers it to all, allowing a varying count of data to be contributed by each process.
`MPI_Scatterv`	Scatters a buffer in parts to all tasks in a communication group, allowing a varying count of data to be sent to each process.
`MPI_Gatherv`	Gathers varying amounts of data from all processes to the root process.
`MPI_Alltoall`	A collective operation in which all processes send the same amount of data to each other, and receive the same amount of data from each other.
`MPI_Dims_create`	Creates a division of processors in a Cartesian grid.
`MPI_Cart_create`	Makes a new communicator to which Cartesian topology information has been attached.
`MPI_Cart_get`	Returns information about the Cartesian topology of the communicator whose handle is supplied to it.
`MPI_Cart_coords`	Returns the coordinates in Cartesian topology of a process with a given rank in group.
`MPI_cart_rank`	Returns the rank of a process at the given coordinates in a Cartesian topology.
`MPI_Comm_split`	Creates new communicators based on colors and keys.

8.2 A Brief Review

This is a brief refresher on matrix-vector multiplication. An $n \times 1$ vector may be multiplied on the left by an $m \times n$ matrix, resulting in an $m \times 1$ vector. Mathematically, if A is an $m \times n$ matrix and X is a $n \times 1$ vector, then the product AX is an $m \times 1$ vector B whose i^{th} entry is the ***inner product*** of row i of A and

X:

$$B_i = \sum_{j=1}^{n} A_{i,j} \cdot X_j \tag{8.1}$$

The inner product of two vectors x and y of length n each is the sum $x_1y_1 + x_2y_2 + \cdots + x_ny_n$. Some people call this the ***dot product***. Because the inner product is the sum of terms x_iy_i, its computation is an example of a reduction. Figure 8.1 illustrates the matrix-vector product and shows each inner product that must be computed. Each inner product requires n multiplications and $n-1$ additions, and there are m inner products. Therefore, any sequential algorithm, in the general case, has time complexity $\Theta(mn)$.

$$\begin{array}{|c|c|c|c|c|c|}\hline 3 & 1 & 0 & 4 & 2 & -1 \\ \hline 0 & 1 & -1 & 5 & -2 & 3 \\ \hline 1 & 0 & 2 & 3 & 1 & 0 \\ \hline 4 & 2 & -1 & -1 & 0 & -3 \\ \hline \end{array} \times \begin{array}{|c|}\hline 1 \\ \hline 0 \\ \hline 2 \\ \hline 4 \\ \hline 1 \\ \hline -2 \\ \hline \end{array} = \begin{array}{|c|}\hline 23 \\ \hline 10 \\ \hline 18 \\ \hline 4 \\ \hline \end{array}$$

implies the following four inner products:

$$(3 \cdot 1) + (1 \cdot 0) + (0 \cdot 2) + (4 \cdot 4) + (2 \cdot 1) + ((-1) \cdot (-2)) = 23$$
$$(0 \cdot 1) + (1 \cdot 0) + (-1 \cdot 2) + (5 \cdot 4) + (-2 \cdot 1) + (3 \cdot (-2)) = 10$$
$$(1 \cdot 1) + (0 \cdot 0) + (2 \cdot 2) + (3 \cdot 4) + (1 \cdot 1) + (0 \cdot (-2)) \quad = 18$$
$$(4 \cdot 1) + (2 \cdot 0) + (-1 \cdot 2) + (-1 \cdot 4) + (0 \cdot 1) + (3 \cdot (-2)) = 4$$

Figure 8.1: A 4×6 matrix times a 6×1 vector.

8.3 A Sequential Algorithm

The obvious sequential algorithm for matrix-vector multiplication is given in Listing 8.1.

Listing 8.1: Sequential Matrix-Vector Product

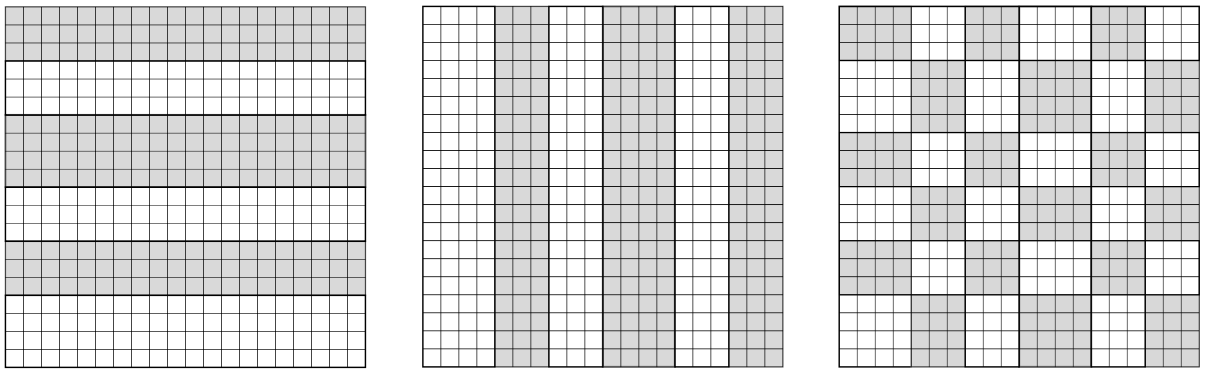
```
// Input:   A, an m by n matrix,
//          X, an n by 1 vector
// Output: B, an m by 1 vector

for i = 0 to m-1
    B[i] = 0;
    for j = 0 to n-1
        B[i] = B[i] + A[i,j] * X[j];
    end for
end for
```

8.4 Data Decomposition

The sequential algorithm clearly has data parallelism. For one, each iteration of the outer loop is independent of all others, because each updates a different element of the output vector B. The inner loop also has data parallelism, because each iteration multiplies $X[j]$ by a unique matrix element $A[i,j]$ before adding it to $B[j]$. This is a reduction whose elements are products of the form $A[i,j] \cdot X[j]$. We could, if we wanted, associate a primitive task to each matrix element $A[i,j]$ which would be responsible for the product $A[i,j] \cdot X[j]$. This would result in mn primitive tasks. We should assume that m and n are very large numbers, otherwise we would not bother to parallelize the algorithm, so we will probably not have anything near mn processors available. Therefore, we will need to agglomerate to reduce the number of tasks. We will consider three alternative decomposition strategies: two one-dimensional strategies and one two-dimensional strategy.

This leads to three ways to decompose the matrix: by rows, by columns, or by rectangular subblocks. In Chapter 6, for the parallel version of Floyd's algorithm, we used a one-dimensional, row-wise decomposition



(a) Row-wise block-striped matrix decomposition.

(b) Column-wise block-striped matrix decomposition.

(c) Checkerboard block decomposition.

Figure 8.2: Row-wise striped, column-wise striped, and checkerboard block decomposition of a matrix.

to decompose the adjacency matrix into sets of adjacent rows to be assigned to each process. This method of decomposition is called ***row-wise block-striped decomposition***, shown here in Figure 8.2a. Symmetrically, we could decompose column-wise, as shown in Figure 8.2b, and this is called ***column-wise block-striped decomposition***. The *block-striped* part of the name expresses the idea that each process has a block of data, and in these cases, the blocks are stripes – long and thin as opposed to fuller and closer to square-shaped. These are essentially one-dimensional. The third alternative is to decompose the matrix in two-dimensions, into an $r \times c$ grid of rectangular subblocks, as shown in Figure 8.2c, which is called a ***checkerboard block decomposition*** or a ***2D-block decomposition***. In the figure, $r = c = 6$; although it need not be a square grid, for performance purposes it is better if it is.

If there are p processes, then in row-wise block-striped decomposition, each process will have a group of either $\lceil n/p \rceil$ or $\lfloor n/p \rfloor$ consecutive rows, each a primitive task. In the column-wise block-striped decomposition, each process will have a group of either $\lceil n/p \rceil$ or $\lfloor n/p \rfloor$ consecutive columns, each a primitive task. In a checkerboard block decomposition, each matrix element is a primitive task, and the number p determines the possible decomposition of the matrix. It works best when p is a square number, i.e., $p = q^2$ for some q, so that the grid can have an equal number of rows and columns. If p is a prime number, the matrix cannot be decomposed using the checkerboard block method, because it degenerates into either the row-wise striped or column-wise striped method, as either the number of rows or the number of columns in the grid will be 1. If p is not square and not prime, there are two factors r and c such that $p = rc$, and they should be chosen to be as close to each other as possible. For example, if $p = 24$, then it is better to create a 4×6 grid than an 8×3 grid. Given that the grid is $r \times c$, then each process will have a rectangular block of adjacent matrix elements with either $\lceil m/r \rceil$ or $\lfloor m/r \rfloor$ rows and either $\lceil n/c \rceil$ or $\lfloor n/c \rfloor$ columns.

No matter which decomposition we choose, each process will store in its local memory just the part of the matrix for which it is responsible. The alternative is to replicate the entire matrix within each process's memory. There are two reasons for which this is not a good idea. First is that the solution would not scale well because as the problem size increased, we would not be able to store the entire matrix in each process's local memory; under the assumption that these would be large matrices, we would run out of memory. Secondly, replicating the entire matrix is very inefficient in terms of time and space. In terms of space, we would be storing p copies of an $m \times n$ matrix instead of a total of one copy distributed among the p processes. If each entry used 8 bytes, this would be a waste of $8mn(p-1)$ bytes of total memory. In terms of time, it means that the algorithm would have taken time to distribute those matrix elements in the beginning, even though the processes had no need for most of the matrix that they received.

In contrast to this, replicating the two vectors is not such a bad idea, for various reasons. First, depending on the method of matrix decomposition, each process might need the entire vector anyway. For example, in the row-wise decomposition, each process will need the entire X vector. Even if the process will not need the entire vector, it is not particularly wasteful of storage or time because the vectors are of length n, which is much smaller than the matrix size mn, and the complexity of storing the vector will not increase the space or time complexity of the algorithm, and will ultimately reduce communication overhead. Nonetheless, it is worth investigating solutions with both replicated vectors and vectors decomposed and distributed among the processes, which are called ***block-decomposed vectors***.

We will partly follow the approach used by Quinn [1]: we will develop an algorithm using a row-wise block-striped decomposition of the matrix with replicated vectors, and a checkerboard block decomposition of the matrix with block-decomposed vectors. We willonly briefly describe an algorithm using column-wise block striped decomposition of the matrix with block-decomposed vectors.

8.5 Row-Wise Block-Striped Matrix Decomposition

This is probably the easiest algorithm to understand, in part because we developed Floyd's algorithm in Chapter 6, and in part because the communication pattern is the simplest. The product of an $m \times n$ matrix and a $n \times 1$ vector can be viewed as m independent inner products in which a row of the matrix and the vector are multiplied together. The task that owns row j computes the j^{th} value of the result vector, i.e., $B[j]$. Therefore, if a primitive task is assigned to each row, then each primitive task needs one row of the matrix A, the vector X, and one element of the result vector B. However, in order to assemble the entire result vector when each task has finished computing its result, the algorithm will replicate the entire result vector B in each task, because it will be easy to gather the final results if each task has the entire vector.

What communication will be necessary? Initially, each row of the matrix and a copy of the input vector need to be distributed to each task. To perform the computation there is no communication among the tasks. But after they each compute their respective elements of the result vector, that vector needs to be assembled, which implies that they each need to contribute their value $B[j]$ to a result vector stored in one of the tasks. As mentioned above, it is actually more convenient to create a copy of the entire result vector in each task, so that a gathering operation can collect the results into every task.

Because the number of rows of the matrix will be much larger than the number of processors, we agglomerate the primitive tasks to reduce task creation overhead and to reduce the communication overhead of distributing rows and collecting results. Assuming that the matrix is dense[1], the inner product takes the same amount of time in each primitive task, and the decision tree suggests that we agglomerate by assigning adjacent rows to each task and assigning that task to a single process. This leads to a row-wise block striping decomposition.

As with all MPI programs, the algorithm uses the *SPMD* (single program multiple data) model, in which each process executes the same program. Therefore the following pseudo-code description is written with that in mind:

1. Initialize MPI and then store the process rank into `id` and number of processes into `p`.
2. Read the set of rows for the current process from the matrix into a local two-dimensional array named `A` in the current process. We can use the same function as we did in Chapter 6, namely `read_and_distribute_matrix_byrows`. This function returns the total number of rows and columns of the matrix in `m` and `n` respectively, but each process only gets the rows it will own.
3. Read the entire vector into a local one-dimensional array named `X` in the current process. Each process will have a copy of the vector. Reading the entire vector into an array in each process is simpler than reading the entire matrix and distributing it, but the structure of the code is the same. Essentially, we need a function that will do the following:
   (a) If the process rank is p-1, then open the file and read the size of the vector into a variable;
   (b) Each process participates in a broadcast by process p-1 of the size;
   (c) Each process allocates locally the storage needed for a vector of this size;
   (d) If the process rank is p-1, then the process reads the rest of the file into its storage for the vector;

[1]A matrix is ***dense*** if most of its entries are non-zero, otherwise it is ***sparse***. Although there is no uniformly-agreed upon cut-off point between dense and sparse, the operational definition is that, if the number of non-zeros in an $N \times N$ matrix is $O(N)$, it is sparse, otherwise it is dense.

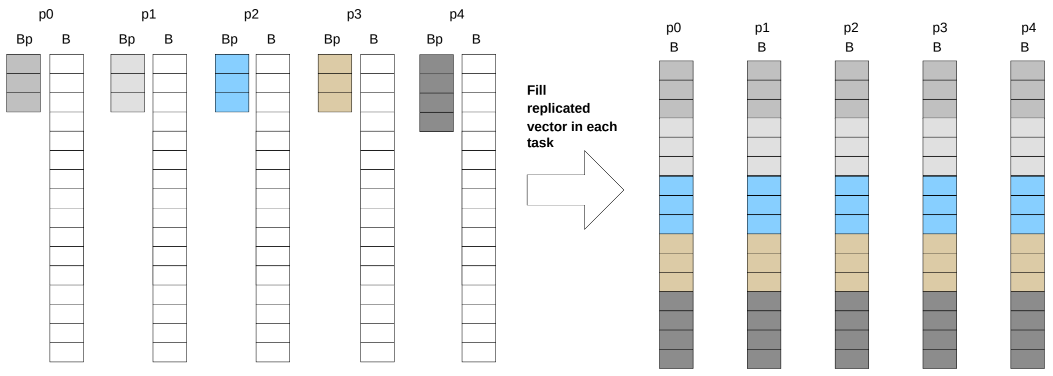


Figure 8.3: Using a block-distributed vector to create a replicated vector in every task. Before the operation, each task has a different block of the vector that it has computed (in `Bp`), as well as uninitialized storage for a result vector (`B`). After the operation, the result vector `B` has been filled with the blocks contributed by all of the tasks, in rank order. This illustrates what has to take place in the final stage of the row-wise block striped algorithm. (The arrays `Bp` still exist, but are omitted to reduce space in the figure.)

(e) Each process participates in a broadcast by process p-1 of the vector contents.

We will name this function `read_and_replicate_vector`.

4. Determine the number of rows that current process owns and allocate a local one-dimensional array named `B_partial` of that size to store the results of the inner products of its rows and the vector `X`. For example, if the current process has five rows of a matrix of doubles, then it will allocate `5*sizeof(double)` bytes. E.g.,

```
B_partial = (double*) malloc ( num_local_rows * sizeof(double) );
```

   This is the subblock of the result vector that must be gathered when all processes finish.

5. Create an uninitialized copy of the result vector for the current process. The result vector has to have the same number of rows as the matrix. E.g.,

```
B = (double*) malloc ( m * sizeof(double) );
```

   This is the vector into which every process will send its subblock of partial results.

6. Compute the inner products of the rows assigned to this process with the vector `X`, storing the results in the partial vector `B_partial`:

```
for (i = 0; i < num_local_rows; i++) {
    B_partial[i] = 0.0;
    for (j = 0; j < n; j++)
        B_partial[i] += A[i][j] * X[j];
}
```

7. Collect the partial results from all processes into the result vector `B`. This will use the `MPI_Allgatherv` communication function. Figure 8.3 illustrates the state of the result vector in each of five tasks before and after this step. The function that carries out this step will be called `replicate_block_vector`.

8. If the rank of the process is 0, print the results to standard output.

There are two parts of the above algorithm that require further development. One is the function in step 2 that will read and replicate the vector. The other is the function in step 7 that will gather the partial results from all processes into the replicated result vector, which uses a new collective communication function. We will start with reading and replicating the vector.

8.5.1 Reading and Replicating a Vector

Reading a vector from a file and replicating it in all processes can be viewed as a simpler, special case of reading a two-dimensional matrix in a distributed way among the processes. Instead of having n columns, there is a single column, so we can view a vector as a matrix whose rows are of length one. With the matrix distribution algorithm, each process received selected rows. In this case however, each process will acquire all rows. This difference leads to a different communication structure. Listing 8.2 contains the pseudo-code for this function.

Listing 8.2: Pseudo-code for read_and_replicate_vector()

```
input parameters:
                  filename    - name of file to read
                  **V         - vector read from file
                  *length     - length of vector

int id           = process rank process;
int p            = number of processes;
int element_size = number of bytes in element type;

if ( p-1 == id ) {
    open the binary file named by filename, which contains the vector.
    if  failure
        set an error code and set length to zero.
    otherwise
        read the first number, assigning to *length.
}
Do a broadcast from process p-1 of the number *length to all other processes.
if ( 0 == *length )
    process must exit -- vector was not read correctly.

Allocate a vector V of size *length * element_size.

if ( p-1 == id ) {
    read *length *element_size bytes from the file into the location starting at V
    and close the file.
}
Do a broadcast from process p-1 of the data whose start address is V, consisting
   of *length elements of the given element type.
```

If you refer back to the code for `read_and_distribute_matrix_byrows.c`, you will see the similarities in the two algorithms. However, because all processes get a copy of the entire vector, the communication in this algorithm is a global communication – every process gets a copy of the same data, and hence this algorithm uses a broadcast, whereas the other function needed point-to-point communication using `MPI_Send` and `MPI_Recv`.

This pseudo-code leads naturally to the function appearing in Listing 8.3.

Listing 8.3: read_and_replicate_vector()

```
void read_and_replicate_vector (
          char          *filename,       /* [IN]  name of file to read          */
          MPI_Datatype dtype,            /* [IN]  matrix element type           */
          void        **vector,          /* [OUT] vector read from file         */
          int           *length,         /* [OUT] length of vector              */
          int           *errval,         /* [OUT] success/error code on return  */
          MPI_Comm       comm)           /* [IN] communicator handle            */
{
    int     id;               /* process rank process */
    int     p;                /* number of processes  in communicator group */
    size_t element_size;      /* number of bytes in matrix element type */
    int     mpi_initialized; /* flag to check if MPI_Init was called already */
    FILE    *file;           /* input file stream pointer */
    int     nlocal_rows;     /* number of rows calling process "owns" */

    /* Make sure we are being called by a program that init-ed MPI */
    MPI_Initialized(&mpi_initialized);
    if ( !mpi_initialized ) {
       *errval = -1;
       return;
    }

    /* Get process rank and the number of processes in group */
    MPI_Comm_size (comm, &p);
    MPI_Comm_rank (comm, &id);

    /* Get the number of bytes in a vector element */
    element_size = get_size (dtype);
    if ( element_size <= 0 ) {
       *errval = -1;
       return;
    }

    if ( p-1 == id ) {
        /* Process p-1 opens the binary file containing the vector and
           reads the first number, which is the length of the vector. */
        file = fopen (filename, "r");
        if ( NULL == file )
            *length = 0;
        else
            fread (length, sizeof(int), 1, file);
    }

    /* Process p-1 broadcasts the length to all other processes */
    MPI_Bcast (length, 1, MPI_INT, p-1, MPI_COMM_WORLD);

    /* No process continues if length is zero. */
    if ( 0 == *length  ) {
       *errval = -1;
       return;
    }

    /* Each process allocates memory for full size of vector. */
    *vector = malloc (*length * element_size);
    if ( NULL == *vector ) {
        printf ("malloc failed for process %d\n", id);
        MPI_Abort (MPI_COMM_WORLD, MALLOC_ERROR);
    }

    /* Process p-1 reads the file and stores data in its local copy. */
    if ( p-1 == id ) {
       fread (*vector, element_size, *length, file);
       fclose (file);
    }

    /* Process p-1 broadcasts the vector to all other processes. */
    MPI_Bcast (*vector, *length, dtype, p-1, MPI_COMM_WORLD);
}
```

8.5.2 Replicating a Block-Mapped Vector

The last remaining piece of this puzzle is to create a function that can collect the pieces of the result vector computed by each process into a single vector that is replicated in each process, as depicted in Figure 8.3. Having done this, the vector can be printed by any of those processes. We cannot use a broadcast to do this, nor can we easily use point-to-point communication, without extensive conditional code. In Chapter 3, we introduced the all-gather operation. Recall that the all-gather operation collects data from every process onto every process. The MPI library provides two different all-gather functions, one that expects every process to contribute the exact same amount of data, and a second that allows each process to contribute a varying amount of data. We cannot use the first one, which is much simpler than the second, because in the row-wise block-striped decomposition of the matrix, different processes may have different numbers of rows, and therefore will have different size blocks of the result vector. This latter one is named `MPI_Allgatherv` (v for varying.)

8.5.2.1 `MPI_Allgatherv`

The syntax of `MPI_Allgatherv` is

```
    #include <mpi.h>
    int MPI_Allgatherv(
        void *sendbuf,         /* Initial address of send buffer          */
        int sendcount,         /* Number of elements in the send buffer   */
        MPI_Datatype sendtype, /* Datatype of each send buffer element    */
        void *recvbuf,         /* Address of receive buffer               */
        int *recvcount,        /* Integer array containing the number of
                                  elements to be received from each process.
        int *displs,           /* Integer array of displacements.         */
                                  Entry i specifies the offset relative to
                                  recvbuf at which to place the incoming
                                  data from process i.                    */
        MPI_Datatype recvtype, /* Datatype of receive buffer elements     */
        MPI_Comm comm          /* Communicator handle                     */
    )
```

The `MPI_Allgatherv` function has several parameters. The first is the starting address of the data that the calling process is sending, and the second is a count of the number of elements to be sent from that address. The type of each element is specified in the third parameter. The fourth parameter is the starting address in the process's local memory into which the gathered data will be stored. The next two parameters are integer arrays, each containing p elements, where p is the number of processes in the communicator group. The first is an array of sizes; the j^{th} entry of this array specifies the number of elements that process j is contributing to the gathered data. The second array is an array of displacements from the beginning of the `recvbuffer`'s address; the j^{th} entry is the displacement (or offset) in that array where the data from process j must start. The next parameter is the MPI data type of the data items to be received; usually it is the same as the type being sent, but the data can be converted in the `recvbuffer`. Perhaps the best way to understand the function is from a picture; Figure 8.4 illustrates how the function is used by four processes concatenating their arrays by rank order (first process 0, then 1, then 2 and finally 3.)

Notice that in the figure, the displacement array is just the partial sums of the sizes in the `recvcount` array, meaning that, `displs[j]` is the sum of `recvcount[i]` for all `i < j`. This is a consequence of the fact that the data is being concatenated into the resulting array in ascending order of process rank. The function can be used more generally than this; the array of displacements could be designed to rearrange the locations of the gathered data blocks, if there were some reason to do so. However, for many of the algorithms we will study, the array is a concatenation of the data blocks by process rank, and the `displs` array and `recvcount` array can be initialized simultaneously to be passed to `MPI_Allgatherv`. The following function, `create_communication_arrays`, could be called by each process to construct these arrays for that process.

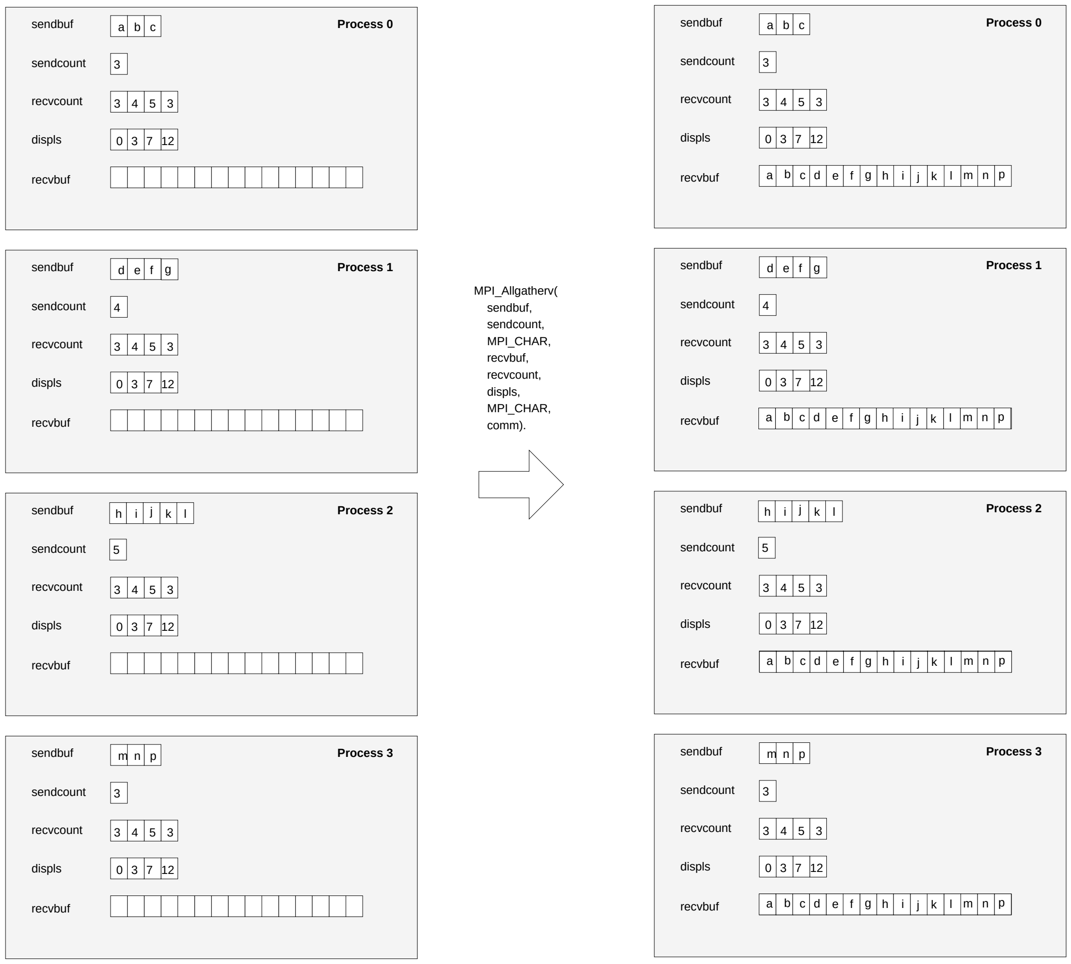


Figure 8.4: Example of four processes using `MPI_Allgatherv` to gather four different size arrays of data into an identical, replicated array in each process. Each process calls the function with the same values in the arrays passed as recvcount and displs, and when the call returns, all processes have the data in the order specified by those two arrays.

This would be an example of redundant code, because every process would call this function with identical parameters, and identical arrays would be created in each process.

The function `create_communication_arrays` expects its two array arguments to have been allocated before it is called.

```
/* Fill in recvcount and displs arrays with values to concatenate in
   rank order in an MPI_Allgatherv (or MPI_Allscatterv ) operation   */
create_communication_arrays (
            int p,           /* number of processes in group         */
            int n            /* total size of array to be gathered   */
            int *recvcount,  /* previously allocated array of n ints */
            int *displs,     /* previously allocated array of n ints */
            )
{
    int i;

    recvcount[0] =  number_of_elements( 0, n, p ); // called size_of_block elsewhere
```

```
        displs[0]    = 0;
        for (i = 1; i < p; i++) {
             displs[i]    = displs[i-1] + recvcount[i-1];
             recvcount[i] = number_of_elements( i, n, p );
    }
```

The function `create_communication_arrays` can be used in a function to collect the blocks from each process's result vector into a vector containing all of their blocks in process rank order, as depicted in Figure 8.3. This function, `replicate_block_vector`, appears in Listing 8.4.

The last step is to print out the matrix and the two vectors. We already have a function that prints out a distributed matrix, which we developed in Chapter 6. It is named `collect_and_print_matrix_byrows`. We need a function that prints a vector. Because each of the two vectors is replicated in every process, the function does not have to gather its pieces from all tasks. It can be a simple function that prints each element of the vector on a single line of an output stream. Because this is a relatively simple function, we omit the code here. We name it `print_vector`.

We have all of the pieces needed for the complete program, which is displayed in Listing 8.5.

8.5.3 Performance Analysis

What is the time complexity of this algorithm and how scalable is it? These are the questions we address. For an $m \times n$ matrix, the sequential algorithm has time complexity $\Theta(mn)$. To simplify the analysis, however, we will assume that the matrix is square, i.e., $m = n$, so that the sequential algorithm has time complexity $\Theta(n^2)$. As a reminder, we do not include the time spent performing I/O, whether it is a sequential or a parallel algorithm; in particular we do not include the overhead of distributing the matrix to each processor, which is consolidated into the function `read_and_distribute_matrix`[2]. Similarly, we exclude the communication overhead of `read_and_replicate_vector,` which is $\Theta(n \log p)$.

Therefore, we start by determining the time complexity of the computational portion of the algorithm. Each task has at most $\lceil n/p \rceil$ rows of the matrix and iteratively computes their inner products with a vector of size n; therefore each task spends time proportional to $n \lceil n/p \rceil$ in computation. Hence, the time complexity of the computational part of the parallel algorithm is

$$\Theta(n^2/p)$$

.

What about the communication time complexity? After each task completes its computation, the result vector is replicated in every task. The time complexity of the all-gather operation depends upon how it is implemented in MPI. If MPI uses the hypercube based approach, then each process sends $\lceil \log p \rceil$ messages, but each is of increasing size. If you look back at the analysis of the all-gather operation in Chapter 3, you will see that the time complexity of the hypercube-based all-gather operation is

$$\lambda \log p + \frac{n\,(p-1)}{\beta p} \tag{8.2}$$

which is $\Theta(\log\, p + n)$. The total time complexity would be

$$\Theta((n^2/p) + \log\, p + n$$

If we assume that p is very small in comparison to n, then n^2/p is much larger than n and this would be the term that dominates the time complexity.

Now we determine the isoefficiency relation of the algorithm to see how scalable it is. As noted above, the time complexity of the sequential algorithm is $\Theta(n^2)$, so $T(n,1) = n^2$. Now we need to determine the parallel overhead $T_0(n,p)$, i.e., the work done by the parallel program not done by the sequential program. The

[2]We worked out in Chapter 6 that the time complexity of `read_and_distribute_matrix` was $\Theta((p-1)\lceil n^2/p \rceil) \approx \Theta(n^2)$ because process $p-1$ sends a message whose average size is $\Theta(\lceil n^2/p \rceil)$ to each of $p-1$ processes, one after the other.

only parallel overhead takes place after each process computes its subblock of the result vector and the final vector has to be assembled using the all-gather operation. Assuming that the matrix is much larger than the number of processors, the time complexity in Eq. 8.2is dominated by the $n(p-1)/\beta p$ term, which is $\Theta(n)$. Since every processor has this overhead term, $T_0(n,p) = np$. The isoefficiency relation

$$T(n,1) \geq C \cdot T_0(n,p)$$

is therefore

$$n^2 \geq Cnp$$

which implies

$$n \geq Cp$$

For problem size n, as we have defined it, the memory storage is $\Theta(n^2)$, so the memory utilization function is $M(n) = n^2$ and the scalability function, $M(f(p))/p$, is

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

This means that to maintain efficiency as the number of processors is increased, the memory per processor would have to grow in proportion to the number of processors, which is not possible for large numbers of processors. In other words, this algorithm is not very scalable.

8.6 Column-Wise Block-Striped Decomposition

An alternative to the row-wise block-striped decomposition of the matrix is a column-wise decomposition. At first this might seem counterintuitive, because the result vector is formed from the inner product of the matrix rows with the input vector, so it seems natural to associate tasks with matrix rows. However, we can view the matrix-vector product in a different way. When we multiply a vector by a matrix on its left, the product vector is the linear combination of the columns of the matrix multiplied by the scalar elements of the vector. In other words, if we let A_j denote the j^{th} column of $m \times n$ matrix A, then the product vector B is also defined by

$$B = \sum_{j=0}^{n-1} X_j A_j$$

where we use X_j to denote the j^{th} component of X. Figure 8.5 depicts this interpretation. This view suggests that a primitive task k can be given a single column A_k of A and component X_k of input vector X and perform the scalar multiplication $X_k A_k$, storing the results into a temporary vector V_k. The i^{th} component of the temporary vector V_k would be the product $X_k A_{ik}$. The problem though is that after each task has performed this scalar-vector multiplication and stored its results in these temporary vectors, none of them are part of the final result vector. Remember that the result vector $B = (B_0, B_1, ..., B_{m-1})^T$ consists of the m sums

$$B_k = X_0 A_{k0} + X_1 A_{k1} + X_2 A_{k2} + \cdots + X_{n-1} A_{kn-1} \quad (0 \leq k < m) \tag{8.3}$$

but each task's temporary result vector V_k has the products

$$X_k A_{0k},\ X_k A_{1k},\ X_k A_{2k}, \ldots,\ X_k A_{m-1k}$$

Of these m products, only one is part of the sum needed in Eq. 8.3, namely $X_k A_{kk}$. What has to happen is that the tasks have to participate in a global data exchange in which each task delivers the results it has just computed to all other tasks. This is an example of an all-to-all collective communication operation. This is just one of several aspects of this solution that make it more complex.

In general, this is a harder algorithm to develop than the row-wise block-striped decomposition, for several reasons:

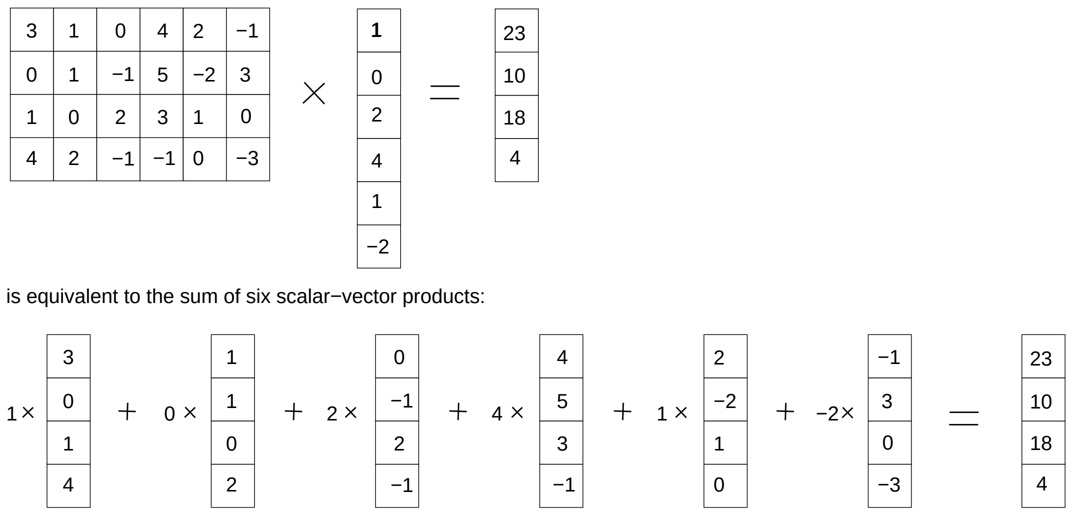


Figure 8.5: Matrix-vector multiplication as a linear combination of the columns of the matrix.

- The result vector has as many rows as the matrix. If the matrix has more rows than columns, then assigning the i^{th}component of the result vector to task i is not enough, because there will be unassigned rows. The result vector components will have to be assigned cyclically.
- Most of the values computed by each task are needed by other tasks and not by the task computing them. As a consequence, a large amount of data must be transferred among all of the tasks, using an all-to-all type of collective communication.
- The matrix is stored in the file in row-major order but must be distributed column-wise. This means that the input algorithm has to be more complex, because it has to scatter each row as it reads it.
- The tasks have columns of the matrix and yet when it has to be printed out, it must be printed row by row, implying that some type of gathering of the data from each task is necessary.

Although there are some lessons to be learned by developing this algorithm, the effort outweighs the benefits and we will explore the various collective communication operations by other examples later. Instead we turn to the third decomposition, the checkerboard approach.

Listing 8.4: replicate_block_vector()

```
/** replicate_block_vector() copies a distributed vector into every process
 *  @param  void         *invec    [IN]   Block-distributed vector
 *  @param  int           n        [IN]   Total number of elements in vector
 *  @param  MPI_Datatype dtype     [IN]   MPI element type
 *  @param  void         *outvec   [OUT]  Replicated vector
 *  @param  MPI_Comm      comm     [IN]   Communicator
 */
void replicate_block_vector (
        void         *invec,
        int           n,
        MPI_Datatype dtype,
        void         *outvec,
        MPI_Comm      comm
        )
{
    int *recv_count;  /* Elements contributed by each process       */
    int *recv_offset; /* Displacement in concatenated array         */
    int id;           /* Process id                                 */
    int p;             /* Number of processes in communicator group */

    MPI_Comm_size (comm, &p);   /* Get number of processes into p   */
    MPI_Comm_rank (comm, &id);  /* Get process rank into id         */

    /* Try to allocate recv_count and recv_offset arrays, and bail out if either
        fails. */
    recv_count  = malloc ( p * sizeof(int));
    recv_offset = malloc ( p * sizeof(int));
    if ( NULL == recv_offset || NULL == recv_count ) {
        printf ("malloc failed for process %d\n", id);
        MPI_Abort (MPI_COMM_WORLD, MALLOC_ERROR);
    }

    /* Fill the count and offset arrays to pass to MPI_Allgatherv so that the
        blocks are concatenated by process rank in the output vector */
    create_communication_arrays (p, n, recv_count, recv_offset);

    /* Use MPI_Allgatherv to copy the distributed blocks from invec in each
        process into a replicated outvec in each process. */
    MPI_Allgatherv (invec, recv_count[id], dtype, outvec, recv_count, recv_offset,
         dtype, comm);

    /* Release the storage for the count and offset arrays. */
    free (recv_count);
    free (recv_offset);
}
```

Listing 8.5: matrix_vector01.c

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include "utils.h"

typedef int Element_type;
#define MPI_TYPE MPI_INT

int main (int argc, char *argv[])
{
    Element_type** A;          /* matrix to be multiplied       */
    Element_type*  A_storage; /* backing storage for matrix    */
    Element_type*  X;          /* vector to left-multiply by A  */
    Element_type*  B;          /* result vector AX=B            */
    Element_type*  B_partial; /* subblock of result vector     */
    int      nrows;            /* number of rows in matrix      */
    int      ncols;            /* number of columns in matrix   */
    int      length;           /* length of vector (=ncols)     */
    int      id;               /* process rank                  */
    int      p;                /* number of processes           */
    int      i, j;             /* loop indices                  */
    int      error;            /* error exit value of calls     */
    char     errstring[127];   /* error message string          */
    int      rows;             /* number of rows on this process*/

    /* Initialize MPI, get rank and number of processes in group. */
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);
    MPI_Comm_size (MPI_COMM_WORLD, &p);

    /* Check command usage and exit if incorrect */
    if ( argc < 3 ) {
        sprintf(errstring, "Usage: %s filename1 filename2\n", argv[0]);
        terminate(id, errstring);
    }

    /* Get the input matrix and distribute its rows among all processes. */
    read_and_distribute_matrix (argv[1],
                                (void *) &A,
                                (void *) &A_storage,
                                MPI_TYPE, &nrows, &ncols, &error,
                                MPI_COMM_WORLD);

    /* Calculate number of rows for this process. */
    rows = size_of_block(id,nrows,p);

    /* Collect the subblocks of the matrix and print it to standard output. */
    collect_and_print_matrix ((void **) A, MPI_TYPE, nrows, ncols,
                  MPI_COMM_WORLD);

    /* Process 0 reads the vector and distributes full vector to all tasks. */
    read_and_replicate_vector  ( argv[2], MPI_TYPE, (void *) &X,
            &length, &error, MPI_COMM_WORLD);

    /* Process 0 prints the full vector on standard output */
    if ( 0 == id )
        print_vector (X,  ncols, MPI_TYPE, stdout );

    /* Every task allocates storage for the full result vector and the piece that
       it computes. */
    B_partial = (Element_type *) malloc (rows * sizeof(Element_type));
    B         = (Element_type *) malloc (nrows * sizeof(Element_type));

    /* And now the real work -- each task computes the inner product of the rows
       it owns and the input vector X. */
    for (i = 0; i < rows; i++) {
        B_partial[i] = 0.0;
        for (j = 0; j < ncols; j++)
             B_partial[i] += A[i][j] * X[j];
    }

   /* The result vector is assembled in each task from the subblocks. */
   replicate_block_vector (B_partial, nrows, MPI_TYPE, (void *) B,
      MPI_COMM_WORLD);

   /* Process 0 prints it. */
   if ( 0 == id )
       print_vector (B, nrows, MPI_TYPE, stdout);

   MPI_Finalize();
   return 0;
}
```

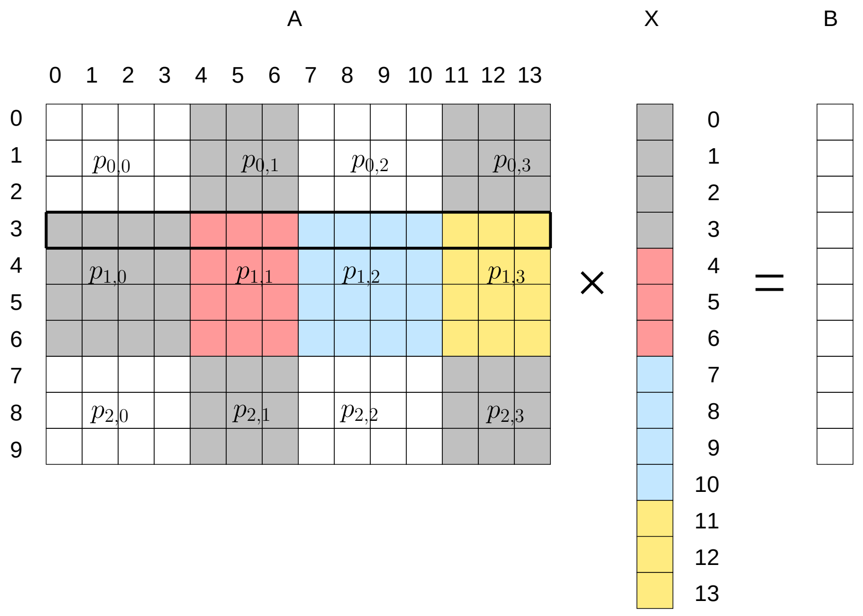


Figure 8.6: Example of the matrix-vector product when the matrix is decomposed into 2D blocks. Rows 3 through 6 of the matrix are color-coded with the parts of the vector that are needed by each subblock. Each row is divided among four processes, each having a unique color. The elements of the vector needed by those processes have the processes' corresponding colors.

8.7 Two-Dimensional Block Decomposition (Checkerboard)

8.7.1 The Decomposition and Related Problems

This will be a more complicated algorithm to understand. In this approach, we decompose the matrix into a grid of rectangular subblocks and assign a process to each subblock. Doing this raises a number of questions, namely

1. which elements of the input vector X are needed by each process;
2. which subblocks contribute to the final values of the elements of the output vector B;
3. how we can distribute the different parts of the input matrix to the processes that own them;
4. how we can distribute the different parts of the input vector to the processes that need them; and
5. how we can collect the partial results computed by each process into the complete result vector that can then be output.

We use an example to reach an answer to these questions. Consider the matrix-vector product illustrated in Figure 8.6. The matrix A has been decomposed into a 3×4 grid of subblocks, not necessarily of the same dimensions. Some blocks are 3×4 whereas some are 3×3 for example. We use the term ***grid*** to refer to the two-dimensional arrangement of subblocks and we will identify the subblocks by their coordinates in the grid using a doubly-subscripted notation. For example, the subblock in row 2 and column 1 of the grid will be called *block[2,1]*. There is a one-to-one correspondence between subblocks and processes; each subblock is owned by a unique process and no process has more than one subblock. Therefore, to make it easy to identify which processes own which subblocks, we will use the same subscripts on the processes as we use for the blocks. For example, process $p_{0,2}$ is the process in grid row 0 and grid column 2 owning *block[0,2]*. Having clarified how we refer to the processes and blocks, we turn to the first question.

1. Which elements of the input vector X are needed by each process? Look at Figure 8.6. There, rows 3 through 6 are color-coded, with the same colors as the input vector. When a row of the matrix, for example row 3, is multiplied by vector X, the gray elements of the row are multiplied by the gray elements of the vector, the red ones are multiplied together, the blue ones together and the yellow one together. Therefore, the processes in the grid that own parts of that row are responsible for the following partial sums:

Process	**Partial sum computed by this process**
$p_{1,0}$	$A_{3,0}X_0 + A_{3,1}X_1 + A_{3,2}X_2 + A_{3,3}X_3$
$p_{1,1}$	$A_{3,4}X_4 + A_{3,5}X_5 + A_{3,6}X_6$
$p_{1,2}$	$A_{3,7}X_7 + A_{3,8}X_8 + A_{3,9}X_9 + A_{3,10}X_{10}$
$p_{1,3}$	$A_{3,11}X_{11} + A_{3,12}X_{12} + A_{3,13}X_{13}$

Notice that the elements of the vector needed by a process correspond to the columns that the process owns. This distribution of the computation is true of every row of the grid. Therefore, we can conclude that, for each process $p_{i,j}$, the process will need to be given the block of the vector that has the same index values as the process's columns. This means that every process in the same grid column needs the same parts of the input vector.

2. Which subblocks contribute to the final values of the elements of the output vector B? By definition, the i^{th} element of the output vector B is the inner product of row i of the matrix with the vector X. Therefore, every process that owns part of row i has a partial sum that must be used to compute the final value of B_i. This suggests that, for each row of the matrix, all processes in the grid row containing that row need to collectively do a reduction on their partial sums of that row onto one of these processes. This implies in turn that all processes in a single grid row must participate in a multiple, parallel reduction of the partial sums of each row that these processes own. For example, processes $p_{1,j}$ for $j = 0, 1, 2, 3$, would each have to call `MPI_Reduce` with partial sums for their subblocks of matrix rows $3, 4, 5$, and 6. The problem is that we do not yet know how a subset of all processes can participate in a reduction but not all processes. This will come soon.

3. How can we distribute the different parts of the input matrix to the processes that own them? There are various ways to do this. The matrix is stored in row-major order in a file, so it needs to be read one row at a time. One process will perform the reading. Ultimately the row needs to be broken up into blocks and delivered to the processes in the grid row to which that matrix row belongs. Neither a single broadcast nor a single scatter operation can selectively send the row's pieces to all of the appropriate processes at once. The easiest algorithm to understand is not the most efficient, but it is the one implemented by Quinn's `read_checkerboard_matrix` function. That algorithm sequentially delivers the blocks to each process of the grid row, one after the other, using point-to-point operations; this takes time proportional to the number of processes in the grid. An alternative is to take advantage of global communication operations to overlap some of the communication.

Suppose process $p_{0,0}$ does the reading of the file. It can read a row of the matrix and determine which grid row the row belongs to. Suppose the row is part of grid row k. Suppose that the grid has c columns. Quinn's approach is to break the row up into blocks and send the blocks to processes $p_{k,0}$, $p_{k,1}$, $p_{k,2}$, and so on up to $p_{k,c-1}$.

An alternative is that $p_{0,0}$ sends the entire row to $p_{k,0}$ and when process $p_{k,0}$ receives the row, it can scatter the row to the processes in that grid row. Because scattering is a global operation, we cannot just make a process scatter to a subset of processes, so there is something else we will have to do to make this happen. Both of these approaches will require MPI features we have not yet learned.

4. How can we distribute the different parts of the input vector to the processes that need them? We can let the first process in the first row, $p_{0,0}$, read the file containing the input vector. The vector needs to be distributed by block to the processes that need it. We determined above that every process in a grid column gets the same part of input vector X. In our example, elements X_0 through X_3 go to processes in the first column, X_4 through X_6 go to processes in the next column, and so on. There are several different ways to do this, but we choose a fairly simple approach. Process $p_{0,0}$ scatters the vector onto the processes in the first row of the grid. Each process gets a piece of the input vector equal to the number of columns that it owns. Now each of these first row processes will broadcast the piece of the vector it received to all processes in its column. When this is finished, every process has the block of the input vector that it needs. Again, the question is how we can perform collective communication operations on subsets of processes, and again the answer is, we will get there shortly.

5. How we can collect the partial results computed by each process into the complete result vector that can then be output? The answer to question 2 indicated that after each process computes the partial sum of the products of the elements in its row with the corresponding elements of the input vector, the processes in each grid row will participate in a reduction to put the final values of the output vector onto the process in column 0 of that row. When that step is finished, all processes in column 0 can participate in a gather operation to put the complete vector onto process $p_{0,0}$, which can then write it to the output stream.

8.7.2 Creating and Using Communicators in MPI

The algorithm we have just described requires the capability of performing various types of collective communication *among subsets of processes* rather than among all processes. In particular,

- Processes in the same grid row need to participate in a sum reduction.
- The I/O process, $p_{0,0}$ needs to scatter data only to the processes in grid row 0.
- Each process in grid row 0 needs to broadcast data only to the processes in its own grid column.
- Processes in column 0 need to gather their data onto the top process, $p_{0,0}$, in that column.
- In distributing the matrix rows, process 0 needs to identify and communicate with processes only in column 0.

Clearly we need a way to arrange for processes to communicate in a grid-oriented way, and to be able to perform collective communications within grid columns and/or rows and point-to-point communications as well.

So far we have only used the `MPI_COMM_WORLD` communicator. This communicator creates a linear ordering of all processes in the program and assigns each process a rank in this ordering. Although we could, if pressed, solve this problem using just this communicator, point-to-point communication, and a great deal of book-keeping within the algorithm, it is far easier to take advantage of the part of the MPI library that provides ***virtual topologies*** for us.

MPI provides functions that allow the specification of virtual process topologies of arbitrary connectivity in terms of graphs in which each vertex corresponds to a process and two vertices are connected if and only if they communicate with each other. The library is powerful enough that graphs of processes can be used to specify any desired topology. However, the most commonly used topologies in message-passing programs are one-, two-, and occasionally, higher-dimensional grids, which are also referred to as ***Cartesian topologies***. For this reason, MPI provides a set of specific functions for specifying and manipulating these types of multi-dimensional virtual grid topologies.

We call them virtual topologies because they are not physically connected to each other, but connected by an opaque layer of software that MPI creates for us. This opaque layer of software is what we have called a communicator so far. It is time to explore communicators in more depth.

A communicator consists of a ***process group***, a set of ***attributes***, and a ***context***. The virtual topology is just one of many attributes of a communicator. A communicator is represented within system memory as an object and is only accessible to the programmer by a ***handle***; `MPI_COMM_WORLD` is actually a handle.

A process group is simply an ordered set of processes. Each process in the group is associated with a unique integer rank. Rank values range from zero to one less than the number of processes in the group. In MPI, a group is represented within system memory as an object. Like communicators, it is only accessible to the programmer by a handle, and is always associated with a communicator object. Although there are special functions for manipulating groups as independent objects, you should think of groups only as parts of a communicator objects. The group functions are primarily used to specify which processes should be used to construct a communicator.

There are some important ideas to remember about communicators:

- They allow you to organize tasks, based upon function, into task groups.
- They enable collective communications operations across a subset of processes.
- They provide the means for implementing user defined virtual topologies.
- They provide the means for safe communications.
- They can be created and destroyed during program execution.
- Processes may be in more than one communicator at a time and will have a unique rank within each group/communicator.
- There are over 60 functions related to groups, communicators, and virtual topologies.

8.7.3 Cartesian Topologies

A Cartesian topology is a topology in which the processes are arranged in a grid of any number of dimensions, and have Cartesian coordinates within that grid. In order to create a Cartesian topology, MPI needs to know the number of dimensions, the size of each dimension, and whether or not, in each dimension, the coordinates wrap around (so that the first and last index values in that dimension are adjacent). In addition, in order to populate the topology with processes, it needs a communicator containing a group or processes, and it needs to know whether it must preserve their ranks or is free to assign new ranks to them if it chooses. The function to create Cartesian topologies is called `MPI_Cart_create`. Its syntax is

```
#include <mpi.h>
int MPI_Cart_create(
    MPI_Comm comm_old,     /* handle to the communicator from which processes come */
    int ndims,             /* number of dimensions in grid                         */
    int *dims,             /* integer array of size ndims of grid dimensions       */
    int *periods,          /* logical array of size ndims of flags (1 = periodic)  */
    int reorder,           /* flag indicating is allowed to reorder processes      */
    MPI_Comm *comm_cart    /* returned handle of new communicator                  */
)
```

This function takes the group of processes that belong to the communicator `comm_old` and creates a virtual Cartesian process topology. The topology information is attached to the new communicator `comm_cart`. Any subsequent MPI functions that want to use this new Cartesian topology must use `comm_cart` as the communicator handle.

Notes

- All of the processes in the `comm_old` communicator must call this function.
- The shape and properties of the topology are specified by the arguments `ndims`, `dims`, and `periods`. The parameter `ndims` specifies the number of dimensions of the topology. The array `dims` specifies the size along each dimension of the topology. The i^{th} element of this array stores the size of the i^{th} dimension of the topology. The array `periods` specifies whether or not the topology has wraparound connections. In particular, if periods[i] is non-zero then the topology has wraparound connections along dimension i, otherwise it does not.

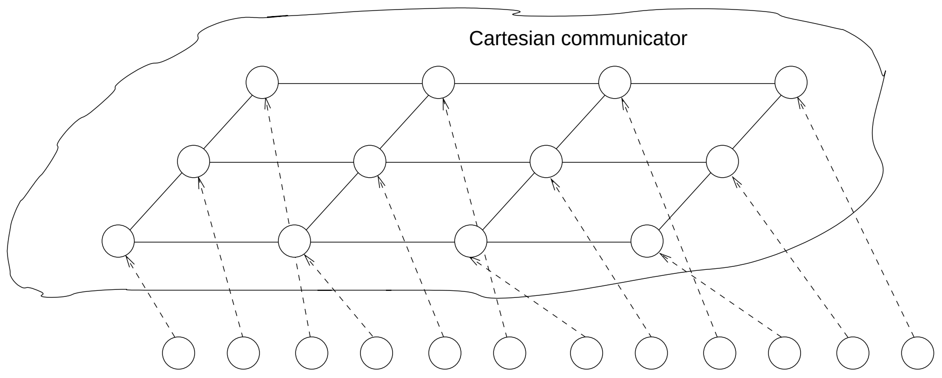


Figure 8.7: Mapping of processes into a 3 × 4 two-dimensional Cartesian topology by a call to `MPI_Cart_create`. It is important to remember that after the call the processes exist in the old communicator as well as the new one; they are not removed from the old one. The Cartesian communicator just makes it possible for them to communicate collectively within a grid topology.

- The `reorder` parameter determines whether the processes in the new group retain their ranks. If reorder is false, then the rank of each process in the new group is identical to its rank in the old group. Otherwise, `MPI_Cart_create` may reorder the processes if that leads to a better embedding of the virtual topology onto the parallel computer.
- If the total number of processes specified in the `dims` array is smaller than the number of processes in the communicator specified by `comm_old`, then some processes will not be part of the Cartesian topology. For this set of processes, when the call returns, the value of `comm_cart` will be set to `MPI_COMM_NULL`.
- It is an error if the total number of processes specified by `dims` is greater than the number of processes in the `comm_old` communicator.

The following code snippet would produce the Cartesian topology shown in Figure 8.7, assuming that there were 12 processes in `MPI_COMM_WORLD` to start.

```
int       ndims       = 2;
int       dims[2]     = {3,4};
int       periodic[2] = {0,0};
MPI_Comm  cartesian_comm;
MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periodic, 1, &cartesian_comm);
```

The reorder parameter is set to true because we do not care whether the ranks in the new communicator are different than they are in the old one.

It will often be the case that we do not know the actual number of processes, as this number is specified at run-time, and that we do not know the shape of the grid until run-time as well. Therefore, if we are trying to decompose a matrix so that the process grid is as square as possible, we would have to do a bit of arithmetic prior to making the call to `MPI_Cart_create`, to determine the best sizes of each dimension. MPI provides a function to do this for us, `MPI_Dims_create`, whose syntax is

```
#include <mpi.h>
int MPI_Dims_create(
    int nnodes,          /* number of nodes in grid                     */
    int ndims,           /* number of dimensions in grid                */
    int *dims,           /* array of size ndims of chosen dimension sizes */
)
```

This function helps the program select a balanced distribution of processes per coordinate direction, depending on the number of processes in the group to be balanced and optional constraints that can be specified. Specifically, the entries in the array `dims` are chosen to define a Cartesian grid with `ndims` dimensions and a total of `nnodes` nodes. The dimensions are set to be as close to each other as possible, using a divisibility algorithm internal to the function. The array `dims[]` may be initialized prior to the call; if `dims[i]` is set to a positive number, the routine will not modify the number of nodes in dimension `i`; only those entries where `dims[i]` $= 0$ are modified by the call.

If, for example, the number of processes is stored in a variable p, and we want to create a virtual two-dimensional grid topology (i.e., a mesh) that is as square as possible, we could use the following code snippet:

```
int       dimensions[2] = {0,0};
int       periodic[2]   = {0,0};
MPI_Comm  cartesian_comm;
MPI_Dims_create(p, 2, dimensions);
MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions, periodic, 1, &cartesian_comm);
```

Now we know how to create a Cartesian topology of the appropriate shape, but to solve some of the typical programming problems, we still need more support from MPI. Suppose that the process that performs the I/O needs to send data to the first process in a particular row, say row k, of the grid. Recall that the `MPI_Send` function needs to be passed the rank of the destination process. This implies that the I/O process needs to know the rank of the process in column 0 and row k of the grid. The question is thus, if we know the Cartesian coordinates of a process, how can we get its rank? This is where the MPI function `MPI_Cart_rank` comes into play. Its syntax is

```
#include <mpi.h>
int MPI_Cart_rank(
    MPI_Comm comm,         /* handle to the communicator                      */
    int *coordinates,      /* integer array of coordinates of calling process  */
    int *rank              /* returned rank of calling process                 */
)
```

We pass the function the coordinates of the process in the virtual grid, and it gives us the process's rank. To get the rank of the process in row k and column 0 of the virtual grid in the communicator `cartesian_comm`, we would use the code

```
int destination_coords[2];
int destination_rank;
destination_coords[0] = k;
destination_coords[1] = 0;
MPI_Cart_rank( cartesian_comm, destination_coords, &destination_rank);
```

It will also be useful for a process that knows its rank to obtain its coordinates in the grid. This is like the inverse to the preceding problem. The `MPI_Cart_coords` function is used for this purpose. Its syntax is

```
#include <mpi.h>
int MPI_Cart_coords(
    MPI_Comm comm,         /* handle to the communicator                      */
    int  rank,             /* rank of calling process                          */
    int  ndims,            /* number of dimensions in the grid                 */
    int  *coordinates,     /* integer array of coordinates of calling process  */
)
```

A process will also need the ability to determine the topology of the grid to which it belongs, i.e., how many rows and columns are in the grid, and perhaps whether wrap is turned on or off in each dimension. There is a special function just to get grid dimensions (`MPI_Cartdim_get`), but if a process also needs its coordinates, it can use the more general-purpose function, `MPI_Cart_get`, whose syntax is

```
#include <mpi.h>
int MPI_Cart_get(
    MPI_Comm comm,          /* handle to the communicator                    */
    int maxdims,            /* number of dimensions (size of arrays to fill)   */
    int *dims,              /* integer array with size of each dimension       */
    int *periods,           /* true/false array of wraparound flags for each dim */
    int *coords             /* coordinates of calling process in the grid      */
)
```

The process supplies the Cartesian communicator handle and the number of dimensions, as well as the addresses of arrays that it has allocated, and the functions fills the arrays with the dimensions, the periodicity of each dimension, and the process's coordinates.

The checkerboard matrix decomposition requires that processes in a given row or a given column perform gather or scatter operations among each other. Assuming that we have created a Cartesian communicator out of the processes in the program, the question is then, how can all of the processes in each of the rows (or columns) participate in a collective communication independently of each other? The answer lies in the ability to form disjoint subsets of the communicator that consist of only the processes in a given row or column. MPI provides us with a very powerful function for this purpose:

```
#include <mpi.h>
int MPI_Comm_split(
    MPI_Comm comm,         /* handle to the communicator                     */
    int color,             /* subgroup identifier                           */
    int key,               /* possibly new rank for calling process          */
    MPI_Comm *newcomm      /* handle to new communicator                     */
)
```

This function partitions the group associated with original communicator `comm` into a collection of disjoint subgroups, one for each value of the parameter `color`. In other words, each process calls the function with a value for the `color` parameter. Those processes that supply the same value for `color` are put into the same subgroup. All of the processes within a subgroup have a new rank in that subgroup. That rank is based on the value of the `key` parameter. If each process in a given subgroup supplies a unique value for `key`, then each will have its new rank equal to `key`. If two or more processes supply the same value for `key`, then `MPI_Comm_split` breaks the tie by assigning ranks to these processes according to their ranks in the old group. For example, if three processes with ranks 5, 8, and 12 in the old group all supply the value 10 for `key`, then the function will pick three distinct unused ranks in the new group and assign them in ascending order to processes 5, 8, and 12 respectively.

A new communicator is created for each subgroup and returned in `newcomm`. A process that does not wish to be placed into any subgroup can opt out by supplying the color value `MPI_UNDEFINED`, in which case `newcomm` stores `MPI_COMM_NULL` for that process. This is a collective call, meaning that every process must call the function, but each process is permitted to provide different values for `color` and `key`.

A few examples will illustrate. Suppose we want to partition the processes in a Cartesian grid so that all processes in the same grid row are in the same subgroup, so that each row of processes can perform its own reduction operation. Then every process can pass the row index to the color parameter. This way every process in a row will be in the same subgroup after the call, and each subgroup will be distinct. If we do not care about the ranks within the new group, we can pass a 0 for the key parameter and let the function assign ranks based on their old ranks. If we want, we can give them a rank equal to their column index in the grid. The code would be as follows.

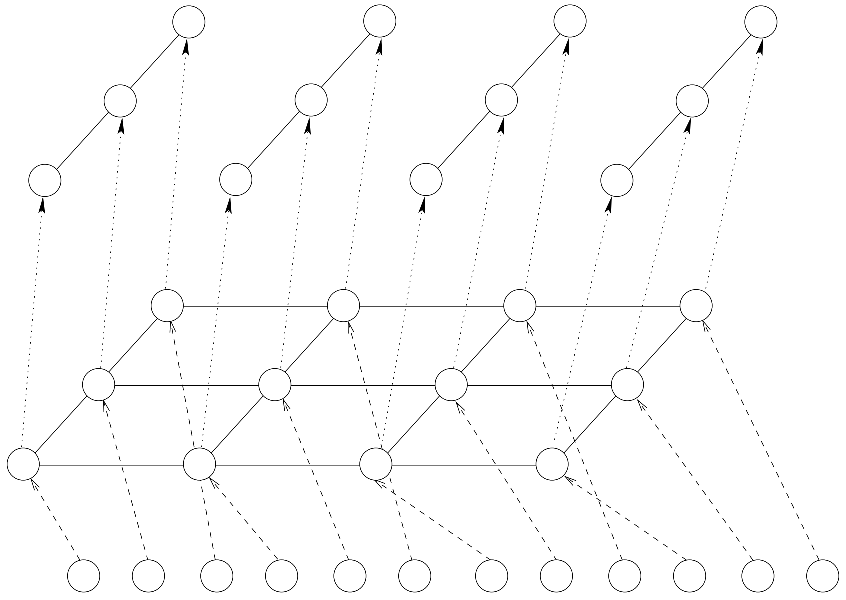


Figure 8.8: The result of splitting a cartesian communicator by columns. After the split, four more communicators exist, and each process is now a part of three different communicator groups, with a rank within each.

```
int       my_rank;                /* rank of process in MPI_COMM_WORLD            */
int       my_cart_rank;           /* rank of process in Cartesian topology        */
int       my_cart_coords[2];      /* coordinates of process in Cartesian topology */
int       dimensions[2] = {0,0};  /* dimensions of Cartesian grid                 */
int       periodic[2]   = {0,0};  /* flags to turn off wrapping in grid           */
MPI_Comm  cartesian_comm;         /* Cartesian communicator handle                */
MPI_Comm  row_comm;               /* Communicator for row subgroup                */

/* Get optimal dimensions for grid  */
MPI_Dims_create(p, 2, dimensions);

/* Create the Cartesian communicator using these dimensions  */
MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions, periodic, 1, &cartesian_comm);

/* Compute for a while ....  */
/* Get rank of process in the Cartesian communicator */
MPI_Comm_rank( cartesian_comm, &my_cart_rank );

/* Use this rank to get coordinates in the grid */
MPI_Cart_coords(cartesian_comm, my_cart_rank, 2, my_cart_coords );

/* Use my_cart_coords[0] as the subgroup id, and my_cart_coords[1] as new rank
   and split off into a group of all processes in the same grid row  */
MPI_Comm_split( cartesian_comm, my_cart_coords[0], my_cart_coords[1], &row_comm);
```

At this point each process has been placed into a subgroup of processes in the same row. Each row can do a collective communication in parallel with all other rows using the newly created communicator, `row_comm`.

With these MPI functions at our disposal, we can return to the design of the algorithm for matrix-vector multiplication using a two-dimensional block decomposition of the matrix.

8.7.4 The Algorithm

The broad steps in the parallel matrix-vector multiplication algorithm are as follows. Remember that every process executes the same program, but to make the meaning clear, the steps indicate whether every process executes the step or just a singled-out process.

1. Every process initializes MPI and stores its rank into `id` and the number of processes into `p`.
2. Every process creates the Cartesian grid, making it as square as possible, with wraparound turned off. Every process must call the function in order to be placed into the Cartesian grid.
3. Every process obtains its coordinates and rank within the Cartesian grid, storing these into `grid_coords[2]` and `grid_id` respectively.
4. Every process creates subgroups of the Cartesian grid by both columns and rows. In particular, each process puts itself into a subgroup consisting of all processes in the same grid column and another consisting of all processes in the same grid row. The handles to these subgroup communicators will be called `col_comm` and `row_comm` respectively.
5. The matrix stored in the input file is read and distributed to each process by subblocks. This step is encapsulated into a function named `read_and_distribute_2dblock_matrix`, which is described below. Every process participates in this step, although it is only one process that opens the file and reads its contents. This function allocates the storage necessary within a process to store its sub-block of the matrix.
6. Although not essential to carry out the multiplication, at this point the matrix is printed to standard output. The function that does this is named `collect_and_print_2dblock_matrix`. An alternative to printing the matrix with a separate function would be to modify `read_and_distribute_2dblock_matrix` so that it prints it while it is reading it. This might be faster, but since in general we need a function that can print a block-decomposed matrix anyway, we develop it for this problem and use it here.
7. Each process determines the number of rows and columns it owns in the block decomposition, storing these into `nlocal_rows` and `nlocal_cols` respectively.
8. Process 0 in row 0 opens the vector file, reads the size of the vector, and allocate memory locally to hold the entire vector. It then reads the vector. If the read was successful, every process computes how much memory it needs for its block of the vector and allocates memory locally for it. Then process 0 in row 0 scatters the vector to all processes in row 0. These processes then broadcast the blocks of the vector they received to all processes in their grid column, because all processes in a given grid column receive the same block of the vector. We could consolidate all of this logic into a function named `read_and_distribute_block_vector`.
9. Every process allocates memory for its local block of the result vector, which will contain the inner products for each row that it owns of the portion of that row that it owns times the portion of the input vector that corresponds to it. For example, if a process manages a subblock of the matrix of dimensions $m \times n$, then its block would contain m elements. Each process also allocates memory of the same size as this one to store the result of the global reduction across the row. The following two instructions take care of this, assuming `nlocal_rows` is the number of rows in the process:

   ```
   B_block = (element_type*) malloc ( nlocal_rows * sizeof(element_type) );
   B_sum   = (element_type*) malloc ( nlocal_rows * sizeof(element_type) );
   ```

10. Every process computes the inner products of the rows of the matrix assigned to this process with the corresponding block of the vector `X`, storing the results in the partial vector `B_block`:

    ```
    for (i = 0; i < nlocal_rows; i++) {
        B_block[i] = 0.0;
        for (j = 0; j < nlocal_cols; j++)
    ```

```
                B_block[i] += A[i][j] * X[j];
        }
```

11. Each row of processes in the grid does a sum reduction across its row. The row communicators created earlier are used in this step. Every process in each row communicator participates in the reduction within that row. The call will be of the form

    ```
    MPI_Reduce(B_block, B_sum, nlocal_rows, mpitype,
               MPI_SUM, rank_of_first_column_process, row_communicator);
    ```

    When this call has returned, the process in the first column of each grid row has the block of the result vector that should be output.

12. Every process in the first column of the grid participates in the printing of the result vector. To ensure that the values are printed in the correct order, the process with rank 0 in that column iteratively requests each other process to send its block, in row order, and prints that block when it receives it. The other processes essentially wait for process 0 to request their data and send it when requested.

13. All processes call `MPI_Finalize`.

This leads to the main program displayed in Listing 8.6 below. Most comments have been removed to save space.

Listing 8.6: Main program for matrix-vector multiplication using 2D blocks.

```
typedef int Element_type;
#define MPI_TYPE MPI_INT
#define REORDER  1

int main (int argc, char *argv[])
{
    int        ncols;            /* total number of columns in input matrix      */
    int        id;               /* process rank in MPI_COMM_WORLD               */
    int        p;                /* number of processes in MPI_COMM_WORLD        */
    int        i, j;             /* loop indices                                 */
    int        error;            /* error exit value of calls                    */
    int        nlocal_rows;      /* number of rows belonging to this process     */
    int        nlocal_cols;      /* number of cols belonging to this process     */
    int        grid_id;          /* rank of process within Cartesian grid        */
    int        row_id;           /* rank of process within its row subgroup      */
    int        grid_size[2];     /* grid dimensions                              */
    int        grid_coords[2];   /* process's coordinates in Cartesian grid      */
    MPI_Comm   grid_comm;        /* Cartesian communicator for grid of procs     */
    MPI_Comm   row_comm;         /* communicator for all processes in a grid row */
    MPI_Comm   col_comm;         /* communicator for all processes in a grid col */
    int       *send_count;       /* array of counts for MPI_Scatterv function    */
    int       *send_offset;      /* array of offsets for MPI_Scatterv function   */
    int        periodic[2];      /* array of wraparound flags when creating grid */
    int        nrows_X;          /* number of rows of X vector local to process  */
    FILE      *file;             /* for open input files                         */

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

    grid_size[0] = 0;            /* Let MPI choose dimensions */
    grid_size[1] = 0;
    MPI_Dims_create (p, 2, grid_size);

```

```
    periodic[0]  = 0;          /* No wraparound  */
    periodic[1]  = 0;          /* No wraparound  */

    MPI_Cart_create (MPI_COMM_WORLD, 2, grid_size, periodic, REORDER, &grid_comm);
    MPI_Comm_rank (grid_comm, &grid_id); /* grid_id is rank in Cartesian grid */
    MPI_Cart_coords (grid_comm, grid_id, 2, grid_coords); /* coordinates in grid
        */

    MPI_Comm_split (grid_comm, grid_coords[0], grid_coords[1], &row_comm);
    MPI_Comm_split (grid_comm, grid_coords[1], grid_coords[0], &col_comm);

    read_and_distribute_2dblock_matrix(argv[1], (void*) &A, (void*) &A_storage,
       MPI_TYPE, &nrows, &ncols, &error, grid_comm);
    if ( 0 != error )
        MPI_Abort(MPI_COMM_WORLD, MP_ERR_IO);

    collect_and_print_2dblock_matrix((void**) A, MPI_TYPE, nrows, ncols, grid_comm
        );

    nlocal_rows = size_of_block(grid_coords[0], nrows, grid_size[0]);
    nlocal_cols = size_of_block(grid_coords[1], ncols, grid_size[1]);

    if ( 0 == grid_coords[0] ) {
        MPI_Comm_rank(row_comm, &row_id);
        if ( 0 == row_id ) { /* we could have also just checked column coord. */
            file = fopen (argv[2], "r");
            if ( NULL == file ) {
                MPI_Abort (MPI_COMM_WORLD, MPI_ERR_FILE );
            }
            else {
                int temp;
                fread (&temp, sizeof(int), 1, file);
                /* Make sure vector length matches matrix width. */
                if ( temp != ncols ) {
                    fprintf(stderr, "Vector size and matrix size unmatched.\n");
                    MPI_Abort(MPI_COMM_WORLD, MPI_ERR_IO);
                }
                input_vec = (Element_type*) malloc(ncols*sizeof(Element_type));
                if ( NULL == input_vec ) {
                    fprintf(stderr, "Could not allocate more storage.\n");
                    MPI_Abort(MPI_COMM_WORLD, MPI_ERR_NO_MEM);
                }
                fread (input_vec, sizeof(Element_type), ncols, file);
            }
        }
    }

    nrows_X = size_of_block (grid_coords[1], ncols, grid_size[1] );
    X_local = (Element_type *) malloc (nrows_X * sizeof(Element_type));
    if (NULL == X_local ) {
        fprintf(stderr, "Could not allocate more storage.\n");
        MPI_Abort(MPI_COMM_WORLD, MPI_ERR_NO_MEM);
    }

    if ( 0 == grid_coords[0] ) {
        if (grid_size[1] > 1) {
            send_count  = (int *) malloc (grid_size[1] * sizeof(int));
            send_offset = (int *) malloc (grid_size[1] * sizeof(int));
            if ( NULL == send_count || NULL == send_offset ) {
                fprintf(stderr, "Could not allocate more storage.\n");
                MPI_Abort(MPI_COMM_WORLD, MPI_ERR_NO_MEM);
            }
            init_communication_arrays(grid_size[1], ncols, send_count, send_offset
                );
            MPI_Scatterv (input_vec, send_count, send_offset, MPI_TYPE, X_local,
                nrows_X, MPI_TYPE, 0, row_comm);
        }
        else
            for (i = 0; i < ncols; i++) X_local[i] = input_vec[i];
    }
    MPI_Bcast (X_local,nrows_X, MPI_TYPE, 0, col_comm);

    B_partial  = (Element_type *) malloc (nlocal_rows * sizeof(Element_type));
    B_sums   = (Element_type *) malloc (nlocal_rows * sizeof(Element_type));
    if (NULL == B_sums || NULL == B_partial ) {
        fprintf(stderr, "Could not allocate more storage.\n");
        MPI_Abort(MPI_COMM_WORLD, MPI_ERR_NO_MEM);
    }

    for (i = 0; i < nlocal_rows; i++) {
        B_partial[i] = 0.0;
        for (j = 0; j < nlocal_cols; j++)
            B_partial[i] += A[i][j] * X_local[j];
    }

    MPI_Reduce(B_partial, B_sums, nlocal_rows, MPI_TYPE, MPI_SUM, 0, row_comm);

    if (grid_coords[1] == 0)
        print_block_vector (B_sums, MPI_TYPE, nrows, col_comm);

    MPI_Finalize();
    return 0;
}
```

We turn to the major components of the algorithm now.

8.7.5 Reading and Distributing a Matrix by Two-Dimensional Subblocks

In Chapter 6, in our parallel version of the Floyd-Warshall algorithm, we saw how to read a matrix and distribute it by rows among the processes in a program when a matrix was decomposed in a row-wise organization. Here, however, we need to distribute subblocks of the input matrix among the processes in the program, and this requires a different algorithm. The function that implements this algorithm has the prototype

```
void read_and_distribute_2dblock_matrix (
          char        *filename,       /* [IN]  name of file to read          */
          void      ***matrix,         /* [OUT] matrix to fill with data      */
          void       **matrix_storage, /* [OUT] linear storage for the matrix */
          MPI_Datatype dtype,          /* [IN]  matrix element type           */
          int         *nrows,          /* [OUT] number of rows in matrix      */
          int         *ncols,          /* [OUT] number of columns in matrix   */
          int         *errval,         /* [OUT] success/error code on return   */
          MPI_Comm     cart_comm)      /* [IN]  communicator handle           */
```

When it has finished, each process in the Cartesian grid defined by the communicator handle p will have its own local subblock of the input matrix, pointed to by the parameter `matrix`, with backing storage pointed to by `matrix_storage`, and it will also have the dimensions of the complete matrix in `nrows` and `ncols`, provided that `errval` was not set to a nonzero value by the function.

The general idea is that process 0 in this communicator group will perform the actual input of the file – opening it, reading the numbers of rows and columns, and reading one row at a time. After process 0 reads the number of rows and columns, it broadcasts these to all other processes. Each process uses its grid coordinates and the total numbers of rows and columns to determine the size of its own subblock and allocates memory for it. Assuming this is successful, the rest of the algorithm requires that process 0 repeatedly read a row of the matrix and send it block by block to the processes that own the subblocks of that row. The detailed steps follow.

1. Every process begins by getting its rank into `id`, the number of processes in the group into `p`, and the element size of the matrix.
2. Process 0 opens the file and reads the number of rows and columns of the matrix, broadcasting these to all processes.
3. Every process then uses `MPI_Cart_get`, passing it `cart_comm` the Cartesian communicator to obtain the grid shape and its coordinates in the grid. It needs these to determine how many rows and columns it has in its subblock.
4. Every process calculates its `nlocal_rows` and `nlocal_cols` and allocates a matrix of this size using the function `alloc_matrix`.
5. Process 0 allocates a buffer large enough to store an entire row of the matrix.
6. For each row of the grid (not the matrix!)
    (a) For each row belonging to the processes in that grid row
        i. Process 0 reads the row from the file
        ii. For each column of the grid, every process calls `MPI_Cart_rank` to get the rank of the process that is associated with the current row and column. Call this rank `destination_rank`. Process 0 calculates that starting address within the buffer of the block of the row that must be sent to the process with rank `destination_rank` as well as the length of that block. If process 0 is not also the process whose rank is `destination_rank`, then it sends the block using `MPI_Send` to that process, and that process calls `MPI_Recv` to receive the block. Otherwise process 0 just copies the block out of the buffer into its local matrix, created in step 4 above.
7. Process 0 frees the buffer that it allocated in step 5 and all processes return from the function.

The function is displayed in Listing 8.7 below, with most comments deleted.

Listing 8.7: read_and_distribute_2dblock_matrix()

```
void read_and_distribute_2dblock_matrix (
          char          *filename,        /* [IN]  name of file to read          */
          void       ***matrix,           /* [OUT] matrix to fill with data      */
          void        **matrix_storage,   /* [OUT] linear storage for the matrix */
          MPI_Datatype dtype,             /* [IN]  matrix element type           */
          int          *nrows,            /* [OUT] number of rows in matrix      */
          int          *ncols,            /* [OUT] number of columns in matrix   */
          int          *errval,           /* [OUT] sucess/error code on return   */
          MPI_Comm       cart_comm)       /* [IN]  communicator handle           */
{
    int     i,j,k;             /* various loop index variables                 */
    int     grid_id;           /* process rank in the cartesian grid           */
    int     p;                 /* number of processes in the cartesian grid    */
    size_t element_size;       /* number of bytes in matrix element type       */
    int     mpi_initialized; /* flag to check if MPI_Init was called already  */
    FILE    *file;           /* input file stream pointer                      */
    int     nlocal_rows;     /* number of rows that calling process "owns"     */
    int     nlocal_cols;     /* number of columns that calling process "owns" */
    MPI_Status   status;     /* result of MPI_Recv call                        */
    int     dest_id;         /* rank of receiving process in cartesian grid    */
    int     grid_coord[2];   /* process coordinates in the grid                */
    int     grid_periodic[2];/* flags indicating if grid wraps around          */
    int     grid_size[2];    /* dimensions of grid                             */
    void*   buffer;          /* address of temp location to store rows         */
    int     block_coord[2];  /* coordinates in grid of current block           */
    void*   source_address;  /* address of block to be sent                    */
    void*   dest_address;    /* location where block is to be received         */

    MPI_Initialized(&mpi_initialized);
    if ( !mpi_initialized ) {
       *errval = -1;
       return;
    }

    MPI_Comm_rank (cart_comm, &grid_id);
    MPI_Comm_size (cart_comm, &p);

    element_size = get_size (dtype);
    if ( element_size <= 0 ) {
       *errval = -1;
       return;
    }

    if ( 0 == grid_id ) {
        file = fopen (filename, "r");
        if ( NULL == file ) {
            *nrows = 0;
            *ncols = 0;
        }
        else { /* successful open */
            fread (nrows, sizeof(int), 1, file);
            fread (ncols, sizeof(int), 1, file);
        }
    }
    MPI_Bcast (nrows, 1, MPI_INT, 0, cart_comm);
    if ( 0 == *nrows  ) {
       *errval = -1;
       return;
    }
    MPI_Bcast (ncols, 1, MPI_INT, 0, cart_comm);

    MPI_Cart_get (cart_comm, 2, grid_size, grid_periodic, grid_coord);
    nlocal_rows = size_of_block( grid_coord[0], *nrows, grid_size[0] );
    nlocal_cols = size_of_block( grid_coord[1], *ncols, grid_size[1] );

    alloc_matrix( nlocal_rows, nlocal_cols, element_size, matrix_storage, matrix,
        errval);
    if ( SUCCESS != *errval )
         MPI_Abort (cart_comm, *errval);

    if ( 0 == grid_id ) {
        buffer = malloc (*ncols * element_size);
        if ( buffer == NULL )
```

```
            MPI_Abort (cart_comm, *errval);
    }

    for (i = 0; i < grid_size[0]; i++) { /* for each grid row */
        block_coord[0] = i;
        for (j = 0; j < size_of_block(i, *nrows, grid_size[0] ); j++) {
            if ( 0 == grid_id ) {
                fread (buffer, element_size, *ncols, file);
            }

            for (k = 0; k < grid_size[1]; k++) {
                block_coord[1] = k;
                MPI_Cart_rank (cart_comm, block_coord, &dest_id);

                if ( 0 == grid_id ) {
                    source_address = buffer +  ( (k*(*ncols))/grid_size[1] ) *
                        element_size;
                    if (0 == dest_id ) {
                        dest_address = (*matrix)[j];
                        memcpy (dest_address, source_address,nlocal_cols *
                            element_size);
                    }
                    else {
                        int blocksize = size_of_block(k,*ncols, grid_size[1]);
                        MPI_Send (source_address,blocksize, dtype, dest_id, 0,
                            cart_comm);
                    }
                }
                else if (grid_id == dest_id) {
                    MPI_Recv ((*matrix)[j], nlocal_cols, dtype, 0, 0, cart_comm, &
                        status);
                }
            } /* end for k */
        } /* end for j */
    } /* for i */

    if (grid_id == 0)
        free (buffer);
    *errval = 0;
}
```

The function to print the matrix is displayed in Listing 8.8 but not explained here.

8.7.6 Performance Analysis

We again ask the questions, “what is the time complexity of this algorithm and how scalable is it?” To simplify the analysis, we will assume again that the matrix is square, i.e., it is $n \times n$, so that the sequential algorithm has time complexity $\Theta(n^2)$. We will also assume for simplicity that the number of processes, p, is a square number, so that the grid is $\sqrt{p} \times \sqrt{p}$ and that each subblock is of size at most $\lceil n/\sqrt{p} \rceil \times \lceil n/\sqrt{p} \rceil$. Although we will not count in our analysis the communication time associated with reading and distributing the matrix and the vector to each processor, we will determine what that time is.

The algorithm to read and distribute the matrix takes each of n rows of the matrix and sends it by subblocks to the processes that own those blocks. As there are $\sqrt{p}$ subblocks in each row, and there are n rows, there are a total of $\lceil n\sqrt{p} \rceil$ messages sent, each of size $\lceil n/\sqrt{p} \rceil$. The communication time is therefore $\lceil n\sqrt{p} \rceil (\lambda + \lceil n/\sqrt{p} \rceil /\beta)$, or approximately $n\sqrt{p}\lambda + n^2/\beta$, which is $\Theta(n^2)$. This is as expected, as the matrix contains n^2 elements that must be delivered to all processes.

The algorithm that distributes the vector begins by scattering the vector to $\sqrt{p}$ processes in the first row.

Assuming a hypercube algorithm is used and that the average message size is $n/\sqrt{p}$, the overhead of this step is $\log\sqrt{p}\lambda + n\log\sqrt{p}/(\sqrt{p}\beta)$. Since $\log\sqrt{p} = (\log p)/2$, this is $\Theta((n\log p)/\sqrt{p})$. After this step each process in the top row simultaneously broadcasts its block to all processes in its column. This step takes the same time as the scattering operation, as it is a broadcast of a message of size $n/\sqrt{p}$ to $\sqrt{p}$ processes, so the overall time complexity is $\Theta((n\log p)/\sqrt{p})$.

The time complexity of the computational portion of the algorithm is the time spent computing the partial inner products. Each task computes the matrix product of an approximately $n/\sqrt{p} \times n/\sqrt{p}$ matrix with a vector of size $n\sqrt{p}$ and therefore takes $\Theta(n^2/p)$ steps to perform the computation.

The time complexity of the communication overhead not associated with input of the matrix or the vector is the time that all processes take when participating in the simultaneous row reductions of their partial sums of the result vector. There are $\sqrt{p}$ processes in each row, and the message size is $n/\sqrt{p}$. This is exactly like the time to broadcast the input vector along a column, so each sum reduction has a complexity of $\Theta((n\log p)/\sqrt{p})$.

The total time to perform the matrix-vector multiplication using the two-dimensional block decomposition is therefore

$$\Theta(n^2/p + (n\log p)/\sqrt{p}).$$

We now determine the isoefficiency and scalability of this parallel system. The sequential algorithm has time complexity $\Theta(n^2)$, so $T(n,1) = n^2$. The parallel overhead of this algorithm, $T_0(n,p)$, is the work not done by the sequential algorithm and is limited to the total communication time (excluding the I/O time.) Since p processes perform the row reduction simultaneously, the communication overhead is p times the time complexity of the sum reduction, which is therefore $\Theta(p \times (n\log p)/\sqrt{p}) = \Theta((n\sqrt{p}\log p))$, so $T_0(n,p) = n\sqrt{p}\log p$. The isoefficiency relation

$$T(n,1) \geq C \cdot T_0(n,p)$$

is therefore

$$n^2 \geq Cn\sqrt{p}\log p$$

which implies

$$n \geq C\sqrt{p}\log p$$

For problem size n, as we have defined it, the memory storage is $\Theta(n^2)$, so the memory utilization function is $M(n) = n^2$ and the scalability function, $M(f(p))/p$, is

$$M(Cp)/p = (C\sqrt{p}\log p)^2/p = C^2(p\log^2 p)/p = C^2\log^2 p$$

This means that to maintain efficiency as the number of processors is increased, the memory per processor would have to grow in proportion to the square of the logarithm of the number of processors. This is a relatively slow-growing function, so this algorithm is fairly scalable, certainly much more scalable than either the row-wise striped or column-wise striped algorithm.

Listing 8.8: collect_and_print_2dblock_matrix()

```
void collect_and_print_2dblock_matrix (
    void         **a,            /* IN -2D matrix */
    MPI_Datatype dtype,          /* IN -Matrix element type */
    int            m,            /* IN -Matrix rows */
    int            n,            /* IN -Matrix columns */
    MPI_Comm       grid_comm)    /* IN - Communicator */
{
    void        *buffer;         /* Room to hold 1 matrix row */
    int          coords[2];      /* Grid coords of process
                                    sending elements */
    int          element_size;    /* Bytes per matrix element */
    int          els;            /* Elements received */
    int          grid_coords[2]; /* Coords of this process */
```

```
   int        grid_id;        /* Process rank in grid */
   int        grid_period[2]; /* Wraparound */
   int        grid_size[2];   /* Dims of process grid */
   int        i, j, k;
   void      *laddr;          /* Where to put subrow */
   int        local_cols;     /* Matrix cols on this proc */
   int        p;              /* Number of processes */
   int        src;            /* ID of proc with subrow */
   MPI_Status status;         /* Result of receive */

   MPI_Comm_rank (grid_comm, &grid_id);
   MPI_Comm_size (grid_comm, &p);
   element_size = get_size (dtype);

   MPI_Cart_get (grid_comm, 2, grid_size, grid_period,
      grid_coords);
   local_cols = size_of_block(grid_coords[1], n, grid_size[1]);

   if (0 == grid_id)
      buffer = malloc ( n * element_size);

   /* For each row of the process grid */
   for (i = 0; i < grid_size[0]; i++) {
      coords[0] = i;

      /* For each matrix row controlled by the process row */
      for (j = 0; j < size_of_block(i,m, grid_size[0] ); j++) {

         /* Collect the matrix row on grid process 0 and
            print it */
         if (0 == grid_id) {
            for (k = 0; k < grid_size[1]; k++) {
               coords[1] = k;
               MPI_Cart_rank (grid_comm, coords, &src);
               els = size_of_block(k,n, grid_size[1]);
               laddr = buffer +
                  ((k*n)/grid_size[1]) * element_size;
               if (src == 0) {
                  memcpy (laddr, a[j], els * element_size);
               } else {
                  MPI_Recv(laddr, els, dtype, src, 0,
                     grid_comm, &status);
               }
            }
            print_vector (buffer, n, dtype, stdout);
            printf ("\n");
         }
         else if (grid_coords[0] == i) {
            MPI_Send (a[j], local_cols, dtype, 0, 0,
               grid_comm);
         }
      }
   }
   if (0 == grid_id) {
      free (buffer);
      printf ("\n");
   }
}
```

References

[1] M.J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Higher Education. McGraw-Hill Higher Education, 2004.

Subject Index