The Ministry of Education and Science of the Russian Federation

Lobachevsky State University of Nizhni Novgorod

Computing Mathematics and Cybernetics faculty

The competitiveness enhancement program
of the Lobachevsky State University of Nizhni Novgorod
among the world's research and education centers

Strategic initiative
“Achieving leading positions in the field of supercomputer technology
and high-performance computing”

Introduction to MPI

Lecture 4. Parallel Programming with MPI.
Collective Data Transmission Operations.

Nizhni Novgorod

2014

Lecture _4_. Parallel Programming with MPI. Collective Data Transmission Operations

This lecture is dedicated to MPI-based parallel programming methods for distributed memory systems. While Lecture 3 described the minimum required set of functions for MPI-based software development, this one describes a number of collective operations ensuring a more efficient data exchange. It also gives a program example demonstrating a way to handle collective operations.

The functions *MPI_Send* and *MPI_Recv*, discussed in lecture 3, provide for *pair* data passing operations between two parallel program processes. To execute *collective* communication operations characterized by participating of all the processes in a communicator, MPI provides a special set of functions. This subsection discusses a number of such functions.

To demonstrate the example of MPI function applications we will use the problem of summing up vector *x* elements

$$S = \sum_{i=1}^{n} x_i \ .$$

The development of parallel algorithm for solving this problem is not complicated. It is necessary to divide the data into equal blocks, to transmit these blocks to the processes, to carry out the summation of the obtained data in the processes, to collect the values of the computed partial sums on one of the processes and to add the values of partial sums to obtain the total result of the problem. In further development of the demontrational programs this algorithm will be simplified. All the vector being summed and, not only separate blocks of the vector, will be transmitted to the program processes.

4.1. Data Broadcasting

The first problem, which arises in the execution of the above discussed parallel algorithm is the need for transmitting vector *x* values to all the parallel program processes. Of course, it is possible to use the above discussed data transmission functions for solving the problem:

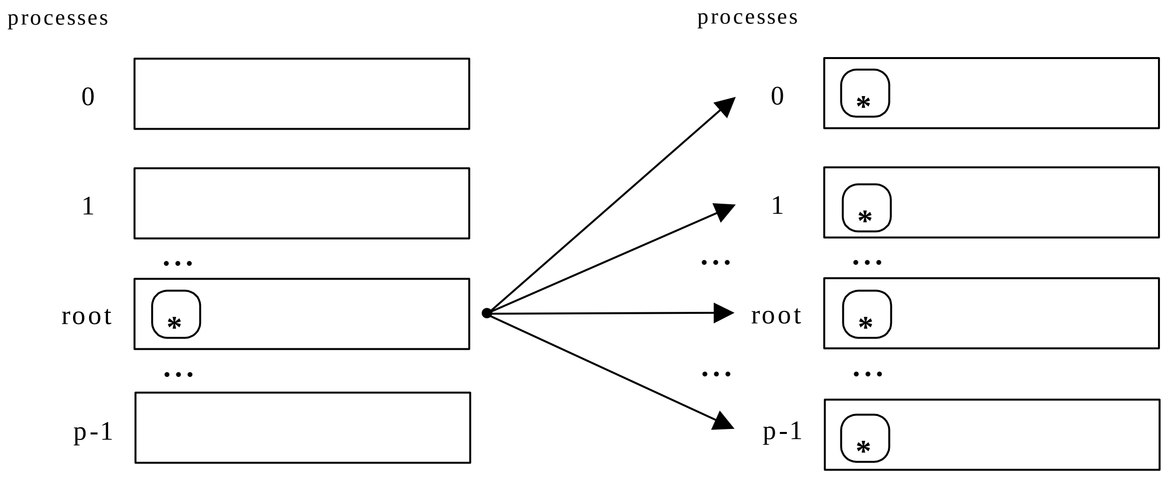
```
MPI_Comm_size(MPI_COMM_WORLD,&ProcNum);
for (i=1; i<ProcNum; i++)
  MPI_Send(&x,n,MPI_DOUBLE,i,0,MPI_COMM_WORLD);
```

However, this solution appears to be inefficient, as the repetition of the data transmission operations leads to summing up the expenses (latencies) on the preparation of the transmitted messages. Besides, as it has been shown in Lecture 3, this operation may be executed in log_2p data transmission iterations.

To achieve efficient broadcasting the following MPI function may be used:

```
    int MPI_Bcast(void *buf,int count,MPI_Datatype type,int
                          root,MPI_Comm comm),
where
  - buf, count, type – memory buffer, which contains the
     transmitted message (for the process 0) and for message
     reception for the rest of the processes,
  - root – the rank of the process, which carries out data
     broadcasing,
  - comm – the communicator, within of which data broadcasting
     is executed.
```

The function *MPI_Bcast* carries out transmitting the data from the buffer *buf*, which contains *count* type elements from the process with the *root* number to the processes within the communicator *comm* – see Figure 4.1



a) before the beginning of the operation b) after the completion of the operation

Figure 4.1. The general scheme of the data broadcasting operation

The following aspects should be taken into consideration:

1. The function *MPI_Bcast* defines the collective operation, and thus, the call of the function MPI_Bcast, when the necessary data should be transmitted, is to be executed by all the processes of a certain communicator (see further the example of the program),

2. The memory buffer defined in the function MPI_Bcast has different designations in different processes. For the root process, from which data broadcasting is performed, this buffer

should contain the transmitted message. For the rest of the processes the buffer is assigned for data reception.

We will give the example of the program for solving the problem of vector elements summation with the use of the above described function.

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main(int argc, char* argv[]){
  double x[100], TotalSum, ProcSum = 0.0;
  int  ProcRank, ProcNum, N=100;
  MPI_Status Status;

  //initialization
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD,&ProcRank);

  // data preparation
  if ( ProcRank == 0 ) DataInitialization(x,N);

// data broadcast
  MPI_Bcast(x, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  // computing the partial sum on each of the processes
  // vector x elements from i1 to i2 are summed at each
process
  int k = N / ProcNum;
  int i1 = k *   ProcRank;
  int i2 = k * ( ProcRank + 1 );
  if ( ProcRank == ProcNum-1 ) i2 = N;
  for ( int i = i1; i < i2; i++ )
    ProcSum  = ProcSum + x[i];

  // collecting partial sums on the process 0
  if ( ProcRank == 0 ) {
    TotalSum = ProcSum;
    for ( int i=1; i < ProcNum; i++ ) {
      MPI_Recv(&ProcSum,  1,  MPI_DOUBLE,  MPI_ANY_SOURCE,  0,
MPI_COMM_WORLD,
        &Status);
      TotalSum = TotalSum + ProcSum;
    }
  }
  else // all the processes send their partial sums
    MPI_Send(&ProcSum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);

  // result outprint
  if ( ProcRank == 0 )
    printf("\nTotal Sum = %10.2f",TotalSum);
  MPI_Finalize();
}
```

In the described program the function *DataInitialization* performs the preparation of the initial data. The necessary data may be input from the keyboard, read from a file, or generated by means of a random number generator. The preparation of the function is given to the reader for individual work.

4.2. Data Transmission from All the Processes to a Process. Reduction Operation.

The procedure of collecting and further data summation available in the above described program is an example of the high frequency operation of transmitting data from all the processes to a process. Different kinds of data processing are carried out over the collected values in this operation (to emphasize this fact the operation is often called as *data reduction*). As previously, the implementation of reduction by means of usual point-to-point data transmission operations appears to be inefficient and rather time-consuming. To execute data reduction in the best way possible, MPI provides the following function:

```
        int MPI_Reduce(void *sendbuf, void *recvbuf,int
                    count,MPI_Datatype type,
               MPI_Op op,int root,MPI_Comm comm),
where
```

- **sendbuf** – memory buffer with the transmitted message,
- **recvbuf** – memory buffer for the resulting message (only for the root rank process),
- **count** – the number of elements in the messages,
- **type** – the type of message elements,
- **op** – the operation, which should be carried out over the data,
- **root** – the rank of the process, on which the result must be obtained,
- **comm** – the communicator, within of which the operation is executed.

The operations predetermined in MPI (see Table 4.1) may be used as data reduction operations.

Table 4.1. The basic (predetermined) MPI operation types for data reduction functions

Operation	**Description**
MPI_MAX	The maximum value determination
MPI_MIN	The minimum value determination
MPI_SUM	The Determination of the sum of the values
MPI_PROD	The determination of the product of the values
MPI_LAND	The execution of the logical operation “AND” over the message values
MPI_BAND	The execution of the bit operation “AND” over the message values
MPI_LOR	The execution of the logical operation “OR” over the message values
MPI_BOR	The execution of the bit operation “OR” over the message values
MPI_LXOR	The execution of the excluding logical operation “OR” over the message values
MPI_BXOR	The execution of the excluding bit operation “OR” over the message

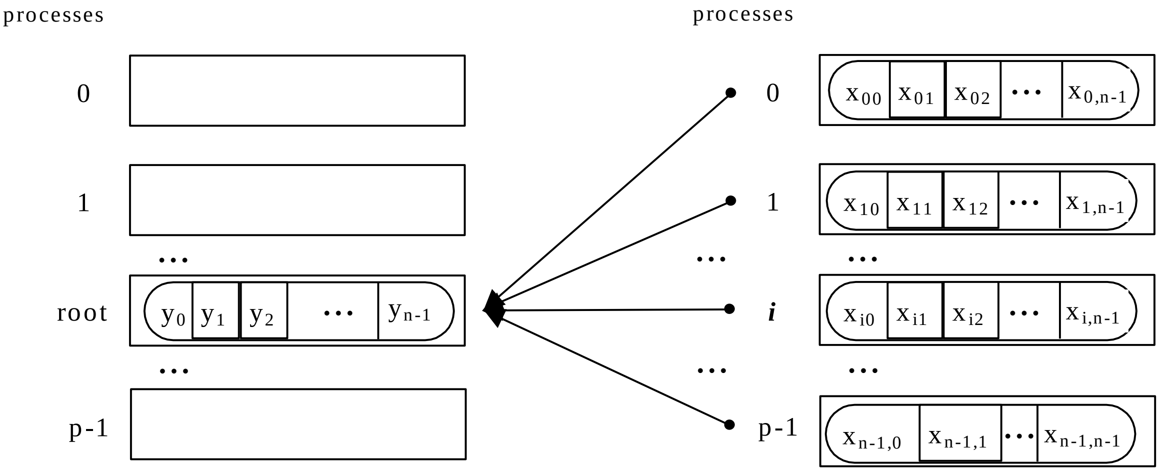
	values
MPI_MAXLOC	The determination of the maximum values and their indices
MPI_MINLOC	The determination of the minimum values and their indices

There may be other complementary operations determined directly by the user of the MPI library besides this given standard operation set (see, for instance, Group, et al. (1994), Pacheco (1996).

The general scheme of the execution of data collecting and processing operation on a processor is shown in Figure 4.1. The elements of the received message on the root process are the results of processing the corresponding elements of the messages transmitted by the processes, i.e.

$$y_j = \bigotimes_{i=0}^{n-1} x_{ij}, \; 0 \le j < n,$$

where ⊗ is the operation, which is set when the function *MPI_Reduce* is called (to clarify this we present an example of the execution of data reduction operation in Figure 4.2.).



a) after the termination of the operation

b) before the beginning of the operation

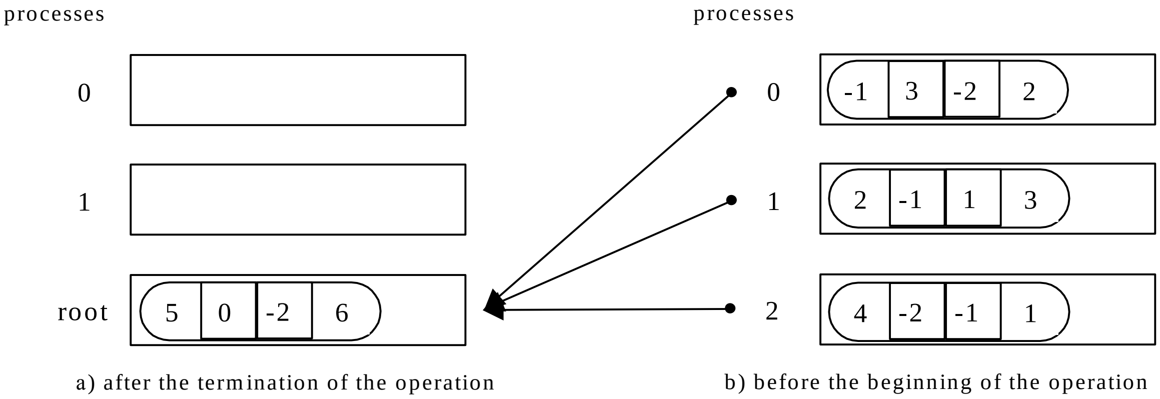
Figure 4.2. The general scheme of collecting and processing the data on a processor from all the other processes

The following aspects should be taken into consideration:

1. The function *MPI_Reduce* defines the collective operation, and thus, the call should be carried out by all the processes of the specified communicator. All the calls should contain the equal values of the parameters *count*, *type*, *op*, *root*, *comm,*

2. The data transmission should be carried out by all the processes. The operation result will be obtained only by the root rank process,

3. The execution of the reduction operation is carried out over separate elements of the transmitted messages. Thus, for instance, if messages contain two data elements each, and the summation operation MPI_SUM is executed, then the result will consist of the two values: the first will contain the sum of the first elements of all the transmitted messages, the second one will be equal to the sum of all the second message elements correspondingly.



a) after the termination of the operation b) before the beginning of the operation

Figure 4.3. The example of reduction in summing up the transmitted data for three processes (each message contains 4 elements, the messages are collected on rank 2 process)

Using the function *MPI_Reduce* to optimize the previously described program of summation the fragment marked by the double frame can be rewrite in the following form:

```
// collecting of partial sums on 0 rank process
MPI_Reduce(&ProcSum,&TotalSum,  1,  MPI_DOUBLE,  MPI_SUM,  0,
MPI_COMM_WORLD);
```

4.3. Computation Synchronization

Sometimes the computational processes executed independently have to be sinchronized. Thus, for instance, to measure the starting time of the parallel program operation it is necessary to complete all the preparatory operations for all the processes simultaneously. Before the program termination all the processes should complete their computations etc.

Process synchronization, i.e. simultaneous achieving certain points of computations by various processes is provided by means of the following MPI function

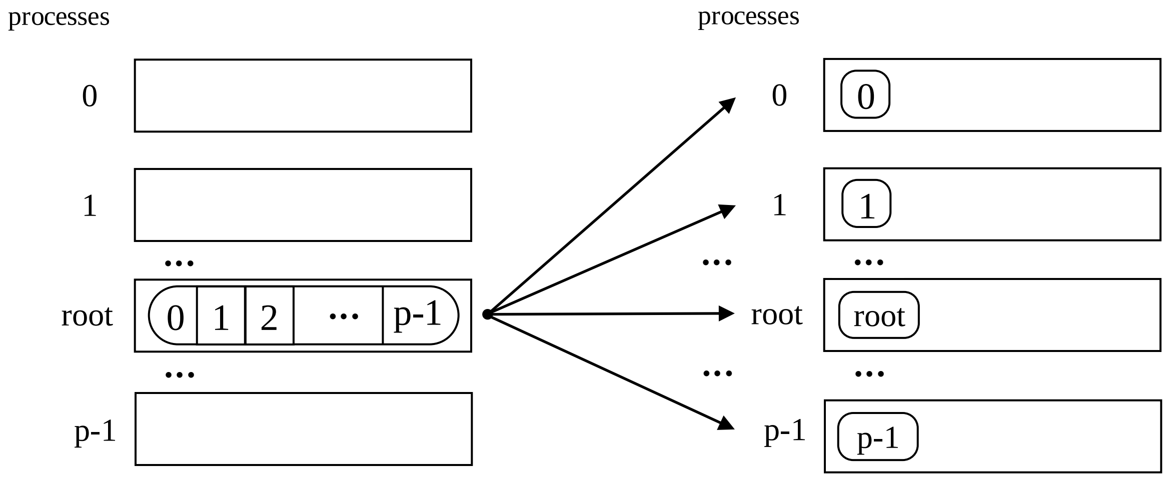
```
int MPI_Barrier(MPI_Comm comm);
```

The function *MPI_Barrier* defines collective operation. Thus, this function should be called by all the processes of the communicator. When the function *MPI_Barrier* is called, the process execution is blocked. The computations of the process will continue only after the function *MPI_Barrier* is called by all the processes of the communicator.

4.3.1. Scattering Data from a Process to all the Processes

The difference between scattering data from a process to all the processes (*data distribution*) and the broadcasting operation is that in case of scattering data, the process transmits different data to the processes (see Figure 4.4). The execution of this operation may be provided by means of the following function:

```
int MPI_Scatter(void *sbuf,int scount,MPI_Datatype stype,
                void *rbuf,int rcount,MPI_Datatype rtype,
                int root, MPI_Comm comm),
where
 - sbuf, scount, stype – the parameters of the transmitted
message
        (scount defines the number of elements transmitted to
each process),
 - rbuf, rcount, rtype – the parameters of the message received
in the
         processes,
 - root – the rank of the process, which performs data
scattering,
 - comm – the communicator, within of which data scattering is
           performed.
```



a) before the beginning of the operation b) after the termination of the operation

Figure 4.4. The general scheme of scattering data from a process to all the other processes

When this function is called, the process with rank *root* transmits data to all the other processes in the communicator. *Scount* elements will be sent to each process. the process 0 will receive the data block of *sbuf* elements with the indices from 0 to *scount-1*; the process with rank 1 will receive the block of elements with the indices from *scount* to *2*scount-1* etc. Thus, the total size of the transmitted message should be equal to *scount*p* elements, where *p* is the number of the processes in the communicator *comm*.

It should be noted that as the function *MPI_Scatter* defines a collective operation, the call of this function in the execution of data scattering should be provided in each communicator process.

It should be also noted that the function *MPI_Scatter* transmits messages of the same size to all the processes. The execution of a more general variant of data distribution operation, when the message sizes for different processes may be different, is provided by means of the function *MPI_Scatterv*.

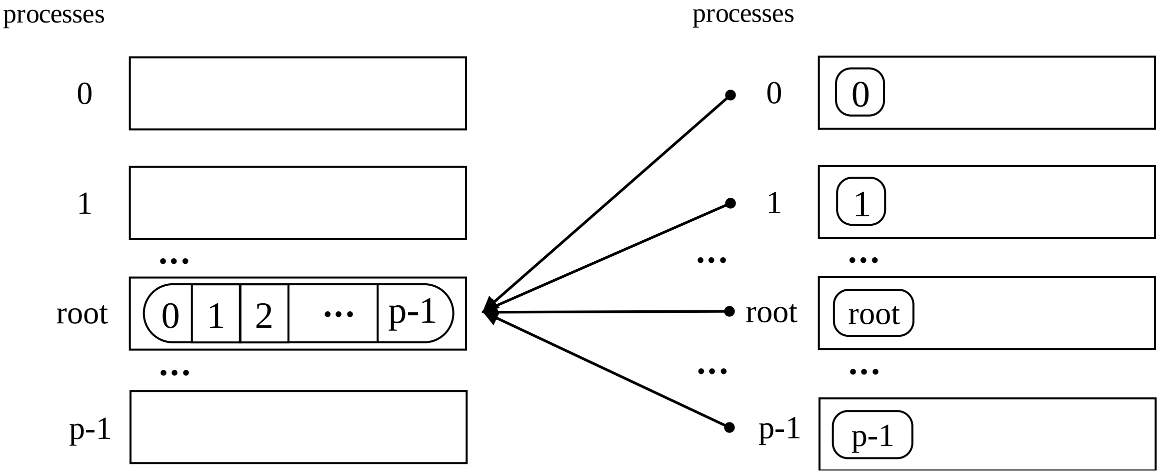
4.4. Gathering Data from All the Processes to a Process

Gathering data from all the processes to a process (*data gathering*) is reverse to data distribution (see Figure 4.5). The following MPI function provides the execution of this operation:

```
int MPI_Gather(void *sbuf,int scount,MPI_Datatype stype,
               void *rbuf,int rcount,MPI_Datatype rtype,
                int root, MPI_Comm comm),
```

where

- **sbuf**, **scount**, **stype** – the parameters of the transmitted message,
- **rbuf**, **rcount**, **rtype** – the parameters of the received message,
- **root** – the rank of the process which performs data gathering,
- **comm** – the communicator, within of which data transmission is executed.



a) after the termination of the operation b) before the beginning of the operation

Figure 4.5. The general scheme of the data gathering from all the processes to a process

When the fuinction *MPI_Gather* is being executed, each process in the communicator transmits the data from the buffer *sbuf* to the process with rank *root*. The *root* rank process gathers all the transmitted data in the buffer *rbuf* (the data is located in the buffer in accordance with the ranks of the sending processes). In order to locate all the received data the buffer size *rbuf* should be equal to *scount*p* elements, where *p* is the number of the processes in the communicator *comm*.

The function *MPI_Gather* also defines a collective operation and its call in gathering the data must be provided in each communicator process.

It should be noted that when the *MPI_Gather* function is used, data gathering should be carried out only on one process. To obtain all the gathered data on each communicator process, it is necessary to use the function:

```
  int MPI_Allgather(void *sbuf, int scount, MPI_Datatype stype,
                 void *rbuf, int rcount, MPI_Datatype rtype,
MPI_Comm comm).
```

The execution of the general variant of data collecting operation, when the sizes of the messages transmitted among the processes may be different, is provided by means of the functions *MPI_Gatherv* and *MPI_Allgatherv*.

4.5. Collective operation use example

The π computation problem will serve as an example of problem requiring distribution.

Theoretically, computation of π can be resolved to computation of an integral:

$$\pi = \int_0^1 \frac{4}{1+x^2}dx$$

This integral may be computed numerically using the method of rectangles. For the purposes of implementation, all processes, first of all, have to know the number of sections the original integration interval will be split into. Then each process must compute the integral value within its integration section. In the end, the zero process will sum all the integral parts that have been computed. See the implementation below.

```
#include "mpi.h"
#include <math.h>
double f(double a) {
  return (4.0 / (1.0 + a*a));
}
int main(int argc, char *argv) {
  int ProcRank, ProcNum, done = 0, n = 0, i;
  double PI25DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x, t1, t2;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD,&ProcRank);
  while (!done ) { // основной цикл вычислений
    if ( ProcRank == 0) {
      printf("Enter the number of intervals: ");
      scanf("%d",&n);
      t1 = MPI_Wtime();
    }
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
  if (n > 0) { // вычисление локальных сумм
    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = ProcRank + 1; i <= n; i += ProcNum) {
      x = h * ((double)i  0.5);
      sum += f(x);
    }
    mypi = h * sum;
    // сложение локальных сумм (редукция)

MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
```

```
   if ( ProcRank == 0 ) { // вывод результатов
    t2 = MPI_Wtime();
    printf("pi is approximately %.16f, Error is
        %.16f\n",pi, fabs(pi  PI25DT));
    printf("wall clock time = %f\n",t2-t1);
   }
  } else done = 1;
 }
 MPI_Finalize();
}
```

4.6. Review of references

There are a number of sources, which provide information about MPI. First of all, this is the internet resource, which describes the standard MPI: http://www.mpiforum.org. One of the most widely used MPI realizations, the library MPICH, is presented on http://www-unix.mcs.anl.gov/mpi/mpich (the library MPICH2 with the realization of the standard MPI-2 is located on http://www-unix.mcs.anl.gov/mpi/mpich2).

The following works may be recommended: Group, et al. (1994), Pacheco (1996), Snir, et al. (1996), Group, et al. (1999a). The description of the standard MPI-2 may be found in Group, et al. (1999b).

We may also recommend the work by Quinn (2003), which described a number of typical problems of parallel programming for the purpose of studying MPI. These are the problems of matrix computations, sorting, graph processing etc.

Test questions

1. What is the difference between point-to-point and collective data transmission operations?
2. Which MPI function provides transmitting data from a process to all the processes?
3. What is the data reduction operation?
4. In what cases should we apply barrier synchronization?
5. What collective data transmission operations are supported in MPI?

Practice

1. Develop a program for finding the minimum (maximum) value of the vector elements.
2. Develop a program for computing the scalar product of two vectors.
3. Develop a program for definite integral computation using the method of rectangles.

$$y = \int_{a}^{b} f(x)\,dx \approx h \sum_{i=0}^{N-1} f_i \,, f_i = f(x_i), x_i = i\,h, h = \frac{b-a}{N}$$

References

1. Pacheco, P. (1996). Parallel Programming with MPI. - Morgan Kaufmann.
2. Gropp, W., Lusk, E., Skjellum, A. (1999a). Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface (Scientific and Engineering Computation). - MIT Press.
3. Gropp, W., Lusk, E., Thakur, R. (1999b). Using MPI-2: Advanced Features of the Message Passing Interface (Scientific and Engineering Computation). - MIT Press.
4. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J. (1996). MPI: The Complete Reference. - MIT Press, Boston, 1996.