Scatter & Gather

- 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

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

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