


Lobachevsky State University of Nizhni Novgorod

Computing Mathematics and Cybernetics faculty

The competitiveness enhancement program among the world's research and education centers

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

Lobachevsky State University of Nizhni Novgorod

Computing Mathematics and Cybernetics faculty

Parallel Programming for Multiprocessor Distributed Memory Systems

02 Lecture
Collective and Point-to-Point Communications

With the support of Microsoft

Sysoyev A.V.
Software department

Contents

COLLECTIVE COMMUNICATIONS

- Data Broadcasting
- Reduction Operations
- Example: Calculating the Constant π
- Scattering and Gathering
- Example: Calculating the Inner Product
- All to All Communications
- Computation Synchronization

Collective Communications…
Data Broadcasting

- To efficiently broadcast the data between processes the following MPI function should be used

```
int MPI_Bcast(void *buf, int count, MPI_Datatype type,
  int root, MPI_Comm comm);
```

- **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 in the message
- **root** – the rank of the process, which carries out data broadcasting
- **comm** – the communicator, within of which the data is transmitted

- The function **MPI_Bcast()** carries out transmitting the data from the buffer **buf**, which contains **count** type elements, from the processor with the rank **root** to the processes within the communicator **comm**

Collective Communications…
Data Broadcasting

- The function **`MPI_Bcast()`** is the collective operation, and thus, the call of this function is to be executed by all the processes of the communicator **`comm`**
- The memory buffer pointed 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 intended for data receiving

Collective Communications…
Data Reduction

- To “reduce” some data from all processes to chosen one the following MPI function can be used

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

- **sendbuf** – memory buffer with the transmitted message
- **recvbuf** – memory buffer with the resulting message (only for the root process)
- **count** – the number of the data elements in the message
- **type** – the type of the data elements in the message
- **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

Collective Communications…
Data Reduction

- The Basic MPI Operation Types for Data Reduction

Operation	Description
MPI_MAX	The maximum value calculation
MPI_MIN	The minimum value calculation
MPI_SUM	The calculation of the sum of the values
MPI_PROD	The calculation 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 calculation of the maximum values and their indices
MPI_MINLOC	The calculation of the minimum values and their indices

Collective Communications…
Data Reduction

- The function **`MPI_Reduce()`** is the collective operation, and thus, the function call should be carried out by all the processes of the communicator **`comm`**.
- All the calls should contain the same values of the parameters **`count`**, **`type`**, **`op`**, **`root`**, **`comm`**
- The data transmission should be carried out by all the processes.
- The operation result will be obtained only by **`root`** process,
- The execution of the reduction operation is carried out over separate elements of the transmitted messages

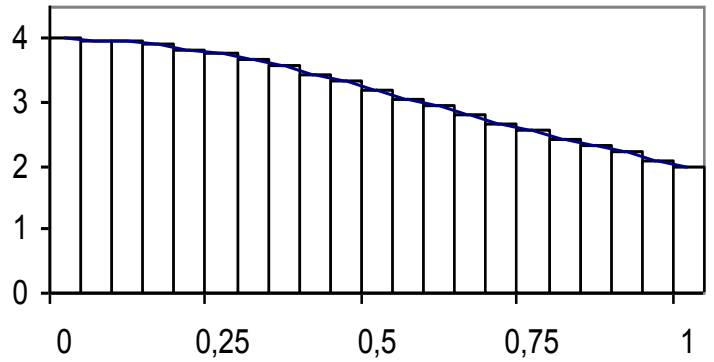
Collective Communications…

Example: Calculating the Constant π

- The value of constant π can be computed by means of the integral

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

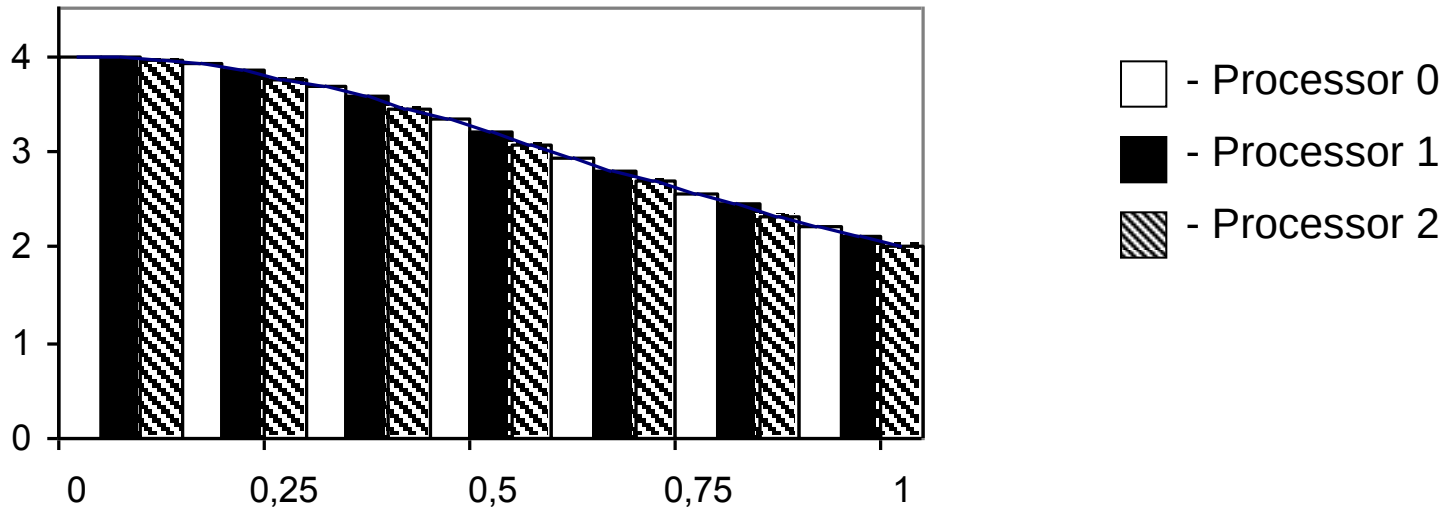
- To compute this integral the method of rectangles can be used for numerical integration



Collective Communications…

Example: Calculating the Constant π

- Cyclic scheme can be used to distribute the calculations among the processors
- Partial sums, that were calculated on different processors, have to be summed



Collective Communications…

Example: Calculating the Constant π

```
#include "mpi.h"
#include <math.h>
double f(double a){
  return (4.0 / (1.0 + a*a));
}
void main(int argc, char *argv[]){
  int ProcRank, ProcNum, done = 0, n = 0, i;
  double PI25DT = 3.14159265358979323846264643;
  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){ // main calculation loop
    if (ProcRank == 0){
      printf("Enter the number of intervals: ");
      scanf("%d", &n);
      t1 = MPI_Wtime();
    }
```

Collective Communications…

Example: Calculating the Constant π

```
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n > 0){ // calculating the local sums
      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;
      MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
        MPI_COMM_WORLD);
      if (ProcRank == 0){ // printing results
        t2 = MPI_Wtime();
        printf("pi = %.15f, Error = %.15f\n", pi, fabs(pi - PI25DT));
        printf("time = %f\n", t2 - t1);
      }
    }
    else done = 1;
  }
  MPI_Finalize();
}
```

Collective Communications…
Scattering and Gathering

- To distribute (“scatter”) some data from chosen process to all the processes the following MPI function can be used

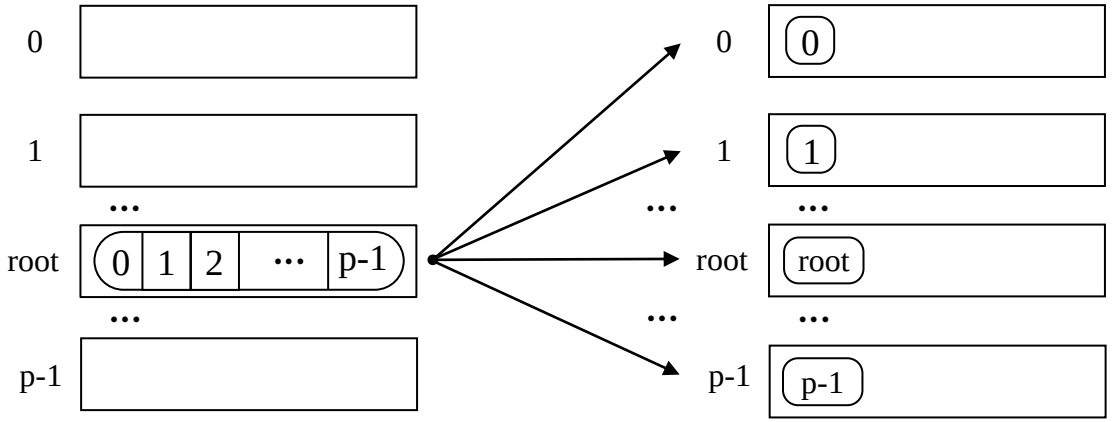
```
int MPI_Scatter(void *sbuf, int scount, MPI_Datatype stype,
                void *rbuf, int rcount, MPI_Datatype rtype,
                int root, MPI_Comm comm);
```

- **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 received message
- **root –** the rank of the process, on which the result must be obtained
- **comm –** the communicator, within of which the operation is executed

- When the message sizes for different processes may be different, the execution of data scattering is provided by means of the function **MPI_Scatterv()**

Collective Communications…
Scattering and Gathering

- The function **`MPI_Scatter()`** should be called by all the processes of the communicator **`comm`**.
- The **`root`** process transmits the equal sized (**`scount`**) messages from the buffer **`sbuf`** to all the processes
- Each process (including **`root`**) receives the message of **`rcount=scount`** length into the buffer **`rbuf`**



a) Before the operation

b) After the operation

Collective Communications…
Scattering and Gathering

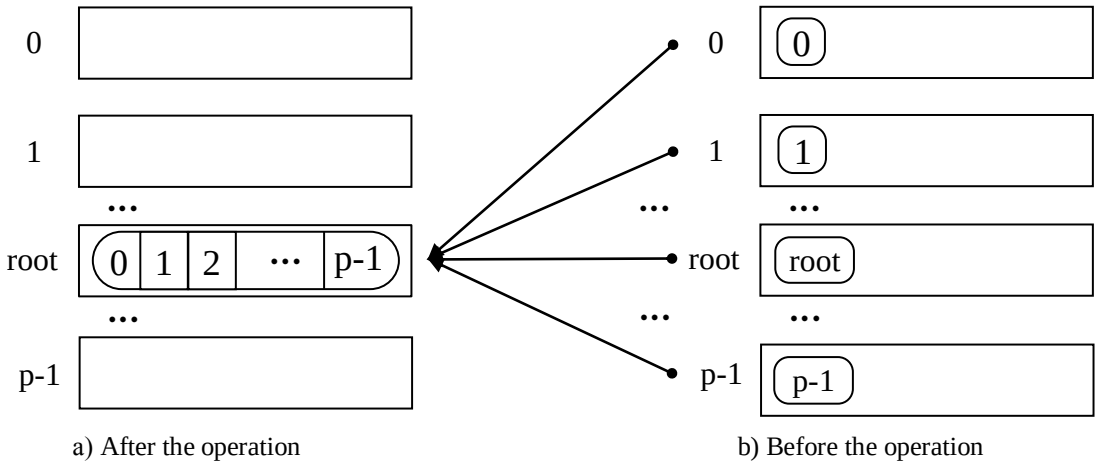
- Gathering data from all the processes to a process is reverse to data scattering. The following MPI function provides the execution of this operation

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

- **sbuf, scount, stype –** the parameters of the transmitted message
- **rbuf, rcount, rtype –** the parameters of the received message
- **root –** the rank of the process, on which the result must be obtained
- **comm –** the communicator, within of which the operation is executed

Collective Communications…
Scattering and Gathering

- The function **`MPI_Gather()`** should be called by all the processes of the communicator **`comm`**.
- The **`root`** process receives the equal sized (**`rcount`**) messages into the buffer **`rbuf`** from all the processes
- Each process (including **`root`**) transmits the message of **`scount=rcount`** length from the buffer **`sbuf`**



Collective Communications…
Example: Calculating the Inner Product

- Let's discuss the following problem

$$S = \sum_{i=1}^{n} a_i \cdot b_i$$

- To develop the parallel implementation it is necessary to
  - divide the vectors a and b into "equal" blocks
  - transmit these blocks to the processes
  - carry out the partial inner product in each process
  - reduce the values of the computed partial sums on one of the processes to obtain the general result of the problem

Collective Communications…
Example: Calculating the Inner Product

```
#include "mpi.h"
#include "stdio.h"

void main(int argc, char *argv[]) {
  double *a, *b;
  int i, n, ProcNum, ProcRank;
  double sum, sum_all;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
  if (ProcRank == 0){
    scanf("%d", &n);
    a = new double[n];
    b = new double[n];
    // initialization of vectors a and b
  }
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
  n = n / ProcNum;
```

Collective Communications…
Example: Calculating the Inner Product

```
  if (ProcRank != 0) {
    a = new double[n];
    b = new double[n];
  }
  MPI_Scatter(a, n, MPI_DOUBLE, a, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Scatter(b, n, MPI_DOUBLE, b, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  sum = sum_all = 0;
  for (i = 0; i < n; i++)
    sum += a[i] * b[i];
  MPI_Reduce(&sum, &sum_all, 1, MPI_DOUBLE, MPI_SUM, 0,
    MPI_COMM_WORLD);
  if (ProcRank == 0)
    printf("\nInner product = %10.2f", sum_all);

  MPI_Finalize();
  delete [] a;
  delete [] b;
}
```

Collective Communications…
All to All Communications

- To obtain all the gathered data on each communicator process, it is necessary to use the function of gathering and distribution **`MPI_Allgather()`**

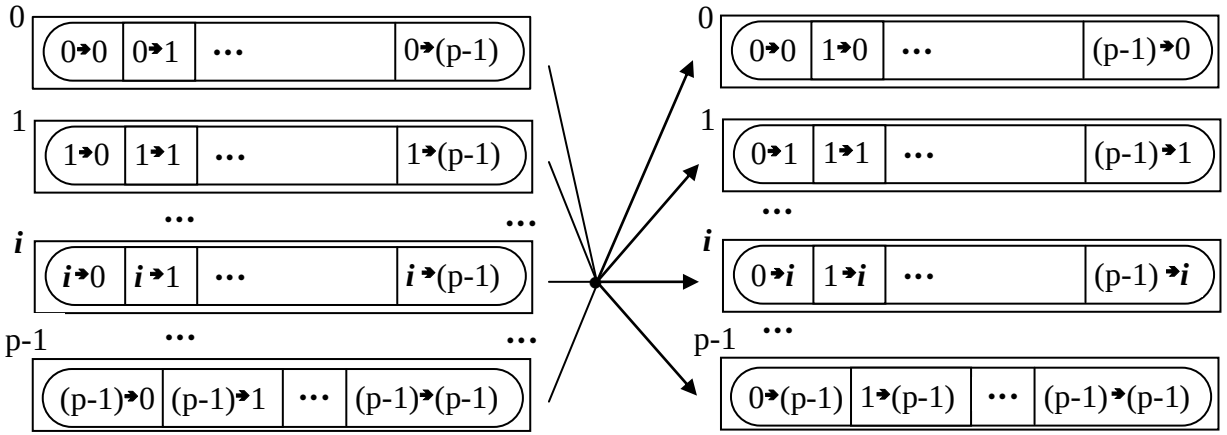
```
int MPI_Allgather(
  void *sbuf, int scount, MPI_Datatype stype,
  void *rbuf, int rcount, MPI_Datatype rtype,
  MPI_Comm comm);
```

- The execution of the general variant of data gathering operation, when the sizes of the messages transmitted among the processes may differ, is provided by means of the functions **`MPI_Gatherv()`** and **`MPI_Allgatherv()`**

Collective Communications…
All to All Communications

- The total data exchange among processes is provided by the function

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



a) Before the operation

b) After the operation

- The variant of this operation in case when the sizes of the transmitted messages may differ is provided by means of the function **`MPI_Alltoallv()`**

Collective Communications
All to All Communications

- The function **`MPI_Reduce()`** provides obtaining the results of data reduction only on one process
- To obtain the data reduction results on each of the communicator processes, it is necessary to use the function **`MPI_Allreduce()`**

```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
  MPI_Datatype type, MPI_Op Op, MPI_Comm comm);
```

Collective Communications
Computation Synchronization

- Process synchronization, i.e. simultaneous achieving the specified points of the parallel program by various processes is provided by means of the MPI function

```
int MPI_Barrier(MPI_Commcomm);
```

- The function **`MPI_Barrier()`** is collective operation
- It should be called by all the processes of the communicator comm
- 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 communicator

COMMUNICATIONS BETWEEN TWO PROCESSES

- Communication Modes
- Nonblocking Communications
- Simultaneous Sending and Receiving

Communications between Two Processes… Communication Modes

The **Standard** mode*:*

- It is provided by the function **`MPI_Send()`**
- The sending process is blocked during the time of the function execution
- The buffer may be used repeatedly after the function termination
- The state of the transmitted message may be different at the moment of the function termination, i.e. the message may be located in the sending process, may be being transmitted, may be stored in the receiving process, or may be received by the receiving process by means of the function **`MPI_Recv()`**

Communications between Two Processes… Communication Modes

The **Synchronous** mode

- The message communication function is terminated only when the process got the confirmation that the receiving process has started receiving the transmitted message

`MPI_Ssend` – the function of sending message in the **Synchronous** mode

The **Ready** mode

- May be used only if the message receiving operation has already been initiated. The message buffer may be repeatedly used after the termination of the message sending function:

`MPI_Rsend` – the function of sending message in the **Ready** mode

Communications between Two Processes… Communication Modes

The **Buffered** mode

- Assumes the use of additional buffer for copying the transmitted messages in them; the function of message sending is terminated immediately after the message has been copied in the buffer

MPI_Bsend – the function of sending message in the **Buffered** mode

- To use the buffered communication mode, the MPI memory buffer for buffering messages should be created and passed into MPI

```
int MPI_Buffer_attach(void *buf, int size)
```

- **buf** – the memory buffer for buffering messages
- **size** – buffer size

- After all the operations with the buffer are terminated, it mustbe disconnected from MPI by means of the following function

```
int MPI_Buffer_detach(void *buf, int *size);
```

Communications between Two Processes… Communication Modes

- The **Ready** mode is formally the fastest of all, but it is used quite seldom, as it is usually rather difficult to provide the readiness of the receiving
- The **Standard** and the **Buffered** modes can also be executed sufficiently fast, but may lead to sizeable recourse expenses (memory). In general, they may be recommended for transmitting short messages
- The **Synchronous** mode is the slowest of all, as it requires the confirmation of receiving. At the same time, this mode is the most reliable one. It may be recommended for transmitting long messages

Communications between Two Processes… Nonblocking Communications

- **Blocking functions** block the process execution until the called functions terminate their operations
- **Nonblocking functions** provide the possibility to execute the functions of data exchange without blocking the processes in order to carry out the message communications and the computations in parallel
- The nonblocking method
  - Is rather complicated
  - May provide significant decreasing the efficiency losses for parallel computations, which arise because of rather slow communication operations

Communications between Two Processes… Nonblocking Communications

- The names of the nonblocking functions are formed by means of adding the prefix **I** (Immediate) to the corresponding blocking function names

```
int MPI_Isend(void *buf, int count, MPI_Datatype type,
  int dest, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Issend(void *buf, int count, MPI_Datatype type,
  int dest, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Ibsend(void *buf, int count, MPI_Datatype type,
  int dest, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Irsend(void *buf, int count, MPI_Datatype type,
  int dest, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Irecv(void *buf, int count, MPI_Datatype type,
  int src, int tag, MPI_Comm comm, MPI_Request *request);
```

Communications between Two Processes… Nonblocking Communications

- The state of the executed nonblocking data communication operation may be checked by means of the following function

```
int MPI_Test( MPI_Request *request, int *flag,
  MPI_status *status);
```

- **request** – is the operation descriptor, which is defined when the nonblocking function is called
- **flag** – is the result of checking (=true, if the operation is terminated)
- **status** – the result of the function execution (only for the terminated operation)

- This function is a nonblocking one

Communications between Two Processes… Nonblocking Communications

- The following scheme of combining the computations and the execution of the nonblocking communication operation is possible

```
MPI_Irecv(buf, count, type, dest, tag, comm, &request);
...
do {
   ...
  MPI_Test(&request, &flag, &status);
} while (!flag );
```

- Blocking operation of waiting for the nonblocking operation termination

```
int MPI_Wait(MPI_Request *request, MPI_status *status);
```

Communications between Two Processes… Nonblocking Communications

- Additional checking and waiting functions for nonblockingexchange operations

MPI_Testall	–	checking the termination of all the enumerated communication operations
MPI_Waitall	–	waiting for the termination of all the enumerated communication operations
MPI_Testany	–	checking the termination of at least one of the enumerated communication operations
MPI_Waitany	–	waiting for the termination of any of the enumerated communication operations
MPI_Testsome	–	checking the termination of each enumerated communication operation
MPI_Waitsome	–	waiting for termination of at least one of the enumerated communication operations and estimating the state of all the operations

Communications between Two Processes
Simultaneous Sending and Receiving

- Efficient simultaneous execution of data sending and receiving operations may be provided by means of the following MPI function

```
int MPI_Sendrecv(void *sbuf, int scount, MPI_Datatype stype,
  int dest, int stag, void *rbuf, int rcount, MPI_Datatype rtype,
  int src, int rtag, MPI_Comm comm, MPI_Status *status),
```

- **sbuf**, **scount**, **stype**, **dest**, **stag** – the parameters of the transmitted message
- **rbuf**, **rcount**, **rtype**, **src**, **rtag** – the parameters of the received message
- **comm** – the communicator, within of which the data communication is executed
- **status** – the results of the operation execution

- In case when the messages are of the same type, MPI is able to use a single buffer

```
int MPI_Sendrecv_replace(void *buf, int count,
  MPI_Datatype type, int dest, int stag, int source,
  int rtag, MPI_Comm comm, MPI_Status *status);
```

Summary

- Collective data communication operations are considered
- The data communication between two processes are discussed
- The modes of operation execution, such as the standard, synchronous, buffered and ready ones, are described in detail
- Nonblocking data communications between the processes is discussed for every operation
- Examples of the MPI based parallel programs are presented

Exercises

- Develop a sample program for each collective operation available in MPI.
- Develop the implementations of collective operations using point-to-point communications. Carry out the computational experiments and compare the execution time of the developed programs to the functions of MPI for collective operations.
- Develop a program, carry out the experiments and compare the results for different algorithms of data gathering, processing and broadcasting (the function **`MPI_Allreduce()`**).

References

1. The internet resource, which describes the standard MPI: http://www.mpiforum.org
2. One of the most widely used MPI realizations, the library MPICH, is presented on http://www.mpich.org
3. Quinn, M.J. (2004). Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill.
4. Pacheco, P. (1996). Parallel Programming with MPI. - Morgan Kaufmann.
5. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J. (1996). MPI: The Complete Reference. – MIT Press, Boston, 1996.
6. Group, W., Lusk, E., Skjellum, A. (1999). Using MPI – 2nd Edition: Portable Parallel Programming with the Message Passing Interface (Scientific and Engineering Computation). – MIT Press.
7. Group, W., Lusk, E., Thakur, R. (1999). Using MPI-2: Advanced Features of the Message Passing Interface (Scientific and Engineering Computation). – MIT Press.