MPI Programming — Part 2

Objectives

- Barrier synchronization
- Broadcast, reduce, gather, scatter
- Example: Dot product
- Derived data types
- Performance evaluation

Collective communications

In addition to point-to-point communications, MPI includes routines for performing *collective communications*, i.e., communications involving all processes in a communicator, to allow larger groups of processors to communicate, e.g., one-to-many or many-to-one.

These routines are built using point-to-point communication routines, so in principle you could build them yourself.

However, there are several advantages of directly using the collective communication routines, including

- The possibility of error is reduced. One collective routine call replaces many point-to-point calls.
- The source code is more readable, thus simplifying code debugging and maintenance.
- The collective routines are optimized.

Collective communications

Collective communication routines transmit data among all processes in a communicator.

It is important to note that collective communication calls *do not use the tag mechanism of send/receive for associating calls*.

Rather, calls are associated by the order of the program execution.

Thus, *the programmer must ensure that all processes execute the same collective communication calls and execute them in the same order*.

The collective communication routines can be applied to all processes or a specified set of processes as defined in the communicator.

For simplicity, we assume all processes participate in the collective communications, but it is always possible to define a collective communication between a subset of processes with a suitable communicator.

MPI Collective Communication Routines

MPI provides the following collective communication routines:

- Barrier sychronization across all processes.
- Broadcast from one process to all other processes.
- Global reduction operations such as sum, min, max, or user-defined reductions.
- Gather data from all processes to one process.
- Scatter data from one process to all processes.
- Advanced operations in which all processes receive the same result from a gather, scatter, or reduction. There is also a vector variant of most collective operations where messages can have different sizes.

MPI Collective Communication Routines

Notes:

1. In many implementations of MPI, calls to collective communication routines will synchronize the processes. However, this synchronization is not guaranteed, so you should not count on it!

2. The `MPI_BARRIER` routine synchronizes the processes but does not pass data. Despite this, it is often categorized as a collective communications routine.

Barrier synchronization

Sometimes you need to hold up some or all processes until some other processes have completed a task.

For example, a root process reads data and then must transmit these data to other processes.

The other processes must wait until they receive the data before they can proceed.

The `MPI_BARRIER` routine blocks the calling process until all processes have called the function.

When `MPI_BARRIER` returns, all processes are synchronized at that point.

WARNING! `MPI_BARRIER` is done in software and can incur a substantial overhead on some machines.

In general, you should use barriers sparingly!

Fortran syntax:

`MPI_BARRIER ( COMM, IERR )`

Input argument `COMM` of type `INTEGER` is the communicator defining the processes to be held up at the barrier.

Output argument `IERR` of type `INTEGER` is the error flag.

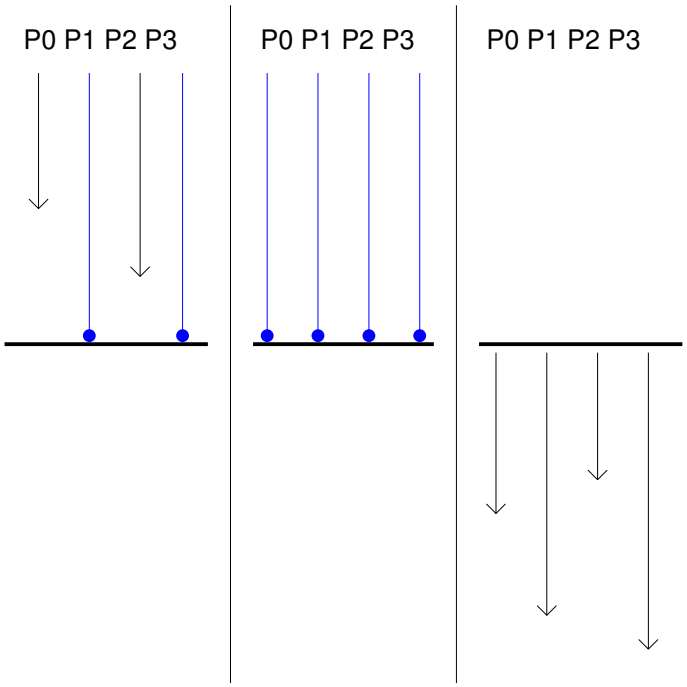

Figure 1: The effect of `MPI_BARRIER`.

Broadcast

The simplest collective operation involving the transfer of data is the *broadcast*.

In a broadcast operation, *a single process sends a copy of some data to all the other processes in a communicator*.

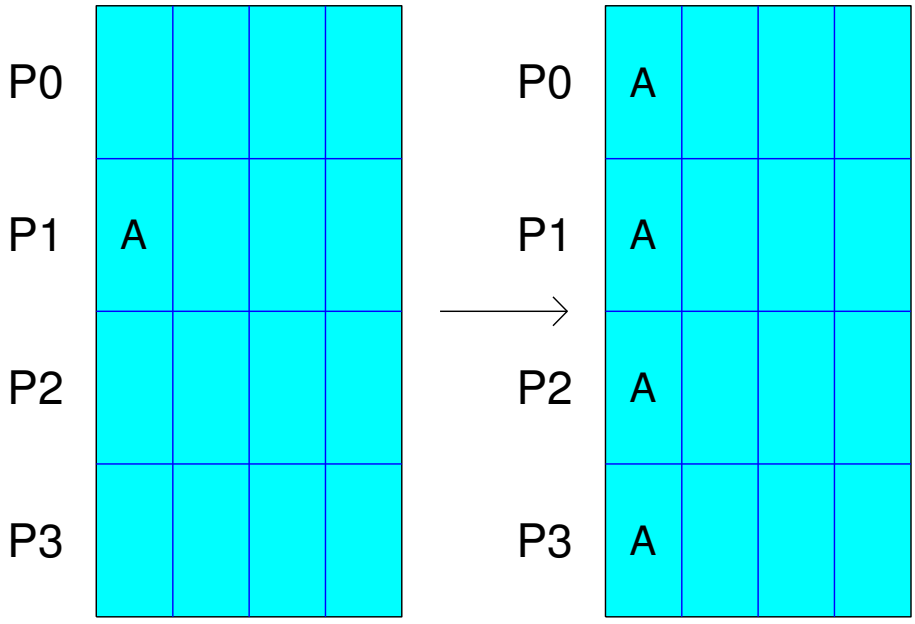

Figure 2: `MPI_BCAST` operation.

Broadcast

Specifically, the `MPI_BCAST` routine copies data from the memory of the root process to the same memory locations for other processes in the communicator.

Clearly, you could accomplish the same thing with multiple calls to a send routine.

However, use of `MPI_BCAST` makes the program

- easier to read (one line replaces loop)
- easier to maintain (only one line to modify)
- more efficient (use optimized implementations)

Fortran syntax:

```
MPI_BCAST ( BUF, COUNT, DTYPE, ROOT, COMM, IERR )
```

Input argument BUF is the array of data to be sent.

Input argument COUNT of type INTEGER gives the number of elements in BUF.

Input argument DTYPE gives the data type of the entries of BUF.

Input argument ROOT of type INTEGER is the rank of the sending process.

Input argument COMM is the communicator of the processes that are to receive the broadcasted data.

Output argument IERR is the usual error flag.

Send contents of array BUF with COUNT elements of type DTYPE from process ROOT to all processes in communicator COMM and return with flag IERR.

Reduction

In a *reduction* operation, a single process collects data from the other processes in a communicator and combines them into a single data item.

For example, reduction could be used to sum array elements that are distributed over several processes.

Operations besides arithmetic are also possible, for example, maximum and minimum, as well as various logical and bitwise operations.

Before the reduction, the data, which may be arrays or scalar values, are distributed across the processes.

After the reduction operation, the reduced data (array or scalar) are located on the root process.

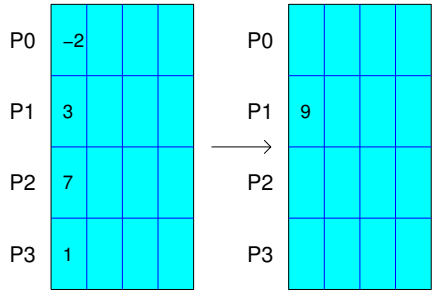

Figure 3: `MPI_REDUCE` operation with `MPI_SUM`.

Reduction

Pre-defined reduction operators to be used with `MPI_REDUCE` are

- `MPI_MAX`, `MPI_MIN`: maximum and minimum
- `MPI_MAXLOC`, `MPI_MINLOC`: maximum and minimum with corresponding array index
- `MPI_SUM`, `MPI_PROD`: sum and product
- `MPI_LAND`, `MPI_LOR`: logical AND and OR
- `MPI_BAND`, `MPI_BOR`: bitwise AND and OR
- `MPI_LXOR`, `MPI_BXOR`: logical, bitwise exclusive OR

Fortran syntax:

```
MPI_REDUCE ( SEND_BUF, RECV_BUF, COUNT, DTYPE,
             OP, RANK, COMM, IERR )
```

Input argument SEND_BUF is the array to be sent.

Output argument RECV_BUF is the reduced value that is returned.

Input argument COUNT of type INTEGER gives the number of elements in SEND_BUF and RECV_BUF.

Input argument DTYPE gives the data type of the entries of SEND_BUF and RECV_BUF.

Input argument OP is the reduction operation.

Input argument RANK of type INTEGER is the rank of the sending process.

Input argument COMM is the communicator of the processes that have the data to be reduced.

Output argument IERR is the usual error flag.

Gather

The *gather* operation collects pieces of the data that are distributed across a group of processes and (re)assembles them appropriately on a single process.

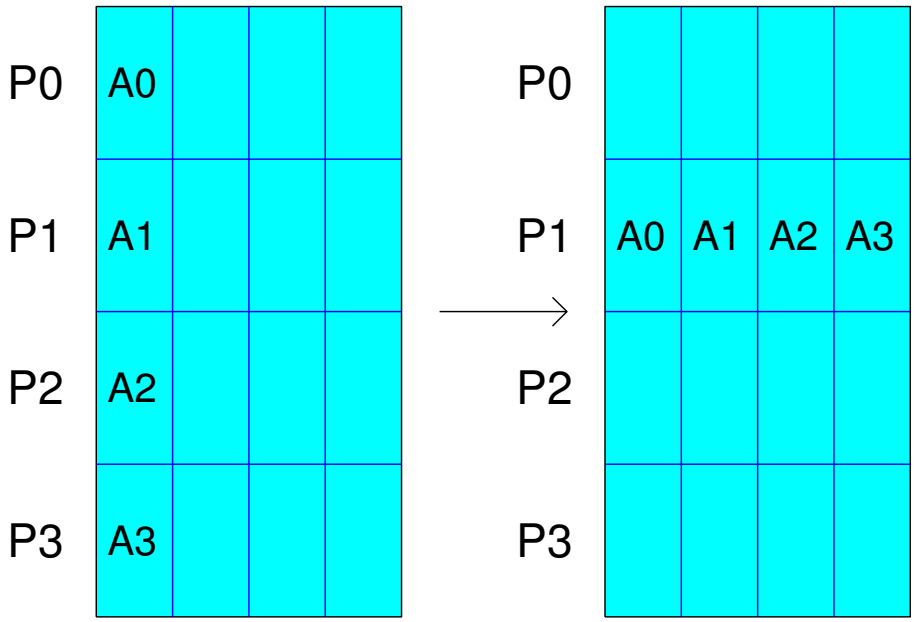

Figure 4: `MPI_GATHER` operation.

Gather

Similar to MPI_REDUCE, the MPI_GATHER routine is an *all-to-one* communication routine.

When MPI_GATHER is called, each process (including the root process) sends the contents of its send buffer to the root process.

The root process receives the messages and stores them in contiguous memory locations and in order of rank.

The outcome is the same as each process calling MPI_SEND and the root process calling MPI_RECV some number of times to receive all of the messages.

MPI_GATHER requires that all processes, including the root, send the same amount of data, and that the data are of the same type.

Thus, the send count equals the receive count.

Fortran syntax:

```
MPI_GATHER(SEND_BUF,SEND_COUNT,SEND_DTYPE,RECV_BUF,
           RECV_COUNT,RECV_DTYPE,RANK,COMM,IERR)
```

Input argument SEND_BUF is the array to be gathered.

Input argument SEND_COUNT of type INTEGER gives the number of elements in SEND_BUF.

Input argument SEND_DTYPE is the data type of the elements of SEND_BUF.

Output argument RECV_BUF is the array to receive the gathered data; it is only meaningful to process RANK.

Input arguments RECV_COUNT of type INTEGER and RECV_DTYPE give the number of elements and data type of RECV_BUF expected from each process.

Input argument RANK of type INTEGER is the rank of the gathering process.

Input argument COMM is the communicator of the processes that have the data to be gathered.

MPI_ALLGATHER

After the data have been gathered into the root process, MPI_BCAST could then be used to distribute the gathered data to all of the other processes.

It is more convenient and efficient to do this via the MPI_ALLGATHER routine.

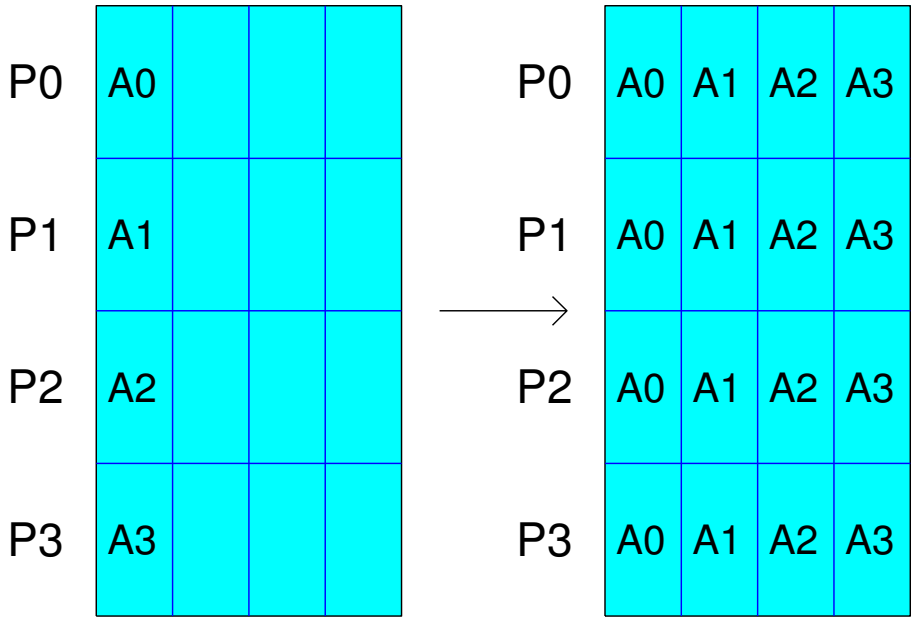

Figure 5: The effect of MPI_ALLGATHER.

The syntax for MPI_ALLGATHER is the same as it is for MPI_GATHER except the RANK argument is omitted.

Scatter

In a *scatter* operation, all of the data are initially collected on a single process.

After the scatter operation, pieces of the data are distributed on different processes.

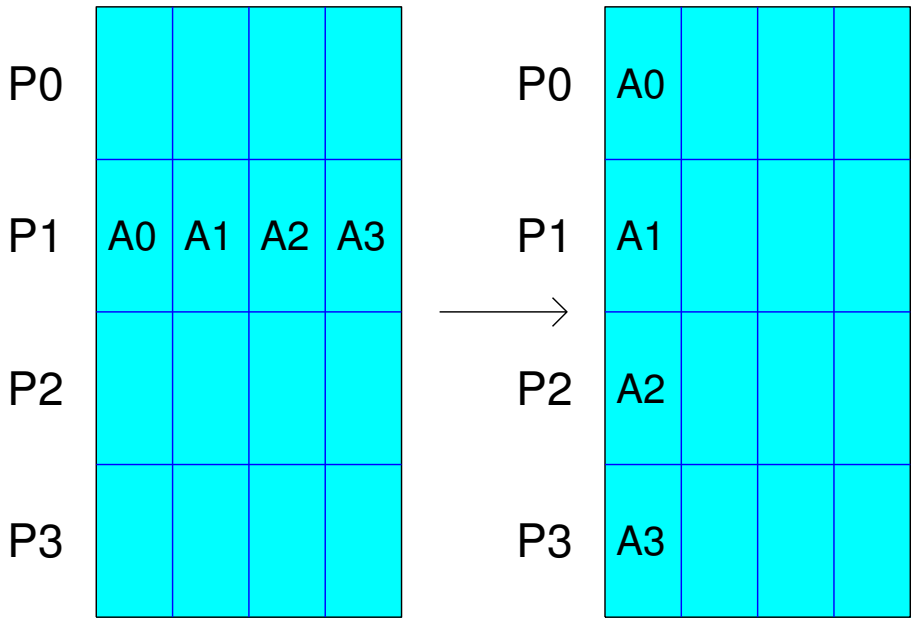

Figure 6: `MPI_SCATTER` operation.

Scatter

The `MPI_SCATTER` routine is a *one-to-all* communication routine.

Different data are sent from the root process to each process (in rank order).

When `MPI_SCATTER` is called, the root process breaks up a set of contiguous memory locations into equal chunks and sends one chunk to each process.

The outcome is the same as root calling `MPI_SEND` some number of times and each process calling `MPI_RECV`.

Fortran syntax:

```
MPI_SCATTER(SEND_BUF,SEND_COUNT,SEND_DTYPE,RECV_BUF,
            RECV_COUNT,RECV_TYPE,RANK,COMM,IERR)
```

Input argument SEND_BUF is the array to be scattered.

Input argument SEND_COUNT of type INTEGER gives the number of elements in SEND_BUF to be sent to each process.

Input argument SEND_DTYPE is the data type of the elements of SEND_BUF.

Output argument RECV_BUF is the array that receives the data.

Input arguments RECV_COUNT of type INTEGER and RECV_DTYPE give the number of elements and data type of RECV_BUF expected for a single receive.

Input argument RANK of type INTEGER is the rank of the scattering process.

Input argument COMM is the communicator of the processes that receive the data to be scattered.

Other operations

- MPI_ALLREDUCE acts like MPI_REDUCE except the reduced result is broadcast to all processes.
- It is possible to define your own reduction operation using MPI_OP_CREATE.
- MPI_GATHERV and MPI_SCATTERV gather or scatter with data items that may have different sizes.
- MPI_ALLTOALL: all processes get all data (total exchange); data items must be same size.
- MPI_ALLTOALLV acts like MPI_ALLTOALL with data items that may have different sizes.
- MPI_SCAN performs a reduction operation on a subset of processes in a communicator.
- MPI_REDUCE_{GATHER|SCATTER} acts like MPI_REDUCE followed by MPI_{GATHER|SCATTER}V.

Example: Dot product

The following Fortran code computes the dot product $\mathbf{x} \cdot \mathbf{y} = \mathbf{x}^T \mathbf{y}$ of two vectors $\mathbf{x}, \mathbf{y} \in \Re^N$.

```
PROGRAM dotProductMPI
!
! This program computes the dot product of two vectors X,Y
! (each of size N) with component i having value i
! in parallel using P processes.
! Vectors are initialized in the code by the root process,
! then statically distributed in blocks to all processes.
! It is not assumed N is divisible by P.
!
  INCLUDE 'mpif.h'
  ! variable declarations

  INTEGER, PARAMETER :: N = 100
  REAL, PARAMETER :: ROOT = 0

  INTEGER :: P, NBAR
  INTEGER :: RANK, I, EXTRA, INDEX, OFFSET = 0
  INTEGER :: IERR
  REAL :: X(N), Y(N)
  REAL :: DOT, DOT_LOC = 0.0

  ! initialize MPI
  CALL MPI_INIT(IERR)
  IF (IERR.NE.MPI_SUCCESS) THEN
     PRINT*, "ERROR: MPI not initialized."
     STOP
  ENDIF
```

```
  ! Get the number of processes:
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, P, IERR)
  IF (IERR.NE.MPI_SUCCESS) THEN
     PRINT*, "ERROR: MPI processes not established."
     STOP
  ENDIF

  ! Get ranks of processes:
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERR)
  IF (IERR.NE.MPI_SUCCESS) THEN
     PRINT*, "ERROR: MPI ranks not established."
     STOP
  ENDIF

  ! Root process initializes vectors X,Y and distributes them
  IF (RANK.EQ.ROOT) THEN
     DO 10 I=1,N
        X(I) = I
        Y(I) = I
10   END DO
  ENDIF
  ! this could probably be done more efficiently by packing X and Y
  ! into one entity and broadcasting it
  CALL MPI_BCAST(X, N, MPI_REAL, ROOT, MPI_COMM_WORLD, IERR)
  IF (IERR.NE.MPI_SUCCESS) THEN
     PRINT*, "ERROR: MPI_BCAST not successful."
     STOP
  ENDIF
  CALL MPI_BCAST(Y, N, MPI_REAL, ROOT, MPI_COMM_WORLD, IERR)
  IF (IERR.NE.MPI_SUCCESS) THEN
     PRINT*, "ERROR: MPI_BCAST not successful."
     STOP
  ENDIF

  ! determine which block of data to work on and compute dot product
  NBAR  = N/P
  EXTRA = MOD(N,P)
  IF (RANK < EXTRA) OFFSET = 1
```

```
  DO 20 I=1,NBAR+OFFSET
     INDEX = RANK*NBAR + I + MIN(EXTRA,RANK)
     DOT_LOC = DOT_LOC + X(INDEX)*Y(INDEX)
20 END DO

  ! gather and reduce the data and print the result
  CALL MPI_REDUCE(DOT_LOC, DOT, 1, MPI_REAL, MPI_SUM, ROOT, &
       MPI_COMM_WORLD, IERR)
  IF (RANK.EQ.ROOT) THEN
     IF (IERR.NE.MPI_SUCCESS) THEN
        PRINT*, "ERROR: MPI_REDUCE not successful."
        STOP
     ENDIF
     PRINT*, 'The dot product is: ', DOT
     PRINT*, 'The answer should be: ', N*(N+1)*(2*N+1)/6
  ENDIF

  ! Finalize MPI:
  CALL MPI_FINALIZE(IERR)
  IF (IERR.NE.MPI_SUCCESS) THEN
     PRINT*, "ERROR: MPI not finalized."
     STOP
  ENDIF

END PROGRAM dotProductMPI
```

Example: Trapezoidal rule revisited

With knowledge of the collective communication features in MPI, we can revisit the program for the trapezoidal rule to improve its communication patterns.

To recall, the basic strategy behind the first version of the trapezoidal rule program was to have each process determine its region of integration, perform the trapezoidal rule, and send its result to Process 0, which would then accumulate and print the result.

We can improve the way Process 0 receives the result through the use of `MPI_Reduce`.

Example: Trapezoidal rule revisited

To achieve this, we can replace the code block

```
/* Add up the integrals calculated by each process */
if (my_rank != 0) {
   MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
         MPI_COMM_WORLD);
} else {
   total_int = local_int;
   for (source = 1; source < comm_sz; source++) {
      MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      total_int += local_int;
   }
}
```

with

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

To generalize this to a sum of N-dimensional vectors, we can use

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

Caveats of collective communication

- All processes in a communicator must call the same collective function; if not the program will hang.
- The arguments must be consistent; e.g., the process on which to collect results must be the same one!
- The output argument is only used on destination process, but all processes must pass an argument to it (even if it is `NULL`).
- Recall that collective communications match based on calling order. So, e.g., if `MPI_Reduce` is used with operator `MPI_SUM` and destination process 0,

t	Process 0	Process 1	Process 2
0	`a=1; c=2;`	`a=1; c=2;`	`a=1; c=2;`
1	`MPI_Reduce(&a,&b,...)`	`MPI_Reduce(&c,&d,...)`	`MPI_Reduce(&a,&b,...)`
2	`MPI_Reduce(&c,&d,...)`	`MPI_Reduce(&a,&b,...)`	`MPI_Reduce(&c,&d,...)`

the final value of `b` is $1 + 2 + 1 = 4$ and the final value of `d` is $2 + 1 + 2 = 5$.

Caveats of collective communication

- Trying to use the same buffer for input and output in `MPI_Reduce` is illegal and its result is *unpredictable* — you could get anything from the right answer to a program crash. The act of having an output argument and an input/output argument refer to the same memory location is called *aliasing*. MPI prohibits aliasing because it is illegal in Fortran, and the MPI Forum wanted to make the C and Fortran versions as similar as possible[1].

  In other words, do *not* use a call such as

```
MPI_Reduce(&x, &x, 1, MPI_DOUBLE, 0, comm);
```

[1] Despite this, we may see a potential workaround later.

Data distribution

There are three usual ways to distribute data to processes. Suppose we have `n` pieces of data, `comm_sz` processes, and that `n` divides evenly over `comm_sz`.

In a *block partition*, we simply assign blocks of size `n_local = n/comm_sz` in order to each process.

In a *cyclic partition*, we assign the components one a time in a round-robin style.

In a *block-cyclic partition* the data are partitioned into blocks and the blocks are distributed cyclically.

p	Block				Cyclic				Block-Cyclic			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	2	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

`MPI_Scatter` uses a block partition, so it is only suitable when `n` divides evenly over `comm_sz`.

If `n` does not evenly divide over `comm_sz`, we can use `MPI_Scatterv`.

Derived data types in MPI

We have stressed the overhead of message passing compared to (local) computation.

It also usually pays to consolidate data into fewer (but larger) messages instead of many small messages.

We have already seen the use of the `count` argument to group contiguous array elements into a single message.

In MPI, a *derived data type* is a way to mix and match any collection of basic MPI data types into a single representation.

This is achieved by storing both the types of the data along with their relative locations in memory.

This way all the data can be collected into one message before they are sent.

(Similarly, they can be distributed into their proper locations by the receiving process.)

Derived data types in MPI

As an example, in the program for the trapezoidal method, it is not hard to imagine that process 0 might have to broadcast information (such as `a`, `b`, and `n`) rather than have each process determine it.

In this case, we can build a derived data type consisting of two `doubles` and an `int` and use one `MPI_Bcast` (instead of three) to distribute `a`, `b`, and `n`.

To create the derived data type, we need to specify the basic MPI data type along with a *displacement*, measured in bytes, of the address of the variable from the beginning of the data type.

Derived data types in MPI

For example, suppose that on process 0, `a`, `b`, and `n` are stored in memory locations 24, 40, and 48, respectively.

These data items could be represented by the derived data type

```
{(MPI_DOUBLE,0), (MPI_DOUBLE,16), (MPI_INT,24)},
```

where the first element is the basic MPI data type and the second element is the displacement of the element from the beginning of the data type.

Thus,

- `a` is of type `MPI_DOUBLE` and its displacement from the beginning of the data type is 0 (by definition).
- `b` is of type `MPI_DOUBLE` and its displacement from the beginning of the data type is $40 - 24 = 16$.
- `n` is of type `MPI_INT` and its displacement from the beginning of the data type is $48 - 24 = 24$.

Derived data types in MPI

Derived data types can be built using the `MPI_Type_create_struct` function with syntax

```
int MPI_Type_create_struct(
    int             count                       /* in  */,
    int             array_of_blocklengths[]     /* in  */,
    MPI_Aint        array_of_displacements[]    /* in  */,
    MPI_Datatype    array_of_types[]            /* in  */,
    MPI_Datatype*   new_type_p                  /* out */);
```

where `count` is the number of elements in the data type; each array argument must have `count` elements.

In our example, `count` $= 3$ and we would define

`int array_of_blocklengths[3] =` $\{$`1,1,1`$\}$`;`

It is possible for individual data items to be arrays (or subarrays), in which case the corresponding element of `array_of_blocklengths` would not be 1.

Derived data types in MPI

The argument `array_of_displacements` specifies the displacement from the start of the data type, so in our example we would set

```
array_of_displacements = {0, 16, 24};
```

In practice, the addresses are obtained using the function `MPI_Get_address` with syntax

```
int MPI_Get_address(
      void*      location_p  /* in  */,
      MPI_Aint*  address_p   /* out */);
```

where the address of the memory location referenced by `location_p` is returned.

The MPI type `MPI_Aint` is a special integer type that is large enough to store an address on the system.

Derived data types in MPI

For our example, we can populate the elements of array_of_displacements via

```
MPI_Aint a_addr, b_addr, n_addr;

MPI_Get_address(&a, &a_addr);
array_of_displacements[0] = 0;
MPI_Get_address(&b, &b_addr);
array_of_displacements[0] = b_addr - a_addr;
MPI_Get_address(&n, &n_addr);
array_of_displacements[0] = n_addr - a_addr;
```

The MPI data types of the elements are stored in array_of_types via the definition

```
MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
```

The new data type can now be built with the call

```
MPI_Datatype input_mpi_t;
...
MPI_Type_create_struct(3, array_of_blocklengths,
       array_of_displacements, array_of_types,
       &input_mpi_t);
```

Derived data types in MPI

Finally, before using `input_mpi_t`, we commit it using

```
int MPI_Type_commit(MPI_Datatype*, new_mpi_t_p /* in/out */);
```

To now use it, we can make the following call on each process

```
MPI_Bcast(&a, 1, input_mpi_t, 0, comm);
```

In other words, we can use it just like any of the basic MPI data types.

Finally, constructing the new data type likely required additional internal storage.

When we are done using the new data type, we can free this additional storage via

```
int MPI_Type_free(MPI_Datatype* old_mpi_t_p /* in/out */);
```

Derived data types in MPI

```
/* File:     mpi_trap4.c
 * Purpose:  Use MPI to implement a parallel version of the trapezoidal
 *           rule.  This version uses collective communications and
 *           MPI derived datatypes to distribute the input data and
 *           compute the global sum.
 *
 * Input:    The endpoints of the interval of integration and the number
 *           of trapezoids
 * Output:   Estimate of the integral from a to b of f(x)
 *           using the trapezoidal rule and n trapezoids.
 *
 * Compile:  mpicc -g -Wall -o mpi_trap4 mpi_trap4.c
 * Run:      mpiexec -n <number of processes> ./mpi_trap4
 *
 * Algorithm:
 *    1.  Each process calculates "its" interval of
 *        integration.
 *    2.  Each process estimates the integral of f(x)
 *        over its interval using the trapezoidal rule.
 *    3a. Each process != 0 sends its integral to 0.
 *    3b. Process 0 sums the calculations received from
 *        the individual processes and prints the result.
 *
 * Note:  f(x) is all hardwired.
* IPP:   Section 3.5 (pp. 117 and ff.) */
#include <stdio.h>

/* We'll be using MPI routines, definitions, etc. */
#include <mpi.h>

/* Build a derived datatype for distributing the input data */
void Build_mpi_type(double* a_p, double* b_p, int* n_p,
      MPI_Datatype* input_mpi_t_p);
```

```
/* Get the input values */
void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p,
      int* n_p);

/* Calculate local integral  */
double Trap(double left_endpt, double right_endpt, int trap_count,
   double base_len);

/* Function we're integrating */
double f(double x);

int main(void) {
   int my_rank, comm_sz, n, local_n;
   double a, b, dx, local_a, local_b;
   double local_int, total_int;

   /* Let the system do what it needs to start up MPI */
   MPI_Init(NULL, NULL);

   /* Get my process rank */
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   /* Find out how many processes are being used */
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

   Get_input(my_rank, comm_sz, &a, &b, &n);

   dx = (b-a)/n;          /* dx is the same for all processes */
   local_n = n/comm_sz;  /* So is the number of trapezoids  */

   /* Length of each process' interval of
    * integration = local_n*dx.  So my interval
    * starts at: */
   local_a = a + my_rank*local_n*dx;
   local_b = local_a + local_n*dx;
   local_int = Trap(local_a, local_b, local_n, dx);
```

```
   /* Add up the integrals calculated by each process */
   MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,
         MPI_COMM_WORLD);

   /* Print the result */
   if (my_rank == 0) {
      printf("With n = %d trapezoids, our estimate\n", n);
      printf("of the integral from %f to %f = %.15e\n",
          a, b, total_int);
   }

   /* Shut down MPI */
   MPI_Finalize();

   return 0;
} /*  main  */

/*--------------------------------------------------------------------
 * Function:     Build_mpi_type
 * Purpose:      Build a derived datatype so that the three
 *               input values can be sent in a single message.
 * Input args:   a_p:  pointer to left endpoint
 *               b_p:  pointer to right endpoint
 *               n_p:  pointer to number of trapezoids
 * Output args:  input_mpi_t_p:  the new MPI datatype
 */
void Build_mpi_type(
      double*        a_p            /* in  */,
      double*        b_p            /* in  */,
      int*           n_p            /* in  */,
      MPI_Datatype*  input_mpi_t_p  /* out */) {

   int array_of_blocklengths[3] = {1, 1, 1};
   MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
   MPI_Aint a_addr, b_addr, n_addr;
   MPI_Aint array_of_displacements[3] = {0};

   MPI_Get_address(a_p, &a_addr);
```

```
   MPI_Get_address(b_p, &b_addr);
   MPI_Get_address(n_p, &n_addr);
   array_of_displacements[1] = b_addr-a_addr;
   array_of_displacements[2] = n_addr-a_addr;
   MPI_Type_create_struct(3, array_of_blocklengths,
         array_of_displacements, array_of_types,
         input_mpi_t_p);
   MPI_Type_commit(input_mpi_t_p);
}  /* Build_mpi_type */

/*---------------------------------------------------------------------
 * Function:     Get_input
 * Purpose:      Get the user input:  the left and right endpoints
 *               and the number of trapezoids
 * Input args:   my_rank:  process rank in MPI_COMM_WORLD
 *               comm_sz:  number of processes in MPI_COMM_WORLD
 * Output args:  a_p:  pointer to left endpoint
 *               b_p:  pointer to right endpoint
 *               n_p:  pointer to number of trapezoids
 */
void Get_input(
      int      my_rank  /* in  */,
      int      comm_sz  /* in  */,
      double*  a_p      /* out */,
      double*  b_p      /* out */,
      int*     n_p      /* out */) {
   MPI_Datatype input_mpi_t;

   Build_mpi_type(a_p, b_p, n_p, &input_mpi_t);

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
   }
   MPI_Bcast(a_p, 1, input_mpi_t, 0, MPI_COMM_WORLD);

   MPI_Type_free(&input_mpi_t);
}  /* Get_input */
```

```
/*--------------------------------------------------------------------
 * Function:     Trap
 * Purpose:      Serial function for estimating a definite integral
 *               using the trapezoidal rule
 * Input args:   left_endpt
 *               right_endpt
 *               trap_count
 *               base_len
 * Return val:   Trapezoidal rule estimate of integral from
 *               left_endpt to right_endpt using trap_count
 *               trapezoids
 */
double Trap(
      double left_endpt  /* in */,
      double right_endpt /* in */,
      int    trap_count  /* in */,
      double base_len    /* in */) {
   double estimate, x;
   int i;

   estimate = (f(left_endpt) + f(right_endpt))/2.0;
   for (i = 1; i <= trap_count-1; i++) {
      x = left_endpt + i*base_len;
      estimate += f(x);
   }
   estimate = estimate*base_len;

   return estimate;
} /*  Trap  */
/*--------------------------------------------------------------------
 * Function:    f
 * Purpose:     Compute value of function to be integrated
 * Input args:  x
 */
double f(double x /* in */) {
   return x*x;
} /* f */
```

Performance evaluation

We have stressed the importance of performance evaluation in parallel programming.

In some sense, the whole point of parallel programming can be understood to be completing a calculation faster and faster as more and more processes are employed.

In order to measure this, we take timings.

We are mostly interested in the wall-clock execution time of the functional part of the code, and we generally report the minimum of several timings.

Recall that it is also important to know the resolution of the timer being used in order to gauge the precision and reliability of the timings.

Performance evaluation

MPI provides the `MPI_Wtime` function that gives the time in seconds from some arbitrary time in the past:

```
double MPI_Wtime(void);
```

So a block of MPI code can be timed using code like

```
double start, finish;
...
start = MPI_Wtime();
/* code to be timed */
...
finish = MPI_Wtime();
printf("Processor %d > Elapsed time = %e seconds.\n",
      my_rank, finish-start);
```

The resolution of `MPI_Wtime` on a given system can be measured using `MPI_Wtick`:

```
double MPI_Wtick(void);
```

Performance evaluation

Of course, we still need to extract a single run time for the program.

To do this, we synchronize all the processes before the timing begins with a call to `MPI_Barrier` followed by a call to `MPI_Reduce` to find the maximum.

```
double local_start, local_finish;
...
MPI_Barrier(comm);
local_start = MPI_Wtime();
/* code to be timed */
...
local_finish = MPI_Wtime();
local_elapsed = local_finish - local_start
MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE,
   MPI_MAX, 0, comm);

if (my_rank == 0),
   printf("Elapsed time = %e seconds.\n",
      elapsed);
```

Example: matrix-vector multiplication

As an example, we consider the performance of a parallel program to compute a matrix-vector product in the "standard" (dot-product) fashion.

Let $\mathbf{A}$ be an $m \times n$ matrix and $\mathbf{x}$ be an n-vector.

Then the common way to compute and/or interpret $\mathbf{Ax}$ is via m inner products:

$$\mathbf{Ax} = \begin{bmatrix} (\mathbf{a}_1, \mathbf{x}) \\ \vdots \\ (\mathbf{a}_m, \mathbf{x}), \end{bmatrix}$$

where $\mathbf{a}_i$ is row i of $\mathbf{A}$.

Example: matrix-vector multiplication

Let $\mathbf{y} = \mathbf{A}\mathbf{x}$ be an m-vector.

Then algorithmically, we have

$\mathbf{y} = \mathbf{0}$
for $i = 1$ to m **do**
  for $j = 1$ to n **do**
    $y_i = y_i + a_{ij}x_j$
  end for
end for

Example: matrix-vector multiplication

Translated to serial code,

```
void Mat_vect_mult(
                    double  A[]  /* in  */,
                    double  x[]  /* in  */,
                    double  y[]  /* out */,
                    int     m    /* in  */,
                    int     n    /* in  */) {
   int i, j;

   for (i = 0; i < m; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
         y[i] += A[i*n+j]*x[j];
   }
}  /* Mat_vect_mult */
```

Example: matrix-vector multiplication

A simple parallel program to compute a matrix-vector product using this approach is given by the following.

```
void Mat_vect_mult(
      double    local_A[]  /* in  */,
      double    local_x[]  /* in  */,
      double    local_y[]  /* out */,
      int       local_m    /* in  */,
      int       n          /* in  */,
      int       local_n    /* in  */,
      MPI_Comm  comm       /* in  */) {
   double* x;
   int local_i, j;
   int local_ok = 1;

   x = malloc(n*sizeof(double));
   if (x == NULL) local_ok = 0;
   Check_for_error(local_ok, "Mat_vect_mult",
         "Can't allocate temporary vector", comm);
   MPI_Allgather(local_x, local_n, MPI_DOUBLE,
         x, local_n, MPI_DOUBLE, comm);

   for (local_i = 0; local_i < local_m; local_i++) {
      local_y[local_i] = 0.0;
      for (j = 0; j < n; j++)
         local_y[local_i] += local_A[local_i*n+j]*x[j];
   }
   free(x);
}  /* Mat_vect_mult */
```

Example: matrix-vector multiplication

Here are some timing results taken from the text for parallel matrix-vector multiplication:

Table 3.5 Run-Times of Serial and Parallel Matrix-Vector Multiplication (times are in milliseconds)

	Order of Matrix				
comm_sz	*1024*	*2048*	*4096*	*8192*	*16,384*
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

Example: matrix-vector multiplication

Some observations:

- For fixed `n`, run times decrease as `P` increases.
- For large `n`, doubling `P` halves run time.
- For small `n`, increasing `P` has less effect.

Recall

$$T_{\text{parallel}}(n, P) = \frac{T_{\text{serial}}(n)}{P} + T_{\text{overhead}},$$

where for our program T_{overhead} basically represents the time to perform the `MPI_Allgather`.

Clearly T_{overhead} is negligible when `n` is large and `P` is small and dominates when `n` is small and `P` is large.

Example: matrix-vector multiplication

The speedups are calculated as:

Table 3.6 Speedups of Parallel Matrix-Vector Multiplication

	Order of Matrix				
comm_sz	*1024*	*2048*	*4096*	*8192*	*16,384*
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

Example: matrix-vector multiplication

Some observations:

- Nearly linear speedups are obtained for small P and large n.
- Little or no (relative) speedups are obtained for large P and small n.
- No (relative) slowdown occurred, but it was close!
- There was a steady improvement in speedup as for fixed $P(> 1)$ as n increased.

Example: matrix-vector multiplication

The efficiencies are calculated as:

Table 3.7 Efficiencies of Parallel Matrix-Vector Multiplication

	Order of Matrix				
comm_sz	*1024*	*2048*	*4096*	*8192*	*16,384*
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

Example: matrix-vector multiplication

Analogous statements hold for efficiencies:

- Nearly perfect efficiencies are obtained for small P and large n.
- Efficiency is poor for large P and small n.
- There was a steady improvement in efficiency as for fixed $P(> 1)$ as n increased.

Example: matrix-vector multiplication

Finally, considering scalability, recall that there are two flavours of scalability:

1. Strong scalability: efficiency remains (essentially) constant for constant problem size as number of processes increases.

2. Weak scalability: efficiency remains (essentially) constant as the problem size and number of processes increase proportionately.

Based on the observations provided, the matrix-vector multiplication program appears to be weakly scalable for n sufficiently large.

Specifically, this can be seen by looking at the values along the super-diagonals in Table 3.7 on efficiencies (or equivalently along the super-diagonals in Table 3.6 on speedups).

Summary

- Collective communication
- Barrier, broadcast, reduction, gather, and scatter operations
- Example: Dot product
- MPI derived data types
- Performance evaluation