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 3. MPI-Based Parallel Programming. Basic Data Transmission Operations.

Nizhni Novgorod
2014

Lecture 3_. MPI-Based Parallel Programming. Basic Data Transmission Operations

This lecture is dedicated to basic parallel programming methods for distributed memory computer systems based on MPI.

The processors in the computer systems with distributed memory work independently of each other. It is necessary to have a possibility to distribute the computational load and to organize the information interaction (data transmission) among the processors in order to arrange parallel computations under these conditions.

The solution to the above mentioned problems is provided by *message passing interface - MPI*.

1. Generally, it is necessary to analyze the algorithm of solving a problem, to select the fragments of the computations, which are independent with regard to the information, in order to distribute the computations among the processors. It is also necessary to perform the program implementation of the fragments and then distribute the obtained program parts on different processors. A simpler approach is used in MPI. According to this approach, *a program is developed for solving the stated problem and this single program is executed simultaneously on all the available processors*. In order to avoid the identity of computations on different processor, it is first of all possible to substitute different data for executing the program on different processors. Secondly, there are means in MPI to identify the processor, on which the program is executed. Thus, it provides the possibility to organize the computations differently depending on the processor used by the program. This method of organizing parallel computations is referred to as the model *single program multiple processes* or *SPMP*(1).

2. The operation of data communication is sufficient for the minimum variant of organizing the information interaction among the processors (it should also be technically possible to provide communication among the processors –we mean the availability of channels or communication lines). There are many data transmission operations in MPI. They provide various means of data passing and implement practically all the communication operations discussed in Section 3. These possibilities are the main advantages of MPI (the very name of the method MPI testifies to it).

It should be noted that the attempts to create software for data transmission among the processors began practically right after the local computer networks came into being. Such software,

---

(1) It is more often referred to as *single program multiple data* or *SPMD*. With regard to MPI it would be more logical to use the term SPMP.

for instance, are described in Buyya (1999), Andrews (2000) and many others. This software, however, were often incomplete and, moreover, incompatible. Thus, the portability of programs in case of transferring the software to other computer systems is one of the most serious problems in software development. This problem is strongly felt in development of parallel programs. As a result, the efforts to standardize the software for organizing message transmission in multiprocessor computational systems have been made since the 90-s. The work, which directly led to the creation of MPI, was initiated by the Workshop on Standards for Message Passing in a Distributed Memory Environment, Williamsburg, Virginia, USA, April 1992. According to the results of the workshop a special workgroup was created, which was later transformed into an international community (*MPI Forum*). The result of its activity was the creation and adaptation in 1994 of the *Message Passing Interface* standard, version 1.0.

So now it is reasonable to explain the concept MPI. First of all, MPI is a standard for organizing message passing. Secondly, MPI is the software, which should provide the possibility of message passing and correspond to all the requirements of MPI standard. According to the standard this software should be arranged as program module libraries (*MPI libraries*) and should be comprehensible for the most widely used algorithmic languages C and Fortran. This "duality" of MPI should be taken into account while using terminology. As a rule, the abbreviation MPI is used to refer to the standard and the phrase "MPI library" points to a standard software implementation. However, the term MPI is often used to denote MPI libraries also, and thus, the correct interpretation of the term depends on the context.

It must be noted that on the one hand MPI is complex enough, as the MPI standard provides for more than 125 functions. On the other hand, MPI has an elaborate structure: one can start developing parallel software as soon as only 6 MPI functions have been studied. All the advanced MPI capabilities may be mastered as the developed algorithms and programs become more complex. This is the structure - simple-to-complex – the MPI training materials will have.

Before we started to describe MPI in details, we find useful to mention some of its advantages:

- MPI makes possible to decrease considerably the complexity of the parallel program portability among different computer systems. A parallel program developed in the algorithmic languages C or Fortran with the use of MPI library will, as a rule, operate on different computer platforms,
- MPI contributes to the increase of parallel computation efficiency, as there are MPI library implementations for practically every type of computer system nowadays. These realizations are carried out with regard to the possibilities of the hardware being used,

- MPI decreases the complexity of parallel program development as on the one hand, the greater part of the basic data transmission operations discussed in Lecture 3, 4 are provided by MPI standard. On the other hand, there are many parallel numerical libraries available nowadays created with the use of MPI.

3.1. MPI: Basic Concepts and Definitions

Let us discuss a number of concepts and definitions, which are fundamental for MPI standard.

3.1.1. The Concept of Parallel Program

Within the frame of MPI a *parallel program* means a number of simultaneously carried out *processes*. The processes may be carried out on different processors. Several processes may be located on a processor (in this case they are carried out in the time-shared mode). In the extreme case a single processor may be used to carry out a parallel program. As a rule, this method is applied for the initial testing of the parallel program correctness.

Each parallel program process is generated on the basis of the copy of the same program code (SPMP model). This program code presented as an executable file must be accessible at the moment when the parallel program is started on all the processors used. The source program code for the program being executed is developed in the algorithmic languages C or Fortran with the use of some MPI library implementation.

The number of processes and the number of the processors used are determined at the moment of the parallel program launch by the means of MPI program execution environment. These numbers must not be changed in the course of computations (the standard MPI-2 provides some means to change dynamically the number of executed processes). All the program processes are sequentially enumerated from 0 to $p-1$, where np is the total number of processes. The process number is referred to as the *process rank*.

3.1.2. Data Communication Operations

Data communication operations form the basis of MPI. Among the functions provided within MPI they usually differentiate between *point-to-point* operations, i.e. operations between two processes, and *collective* ones, i.e. communication actions for the simultaneous interaction of several processes.

This lecture reviews only basic pair data transmission operations.

3.1.3. Communicators

Parallel program processes are united into *groups*. *The communicator* in MPI is a specially designed service object, which unites within itself a group of processes and a number of complementary parameters (*context*), which are used in carrying out data transmission operations.

As a rule, point-to-point data transmission operations are carried out for the processes, which belong to the same communicator. Collective operations are applied simultaneously to all the processes of the communicator. As a result, it is necessary without fail to point to the communicator being used for data transmission operations in MPI.

In the course of computations new groups may be created and the already existing groups of processes and communicators may be deleted. The same process may belong to different groups and communicators. All the processes available in a parallel program belong to the communicator with the identifier MPI_COMM_WORLD, which is created on default.

If it is necessary to transmit the data among the processes, which belong to different groups, an intercommunicator should be created.

3.1.4. Data Types

It is necessary to point to the type of the transmitted data in MPI functions while carrying out data transmission operations. MPI contains a wide set of the basic data types. These data types largely coincide with the data types of the algorithmic languages C and Fortran. Besides this, MPI has possibilities for creating new derived data types for more accurate and precise description of the transmitted message content.

3.1.5. Virtual Topologies

As it has been mentioned previously, point-to-point data transmission operations may be carried out among any processors of the same communicator; all the processors of the communicator take part in collective operations. In this respect, the logical topology of the communication lines among the processes is a complete graph (regardless of the availability of real physical communication channels among the processors).

It is also useful to logically present the available communication network as a topology for further description and analysis of a number of parallel algorithms.

3.2. Introduction to MPI Based Development of Parallel Programs

3.2.1. The Fundamentals of MPI

Let us describe the minimum necessary set of MPI functions sufficient for the development of rather simple parallel programs.

3.2.1.1. MPI Programs Initialization and Termination

The first MPI function, which has to be called, must be the following:

```
int MPI_Init ( int *agrc, char ***argv ).
```

It is called to initialize MPI program execution environment. The parameters of the function are the number of arguments in the command line and the command line text.

The last MPI function to be called must be the following one:

```
int MPI_Finalize (void).
```

As a result, it may be noted that the structure of the MPI-based parallel program should look as follows:

```
#include "mpi.h"
int main ( int argc, char *argv[] ) {
  <program code without the use of MPI functions>
  MPI_Init ( &agrc, &argv );
    <program code with the use of MPI functions>
  MPI_Finalize();
  <program code without the use of MPI functions>
  return 0;
}
```

It should be noted:

1. File *mpi.h* contains the definition , function prototypes and data types of MPI library,
2. Functions *MPI_Init* and *MPI_Finalize* are mandatory and should be executed (only once) by each process of the parallel program,
3. Before *MPI_Init* is called, the function *MPI_Initialized* may be used to determine whether *MPI_Init* has been called or not.

The examples of the functions, which have been discussed, allow us to understand the syntax of MPI function naming. Prefix MPI precedes the name of the function. Further, there are one or more words of the function name. The first word in the function name begins with the capital symbol. The words are separated by the underline character. The names of MPI functions, as a rule, explain the designation of the operations carried out by the functions.

3.2.1.2. Determining the Number and the Rank of the Processes

The number of the processes in the parallel program being executed is carried out by means of the following function:

```
int MPI_Comm_size ( MPI_Comm comm, int *size ).
```

The following function is used to determine the process rank:

```
int MPI_Comm_rank ( MPI_Comm comm, int *rank ).
```

As a rule, the functions *MPI_Comm_size* and *MPI_Comm_rank* are called right after *MPI_Init*:

```
#include "mpi.h"
int main ( int argc, char *argv[] ) {
  int ProcNum, ProcRank;
  <program code without the use of MPI functions>
  MPI_Init ( &agrc, &argv );
  MPI_Comm_size ( MPI_COMM_WORLD, &ProcNum);
  MPI_Comm_rank ( MPI_COMM_WORLD, &ProcRank);
    <program code with the use of MPI functions>
  MPI_Finalize();
  <program code without the use of MPI functions>
  return 0;
}
```

It should be noted:

1. Communicator *MPI_COMM_WORLD*, as it has been previously mentioned, is created on default and presents all the processes carried out by a parallel program,

2. The rank obtained by means of the function *MPI_Comm_rank* is the rank of the process, which has called this function, i.e. the variable *ProcRank* will accept different values in different processes.

3.2.1.3. Message Passing

In order to transmit data, the sending process should carry out the following function:

```
int MPI_Send(void *buf, int count, MPI_Datatype type, int
dest,
  int tag, MPI_Comm comm),
where
```

```
    - buf   - the address of the memory buffer, which contains
             the data
             of the message to be transmitted,
    - count - the number of the data elements in the message,
    - type  - the type of the data elements of the transmitted
              message,
    - dest  - the rank of the process, which is to receive the
              message,
    - tag   - tag-value, which is used to identify messages,
    - comm  - the communicator, within of which the data is
              transmitted.
```

There are a number of basic types to refer to the transmitted data in MPI. The complete list of the basic types is given in Table 3.1.

Table 3.1. The basic (predefined) MPI data types for the algorithmic language C

MPI_Datatype	**C Datatype**
MPI_BYTE	
MPI_CHAR	signed char
MPI_DOUBLE	double
MPI_FLOAT	float
MPI_INT	int
MPI_LONG	long
MPI_LONG_DOUBLE	long double
MPI_PACKED	
MPI_SHORT	short
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

The following issues should be taken into account:

1. The message to be passed is defined by pointing to the memory block (buffer), which contains the message. The following triad is used to point to the buffer

```
   ( buf, count, type )
```

This triad is included into the parameters of practically all data passing functions,

2. The processes, among which data is passed, should belong to the communicator, specified in the function *MPI_Send*,

3. The parameter *tag* is used only when it is necessary to distinguish the messages being passed. Otherwise, an arbitrary integer number may be used as the parameter value (see also the description of the function MPI_*Recv*).

Immediately after the completion of the function *MPI_Send* the sending process may start to use repeatedly the memory buffer, in which the transmitted message was located. But it should be noted that the state of the message being passed may be quite different at the moment of the function MPI_Send termination. The message may be located in the sending process, it may be being transmitted, it may be stored in the receiving process, or may be received by the receiving process by means of the function MPI_Recv. Thus, the termination of the function MPI_Send means only that the data transmission operation has started to be carried out, and message passing will sooner or later be completed.

The example of this function use will be presented after the description of the function MPI_Recv.

3.2.1.4. Message Reception

In order to receive messages the receiving process should carry out the following function:

```
   int  MPI_Recv(void  *buf,  int  count,  MPI_Datatype  type,  int
source,
     int tag, MPI_Comm comm, MPI_Status *status),
   where
     - buf, count, type – the memory buffer for message
         reception, the designation of each separate parameter
         corresponds to the description in MPI_Send,
     - source – the rank of the process, from which message is
              to be received,
     - tag    - the tag of the message, which is to be received
              for the process,
     - comm   - communicator, within of which data is passed,
     - status – data structure, which contains the information
              of the results of carrying out the data
              transmission operation.
```

The following aspects should be taken into account:

1. Memory buffer should be sufficient for data reception and the element types of the transmitted and the received message must coincide. In case of memory shortage a part of the

message will be lost and in the return code of the function termination there will be an overflow error registered,

2. The value *MPI_ANY_SOURCE* may be given for the parameter source, if there is a need to receive a message from any sending process,

3. If there is a need to receive a message with any tag, then the value *MPI_ANY_TAG* may be given for the parameter *tag*,

4. The parameter *status* makes possible to define a number of characteristics of the received message:

```
     - status.MPI_SOURCE – the rank of the process, which has
sent the received message,
     - status.MPI_TAG    - the tag of the received message.
```

The function

```
   MPI_Get_count(MPI_Status *status, MPI_Datatype type, int
*count)
```

returns in the variable *count* the number of elements of datatype *type* in the received message.

The call of the function *MPI_Recv* does not have to be in agreement with the time of the call of the corresponding message passing function *MPI_Send*. The message reception may be initiated before the moment, at the moment or even after the starting moment of sending the message.

After the termination of the function *MPI_Recv* the received message will be located in the specified buffer memory. It is essential that the function *MPI_Recv* is a *blocking* one for the receiving process, i.e. carrying out of the process is suspended till the function terminates its operation. Thus, if due to any reason the expected message is missing, then the parallel program execution will be blocked.

3.2.1.5. The First MPI Based Parallel Program

The described set of functions appears to be sufficient for parallel program development[2]. The program, which is given below, is standard example for the algorithmic language C.

```
  #include <stdio.h>
  #include "mpi.h"
  int main(int argc, char* argv[]){
```

---

[2] As it has been stated previously, the number of MPI functions, which are necessary to start the parallel program development, appears to be equal to six.

```
    int ProcNum, ProcRank, RecvRank;
    MPI_Status Status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
    if ( ProcRank == 0 ){
      // Code is executed only by the process of
rank 0
      printf ("\n Hello from process %3d",
ProcRank);
      for ( int i=1; i<ProcNum; i++ ) {
        MPI_Recv(&RecvRank, 1, MPI_INT,
MPI_ANY_SOURCE,
          MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
        printf("\n Hello from process %3d",
RecvRank);
      }
    }
    else
      // Message is sent by all the processes
except
      // the process of rank 0

MPI_Send(&ProcRank,1,MPI_INT,0,0,MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
  }
```

As it is clear from the program text, each process identifies its rank, and after that all the operations in the program are separated. All the processes, except the process of rank 0, send the value of its rank to the process 0. The process of rank 0 first prints the value of its rank and then prints ranks of all the other processes. It should be noted that the order of message reception is not predetermined. It depends on the execution conditions for parallel program (moreover, the order may change in several sequential program executions). Thus, a possible variant of the process 0 print results may consist in the following (for the parallel program of 4 processes):

```
   Hello from process 0
   Hello from process 2
   Hello from process 1
   Hello from process 3
```

This “changeable” type of the obtained results complicates the developing, testing and debugging parallel programs, as one of the main programming principles, i.e. ”the reproducibility” of the executed computational experiments, does not work in this case. As a rule, if it does not lead to efficiency losses, it is necessary to provide the unambiguity of computations in case of parallel computations. Thus, for the simple example being discussed it is possible to provide the permanence of the obtained result using the rank of the sending process in the message reception operation:

```
  MPI_Recv(&RecvRank,     1,     MPI_INT,     i,     MPI_ANY_TAG,
MPI_COMM_WORLD, &Status).
```

The rank of the sending process regulates the order of message reception. As a result, the lines of printing will appear strictly in the order of increasing the process ranks (we should remind that this regulation in certain situations may slow down the parallel computations).

One more important issue should be mentioned. An MPI based program both in this particular variant and in the most general case is used for generating all the processes of the parallel program. As a result, it should define the computations carried out on all these processes. It is possible to say that an MPI based program is a certain “*macro code”*. Different parts of this code are used by different processes. Thus, for instance, in the example being discussed, the parts of the program code marked by double frame are not used simultaneously in either of the processes. The first marked part with the sending function MPI_Send is used only by the process of rank 0. The second part with the reception function MPI_Recv is used by all the processes except the process of rank 0.

The approach, which is used in the above described program for distributing the code fragments among the processes, can be described as follows. First, the process rank is defined by means of the function *MPI_Comm_rank*. The program code parts necessary for the process are selected in accordance with the rank. The availability of code fragments of different processes in the same program complicates significantly understanding and developing an MPI based program. It is recommended to carry out the program code of different processes into separate program modules (functions), if the amount of the developed programs is significant. The general scheme of an MPI based program in this case looks as follows:

```
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
```

```
if ( ProcRank == 0 ) DoProcess0();
else if ( ProcRank == 1 ) DoProcess1();
else if ( ProcRank == 2 ) DoProcess2();
```

In many cases, as in the above described examples, the operations differ only for the process of rank 0. In this case the general scheme of an MPI based program can be simpler:

```
MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
if ( ProcRank == 0 ) DoManagerProcess();
else DoWorkerProcesses();
```

In conclusion let us comment on the approach, which is used in MPI to control the correctness of function execution. All the MPI functions return the termination code as their value. If the function is completed successfully the return code is *MPI_SUCCESS*. The other values of the termination code testifies to the fact that some errors have been discovered in the course of function execution. To find out the type of the discovered error predetermined named constants are used. Among these constants there are the following ones:

```
- MPI_ERR_BUFFER – incorrect buffer pointer,
- MPI_ERR_COMM – incorrect communicator,
- MPI_ERR_RANK – incorrect process rank,
```

and others. The complete list of the constants for checking the termination code is in file *mpi.h*.

3.2.2. The Determination of the MPI Based Program Execution Time

The need for determining the execution time to estimate the speedup of parallel computations arises practically after the development of the first parallel program. The means, which are used to measure the execution time depend, as a rule, on the hardware, the operating system, the algorithmic language and so on. The MPI standard includes the definition of special functions for measuring time. The use of these functions makes possible to eliminate the dependence on the environment of the parallel program execution.

Obtaining time of the current moment of the program execution is provided by means of the following function:

```
double MPI_Wtime(void).
```

The result of its call is the number of seconds, which have passed since a certain moment in the past. This moment of time in the past may depend on the environment of MPI library implementation and thus, the function MPI_Wtime should be used in order to eliminate this dependence only for determining the duration of execution of parallel program code fragments. A possible scheme of application MPI_Wtime function may consist in the following:

```
double t1, t2, dt;
t1 = MPI_Wtime();
…
t2 = MPI_Wtime();
dt = t2 - t1;
```

The accuracy of time measurement may depend on the environment of the parallel program execution. The following function may be used in order to determine the current value of accuracy:

```
double MPI_Wtick(void).
```

This function allows to measure in seconds the time between two sequential ticks of time of the computer system hardware timer.

3.3. 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 minimum set of operations of sufficient for the organization of parallel computations in the distributed memory systems?
2. Why is it important to standardize message passing?
3. How can a parallel program be defined?
4. What are the difference between the concepts of “process” and “processor”?
5. What minimum set of MPI functions makes possible to start the development of parallel programs?
6. How are the messages being passed described?
7. How do we organize the reception of messages from concrete processes?
8. How do we determine the execution time of an MPI based program?

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, where two processes repeatedly exchange messages of *n* byte length. Carry out the experiments and estimate the dependence of the data operation execution time against the message length.

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.