COLLECTIVE COMMUNICATION

- Consider our trapezoidal rule program, we can find several things that we might be able to improve on. One of the most obvious is that the "global sum" after each process has computed its part of the integral.

- If we hire eight workers to, say, build a house, we might feel that we weren't getting our money's worth if seven of the workers told the first what to do, and then the seven collected their pay and went home.

- But this is very similar to what we're doing in our global sum. Each process with rank greater than 0 is "telling process 0 what to do" and then quitting. That is, each process with rank greater than 0 is, in effect, saying "add this number into the total." Process 0 is doing nearly all the work in computing the global sum, while the other processes are doing almost nothing.

- Sometimes it does happen that this is the best we can do in a parallel program, but if we imagine that we have eight students, each of whom has a number, and we want to find the sum of all eight numbers, we can certainly come up with a more equitable distribution of the work than having seven of the eight give their numbers to one of the students and having the first do the addition.

1. Tree-structured communication

- We might use a "binary tree structure" like shown in Fig6. In this diagram, initially students or processes 1, 3, 5, and 7 send their values to processes 0, 2, 4, and 6, respectively. Then processes 0, 2, 4, and 6 add the received values to their original values, and the process is repeated twice:
-

1. a. Processes 2 and 6 send their new values to processes 0 and 4, respectively.
   b. Processes 0 and 4 add the received values into their new values.

2. a. Process 4 sends its newest value to process 0.
   b. Process 0 adds the received value to its newest value.

- This solution may not seem ideal, since half the processes (1, 3, 5, and 7) are doing the same amount of work that they did in the original scheme. The original scheme required ***comm_sz-1*** = seven receives and seven adds by process 0, while the new scheme only requires three, and all the other processes do no more than two receives and adds.
- Furthermore, the new scheme has a property by which a lot of the work is done concurrently by different processes.

- For example, in the first phase, the receives and adds by processes 0, 2, 4, and 6 can all take place simultaneously. So, if the processes start at roughly

the same time, the total time required to compute the global sum will be the time required by process 0, that is, three receives and three additions. We've thus reduced the overall time by more than 50%.

- Furthermore, if we use more processes, we can do even better. For example, if comm_sz =1024, then the original scheme requires process 0 to do 1023 receives and additions that the new scheme requires process 0 to do only 10 receives and additions. This improves the original scheme by more than a factor of 100!

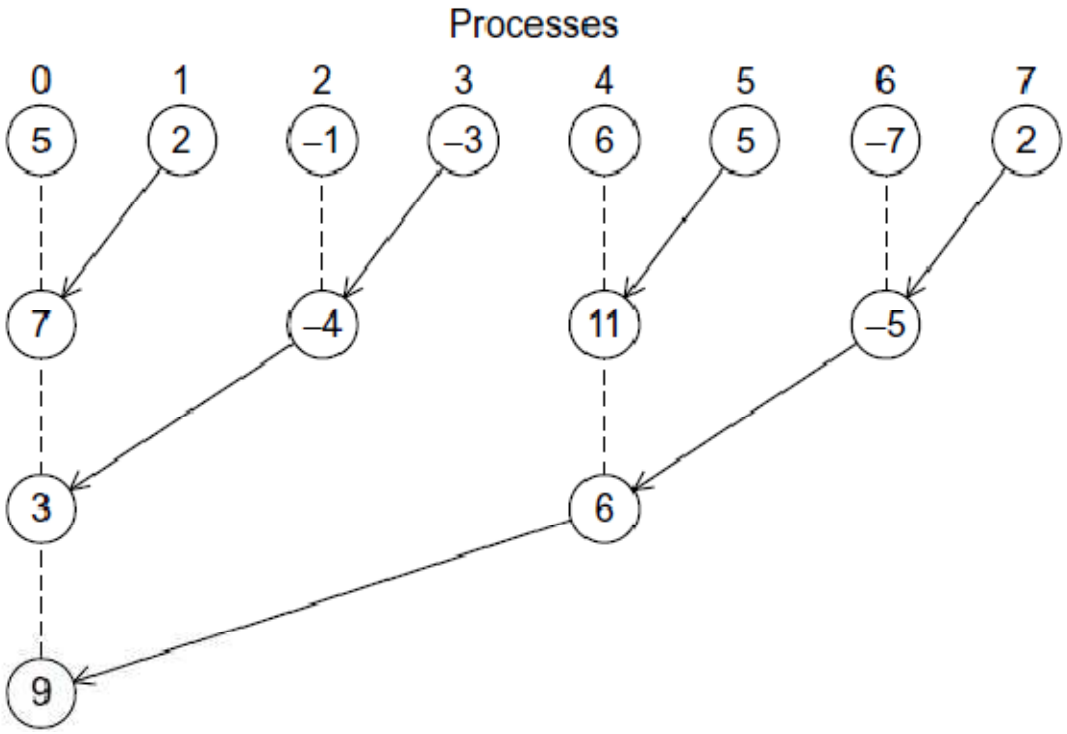

Fig 6:A tree-structured global sum

- The tree structured global sum would take a quite a bit of work.

- It's perfectly feasible to construct a tree-structured global sum that uses different "process-pairings." For example, we might pair 0 and 4, 1 and 5, 2 and 6, and 3 and 7 in the first phase. Then we could pair 0 and 2, and 1 and 3 in the second, and 0 and 1 in the final.
- See Fig7. There are many other possibilities. If we do, is it possible that one method works best for "small" trees, while another works best for "large" trees? Even worse, one approach might work best on system A, while another might work best on system B.

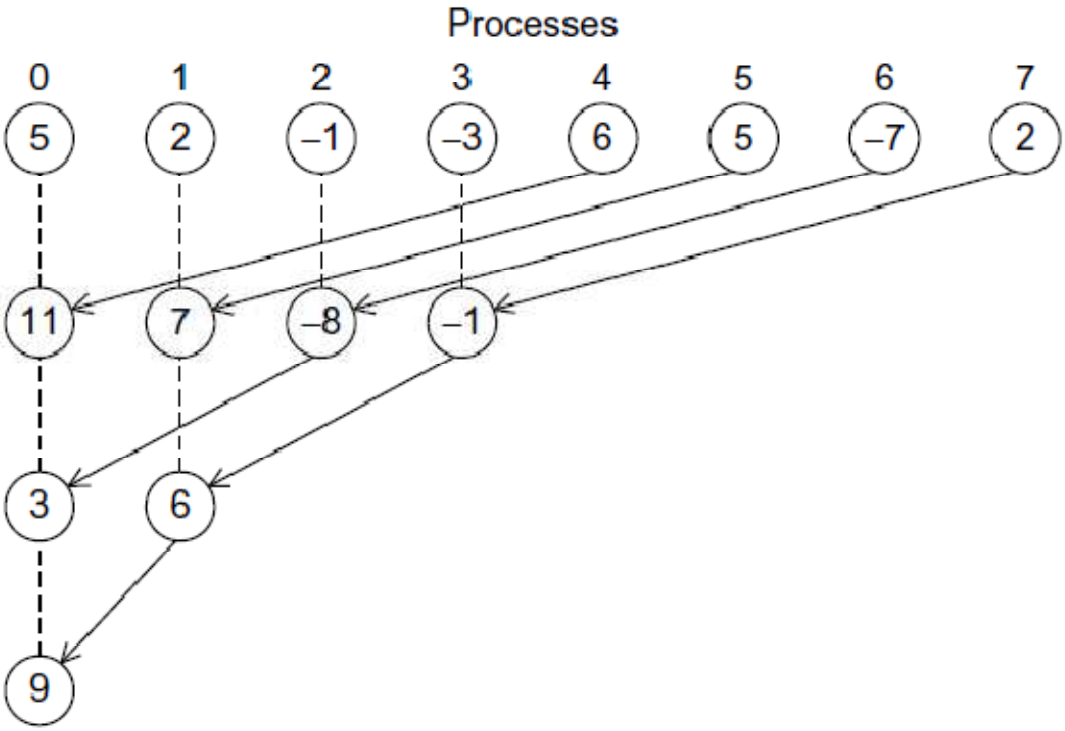

Fig 7:An alternative tree-structured global sum

2. MPI_Reduce

- The developer of the MPI implementation should know enough about both the hardware and the system software so that she can make better decisions about implementation details.

- A “global-sum function” will obviously require communication. However, unlike the MPI_Send – MPI_Recv pair, the global-sum function may involve more than two processes. In fact, in our trapezoidal rule program it will involve all the processes in MPI COMM WORLD. In MPI parlance, communication functions that involve all the processes in a communicator are called collective communications.

- To distinguish between collective communications and functions such as MPI_Send and MPI_Recv, **MPI_Send and MPI_Recv are often called point-to-point communications.**
- In fact, global sum is just a special case of an entire class of collective communications. For example, it might happen that instead of finding the sum of a collection of Comm._sz numbers distributed among the processes, we want to find the maximum or the minimum or the product or any one of many other possibilities.

- MPI generalized the global-sum function so that any one of these possibilities can be implemented with a single function:

```
int MPI_Reduce(
      void*         input_data_p   /* in  */,
      void*         output_data_p  /* out */,
      int           count          /* in  */,
      MPI_Datatype  datatype       /* in  */,
      MPI_Op        operator       /* in  */,
      int           dest_process   /* in  */,
      MPI_Comm      comm           /* in  */);
```

- The key to the generalization is the fifth argument, ***operator***. It has type MPI_Op, which is a predefined MPI type like MPI_Datatype and MPI_Comm. There are a number of predefined values in this type.

Table 3.2 Predefined Reduction Operators in MPI

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

- The operator we want is MPI SUM. Using this value for the operator argument, we can replace the code in Lines 18 through 28 of Program 3.2 with the single function call

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,
    MPI_COMM_WORLD);
```

- One point worth noting is that by using a count argument greater than 1, MPI_Reduce can operate on arrays instead of scalars. The following code could thus be used to add a collection of N-dimensional vectors, one per process:

```
double local_x[N], sum[N];
. . .
MPI_Reduce(local x, sum, N, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
```

3. Collective vs. point-to-point communications

- It's important to remember that collective communications differ in several ways from point-to-point communications:

  1. ***All*** the processes in the communicator must call the same collective function.
     For example, a program that attempts to match a call to ***MPI_Reduce*** on one process with a call to ***MPI_Recv*** on another process is erroneous, and, in all likelihood, the program will hang or crash.

  2. The arguments passed by each process to an MPI collective communication must be "compatible."
     For example, if one process passes in 0 as the dest_process and another passes in 1, then the outcome of a call to MPI_Reduce is erroneous, and, once again, the program is likely to hang or crash.

  3. The ***output_data_p*** argument is only used on ***dest_process***. However, all of the processes still need to pass in an actual argument corresponding to output data p, even if it's just NULL.

  4. Point-to-point communications are matched on the basis of tags and communicators. Collective communications don't use tags, so they're matched solely on the basis of the communicator and the order in which they're called.

4. MPI_Allreduce

- In our trapezoidal rule program, we just print the result, so it's perfectly natural for only one process to get the result of the global sum. However, it's not difficult to imagine a situation in which all of the processes need the result of a global sum in order to complete some larger computation. In this situation, we encounter some of the same problems we encountered with our original global sum. For example, if we use a tree to compute a global sum, we might "reverse" the branches to distribute the global sum (see Figure 8).

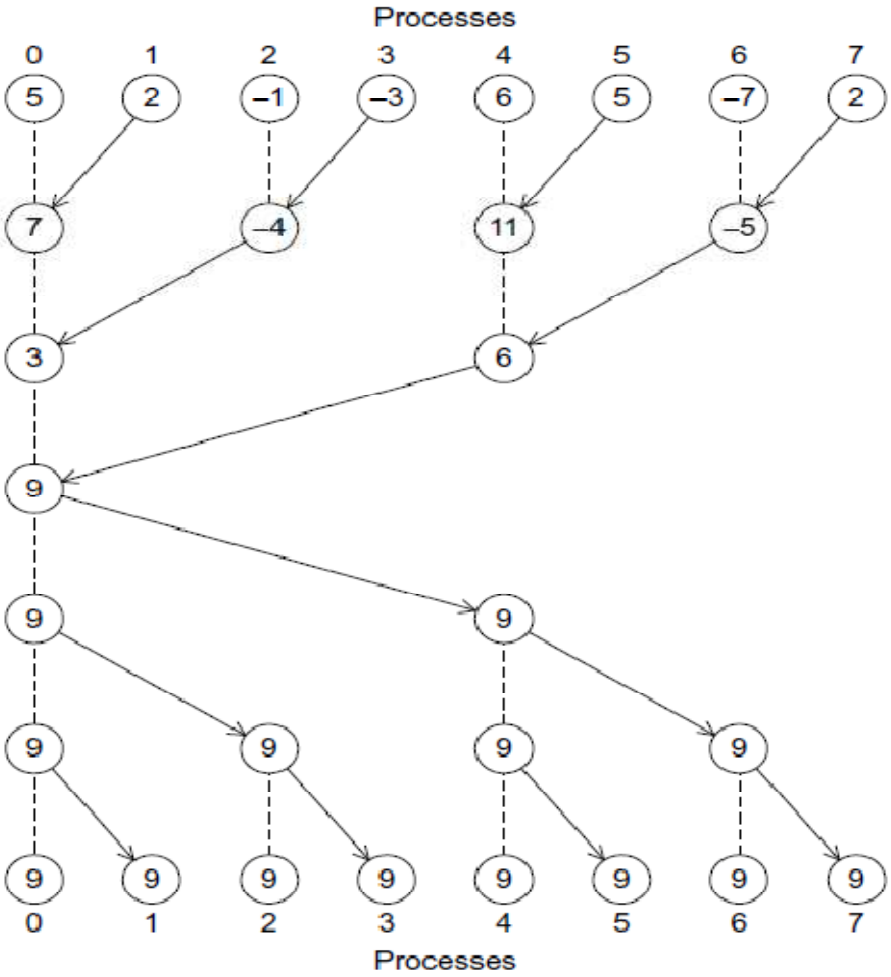

Fig 8. A global sum followed by distribution of the result

- Alternatively, we might have the processes exchange partial results instead of using one-way communications. Such a communication pattern is sometimes called a butterfly (see Figure 9). Once again, we don't want to have to decide

on which structure to use, or how to code it for optimal performance.

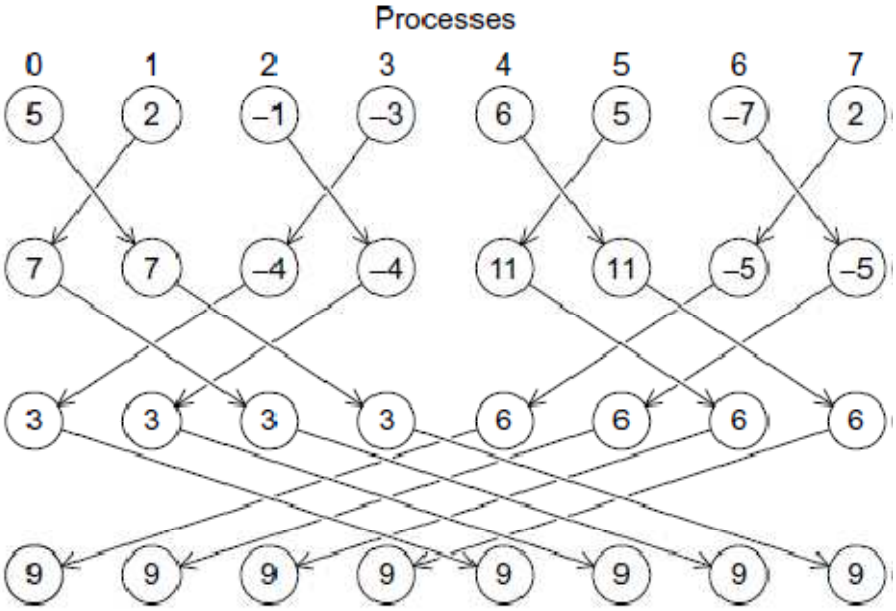

FIGURE 9 A butterfly-structured global sum

- MPI provides a variant of MPI_Reduce that will store the result on all the processes in the communicator:

```
int MPI_Allreduce(
      void*         input_data_p   /* in  */,
      void*         output_data_p  /* out */,
      int           count          /* in  */,
      MPI_Datatype  datatype       /* in  */,
      MPI_Op        operator       /* in  */,
      MPI_Comm      comm           /* in  */);
```

- The argument list is identical to that for MPI_Reduce except that there is no dest _process since all the processes should get the result.

5. Broadcast

- A collective communication in which data belonging to a single process is sent to all of the processes in the communicator is called a broadcast, and you've probably guessed that MPI provides a broadcast function:

```
int MPI_Bcast(
      void*         data_p         /* in/out */,
      int           count          /* in     */,
      MPI_Datatype  datatype       /* in     */,
      int           source_proc    /* in     */,
      MPI_Comm      comm           /* in     */);
```

- The process with rank ***source_proc*** sends the contents of the memory referenced by ***data_p*** to all the processes in the communicator ***comm.***

```
1  void Get_input(
2        int      my_rank  /* in  */,
3        int      comm_sz  /* in  */,
4        double*  a_p      /* out */,
5        double*  b_p      /* out */,
6        int*     n_p      /* out */) {
7
8     if (my_rank == 0) {
9        printf("Enter a, b, and n\n");
10       scanf("%lf %lf %d", a_p, b_p, n_p);
11    }
12    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
13    MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
14    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
15 }  /* Get_input */
```

Program : A version of Get_input that uses MPI_Bcast

- In serial programs, an in/out argument is one whose value is both used and changed by the function.

- For MPI_Bcast, however, the data_p argument is an input argument on the process with rank source_proc and an output argument on the other processes. Thus, when an argument to a collective communication is labeled in/out, it's possible that it's an input argument on some processes and an output argument on other processes.

6. Data distributions

- Suppose we want to write a function that computes a vector sum:

$$\begin{aligned}\mathbf{x}+\mathbf{y} &= (x_0, x_1, \ldots, x_{n-1}) + (y_0, y_1, \ldots, y_{n-1}) \\ &= (x_0 + y_0, x_1 + y_1, \ldots, x_{n-1} + y_{n-1}) \\ &= (z_0, z_1, \ldots, z_{n-1}) \\ &= \mathbf{z}\end{aligned}$$

- If we implement the vectors as arrays of, say, doubles, we could implement serial vector addition with the code shown in Program 7.

```
1  void Vector_sum(double x[], double y[], double z[], int n) {
2     int i;
3
4     for (i = 0; i < n; i++)
5        z[i] = x[i] + y[i];
6  }  /* Vector_sum */
```

Program 7: A serial implementation of vector addition

- The work consists of adding the individual components of the vectors, so we might specify that the tasks are just the additions of corresponding components. Then there is no communication between the tasks, and the problem of parallelizing vector addition boils down to aggregating the tasks and assigning them to the cores.

- If the number of components is n and we have comm_sz cores or processes, let's assume that n evenly divides comm_sz and define local_ n= n/ comm_sz. Then we can simply assign blocks of local_n consecutive components to each process.

- The four columns on the left of Table 4 show an example when n = 12 and comm._ sz =3. This is often called a ***block partition*** of the vector.

- An alternative to a block partition is a ***cyclic partition***. In a ***cyclic partition***, we assign the components in a round robin fashion. The four columns in the middle of Table 4 show an example when n =12 and comm._sz =3. Process 0 gets component 0, process 1 gets component 1, process 2 gets component 2, process 0 gets component 3, and so on.

- A third alternative is a ***block-cyclic partition***. The idea here is that instead of using a cyclic distribution of individual components, we use a cyclic distribution
  of blocks of components, so a block-cyclic distribution isn't fully specified until we decide how large the blocks are. If comm._sz= 3, n =12, and the blocksize b= 2, an example is shown in the four columns on the right of Table 4.

	Components											
									Block-Cyclic			
Process	Block				Cyclic				Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	+6'	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

Table 4 Different Partitions of a 12-Component Vector among Three Processes

- Once we've decided how to partition the vectors, it's easy to write a parallel vector addition function: each process simply adds its assigned components. Furthermore, regardless of the partition, each process will have ***local_n*** components of the vector, and, in order to save on storage, we can just store these on each process as an array of ***local_n*** elements. Each process will execute the function shown in Program 8.

```
1  void Parallel_vector_sum(
2        double  local_x[]  /* in  */,
3        double  local_y[]  /* in  */,
4        double  local_z[]  /* out */,
5        int     local_n    /* in  */) {
6     int local_i;
7
8     for (local_i = 0; local_i < local_n; local_i++)
9        local_z[local_i] = local_x[local_i] + local_y[local_i];
10 }  /* Parallel_vector_sum */
```

Program 8: A parallel implementation of vector addition

7. Scatter

- Now suppose we want to test our vector addition function. It would be convenient to be able to read the dimension of the vectors and then read in the vectors x and

- To read in the dimension of the vectors: process 0 can prompt the user, read in the value, and broadcast the value to the other processes. Process 0 could read them in and broadcast them to the other processes.

- If there are 10 processes and the vectors have 10,000 components, then each process will need to allocate storage for vectors with 10,000 components, when it is only operating on subvectors with 1000 components.

- If, for example, we use a block distribution, it would be better if process 0 sent only components 1000 to 1999 to process 1, components 2000 to 2999 to process 2, and so on. Using this approach, processes 1 to 9 would only need to allocate storage for the components they're actually using.

- For the communication MPI provides just such a function:

```
int MPI_Scatter(
      void*          send_buf_p  /* in  */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      int            src_proc    /* in  */,
      MPI_Comm       comm        /* in  */);
```

- If the communicator ***comm*** contains ***comm_sz*** processes, then MPI Scatter divides the data referenced by send buf p into ***comm_sz*** pieces—the first

piece goes to process 0, the second to process 1, the third to process 2, and so on.

- For example, suppose we're using a block distribution and process 0 has read in all of an n-component vector into ***send_buf_p***. Then, process 0 will get the first ***local_n*** = n/ ***comm_sz*** components, process 1 will get the next ***local_ n*** components, and so on.

- Each process should pass its local vector as the ***recv_buf_p*** argument and the ***recv_count*** argument should be ***local_n***. Both send type and ***recv_type*** should be ***MPI_DOUBLE*** and ***src_ proc*** should be 0.

- The ***send_count*** should also be ***local_n*** — ***send_count*** is the amount of data going to each process; it's not the amount of data in the memory referred to by ***send_ buf_ p***. If we use a block distribution and ***MPI_ Scatter***, we can read in a vector using the function Read vector shown in Program 9.

- One point to note here is that MPI_Scatter sends the first block of ***send_count*** objects to process 0, the next block of ***send_count*** objects to process 1, and so on, so this approach to reading and distributing the input vectors will only be suitable if we're using a block distribution and ***n***, the number of components in the vectors, is evenly divisible by ***comm_sz***.

```
1  void Read_vector(
2        double    local_a[]   /* out */,
3        int       local_n     /* in  */,
4        int       n           /* in  */,
5        char      vec_name[]  /* in  */,
6        int       my_rank     /* in  */,
7        MPI_Comm  comm        /* in  */) {
8
9     double* a = NULL;
10    int i;
11
12    if (my_rank == 0) {
13       a = malloc(n*sizeof(double));
14       printf("Enter the vector %s\n", vec_name);
15       for (i = 0; i < n; i++)
16          scanf("%lf", &a[i]);
17       MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
18             MPI_DOUBLE, 0, comm);
19       free(a);
20    } else {
21       MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
22             MPI_DOUBLE, 0, comm);
23    }
24 }  /* Read_vector */
```

Program 9: A function for reading and distributing a vector

8. Gather

- The test program will be useless unless we can see the result of our vector addition, so we need to write a function for printing out a distributed vector. Our function can collect all of the components of the vector onto process 0, and then process 0 can print all of the components. The communication in this function can be carried out by ***MPI_Gather.***

```
int MPI_Gather(
      void*         send_buf_p   /* in  */,
      int           send_count   /* in  */,
      MPI_Datatype  send_type    /* in  */,
      void*         recv_buf_p   /* out */,
      int           recv_count   /* in  */,
      MPI_Datatype  recv_type    /* in  */,
      int           dest_proc    /* in  */,
      MPI_Comm      comm         /* in  */);
```

- The data stored in the memory referred to by ***send_buf_p*** on process 0 is stored in the first block in recv buf p, the data stored in the memory referred to by ***send_buf_p*** on process 1 is stored in the second block referred to by ***recv_buf_ p***, and so on.

- So, if we're using a block distribution, we can implement our distributed vector print function as shown in Program 10. Note that ***recv_count*** is the number of data items received from each process, not the total number of data items received.

- The restrictions on the use of MPI_Gather are similar to those on the use of MPI_Scatter: our print function will only work correctly with vectors using a block distribution in which each block has the same size.

```
1  void Print_vector(
2        double    local_b[]  /* in */,
3        int       local_n    /* in */,
4        int       n          /* in */,
5        char      title[]    /* in */,
6        int       my_rank    /* in */,
7        MPI_Comm  comm       /* in */) {
8
9     double* b = NULL;
10    int i;
11
12    if (my_rank == 0) {
13       b = malloc(n*sizeof(double));
14       MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n,
15             MPI_DOUBLE, 0, comm);
16       printf("%s\n", title);
17       for (i = 0; i < n; i++)
18          printf("%f ", b[i]);
19       printf("\n");
20       free(b);
21    } else {
22       MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n,
23             MPI_DOUBLE, 0, comm);
24    }
25 }  /* Print_vector */
```

Program 10: A function for printing a distributed vector

9. Allgather

- if ***A = (aij)*** is an ***mxn*** matrix and ***x*** is a vector with ***n*** components, then ***y = Ax*** is a vector with ***m*** components and we can find the ***ith*** component of ***y*** by forming the dot product of the ***ith*** row of ***A*** with ***x:***

$$y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \cdots a_{i,n-1}x_{n-1}.$$

- The pseudo-code for serial matrix multiplication as follows:

```
/* For each row of A */
for (i = 0; i < m; i++) {
   /* Form dot product of ith row with x */
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i][j]*x[j];
}
```

$$\begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\ \vdots \\ y_{m-1} \end{pmatrix}$$

FIGURE 11 Matrix-vector multiplication

- For example, the two-dimensional array:

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix}$$

- would be stored as the one-dimensional array

0 1 2 3 4 5 6 7 8 9 10 11.

- If we start counting rows and columns from 0, then the element stored in row 2 and column 1 in the two-dimensional array (the 9), is located in position

2x4+1 = 9 in the one-dimensional array.

- More generally, if our array has ***n*** columns, when we use this scheme, we see that the element stored in row ***i*** and column ***j*** is located in position ***ixn+j*** in the one-dimensional array. Using this one-dimensional scheme, we get the C function shown in Program 11.

- An individual task can be the multiplication of an element of A by a component of ***x*** and the addition of this product into a component of ***y***. That is, each execution of the statement

```
y[i] += A[i*n+j]*x[j];
```

```
1  void Mat_vect_mult(
2        double  A[]  /* in  */,
3        double  x[]  /* in  */,
4        double  y[]  /* out */,
5        int     m    /* in  */,
6        int     n    /* in  */) {
7     int i, j;
8
9     for (i = 0; i < m; i++) {
10       y[i] = 0.0;
11       for (j = 0; j < n; j++)
12          y[i] += A[i*n+j]*x[j];
13    }
14 }  /* Mat_vect_mult */
```

Program 11: Serial matrix-vector multiplication

- So we see that if ***y[i]*** is assigned to process q, then it would be convenient to also assign row ***i*** of A to process q. This suggests that we partition ***A*** by rows. We could partition the rows using a block distribution, a cyclic distribution, or a blockcyclic distribution.

- In MPI it's easiest to use a block distribution, so let's use a block distribution of the rows of A, and, as usual, assume that ***comm._sz*** evenly divides m, the number of rows.

- We are distributing ***A*** by rows so that the computation of ***y[i]*** will have all of the needed elements of ***A***, so we should distribute ***y*** by blocks. That is, if the ***ith*** row of ***A***, is assigned to process ***q***, then the ith component of ***y*** should also be assigned to process ***q***.
- The computation of ***y[i]*** involves all the elements in the ***ith*** row of ***A*** and all the components of x, so we could minimize the amount of communication by simply assigning all of x to each process.

  - However, in actual applications—especially when the matrix is square—it's often the case that a program using matrix-vector multiplication will execute the multiplication many times and the result vector ***y*** from one multiplication will be the input vector ***x*** for the next iteration. In practice, then, we usually assume that the distribution for ***x*** is the same as the distribution for ***y***.

```
for (j = 0; j < n; j++)
   y[i] += A[i*n+j]*x[j];
```

  - Using the collective communications we're already familiar with, we could execute a call to MPI_Gather followed by a call to MPI_Bcast. This would, in all likelihood, involve two tree-structured communications, and we may be able to do better by using a butterfly. So, once again, MPI provides a single function:

```
int MPI_Allgather(
      void*          send_buf_p   /* in */,
      int            send_count   /* in  */,
      MPI_Datatype   send_type    /* in  */,
      void*          recv_buf_p   /* out */,
      int            recv_count   /* in  */,
      MPI_Datatype   recv_type    /* in  */,
      MPI_Comm       comm         /* in  */);
```

- This function concatenates the contents of each process' ***send_buf_p*** and stores this in each process' ***recv_buf_p***. As usual, ***recv_count*** is the amount of data being received from each process, so in most cases, ***recv_count*** will be the same as ***send_count.***

```
1  void Mat_vect_mult(
2        double    local_A[]  /* in  */,
3        double    local_x[]  /* in  */,
4        double    local_y[]  /* out */,
5        int       local_m    /* in  */,
6        int       n          /* in  */,
7        int       local_n    /* in  */,
8        MPI_Comm  comm       /* in  */) {
9     double* x;
10    int local_i, j;
11    int local_ok = 1;
12
13    x = malloc(n*sizeof(double));
14    MPI_Allgather(local_x, local_n, MPI_DOUBLE,
15          x, local_n, MPI_DOUBLE, comm);
16
17    for (local_i = 0; local_i < local_m; local_i++) {
18       local_y[local_i] = 0.0;
19       for (j = 0; j < n; j++)
20          local_y[local_i] += local_A[local_i*n+j]*x[j];
21    }
22    free(x);
23 }  /* Mat_vect_mult */
```

Program 12: An MPI matrix-vector multiplication function