University of Nizhni Novgorod
Faculty of Computational Mathematics & Cybernetics

Introduction to Parallel Programming

Section 4. *Part 2*.

Parallel Programming with MPI…

Gergel V.P., Professor, D.Sc.,
Software Department

Contents

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 system buffers, which are used 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),
where
    - buf   -  the memory buffer for buffering messages,
    - size  –  buffer size.
```

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

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

Communications between Two Processes…

Communication Modes:

- The R*eady* 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 S*tandard* and the B*uffered* 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 S*ynchronous* 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 source, 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),
where
- 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_Isend(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 nonblocking exchange 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

The 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 source, int rtag,
  MPI_Comm comm, MPI_Status *status),
```

where

- **sbuf**, **scount**, **stype**, **dest**,   **stag** - the parameters of the transmitted message,
- **rbuf**, **rcount**, **rtype**, **source**, **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).
```

Collective Communication Operations…

MPI operations are called ***collective****,*
if all the processes of the communicator
participate in them

Collective Communication Operations…

Scattering Data from a Process to all the Processes…

- *Scattering data* – the root process transmits the equal sized messages to all the processes

```
int MPI_Scatter(void *sbuf,int scount,MPI_Datatype sto type,
                void *rbuf,int rcount,MPI_Datatype rtype,
                int root, MPI_Comm comm),
where
```

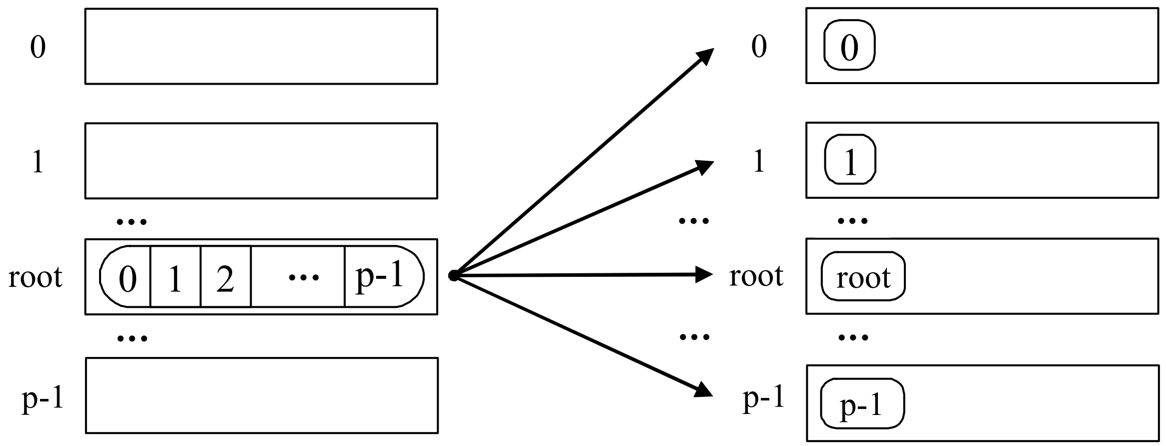
- **`sbuf`, `scount`, `stype`** - the parameters of the transmitted message (*scount* defines the number of elements transmitted to each process),
- **`rbuf`, `rcount`, `rtype`** - the parameters of the message received in the processes,
- **`root`** – the rank of the process, which performs data scattering,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

- In contrast to the broadcast operation, in this case the transmitted data to different processes may be differed

Collective Communication Operations…

Scattering Data from a Process to all the Processes

- The call of *MPI_Scatter* should be provided in each communicator process
- *MPI_Scatter* transmits messages of the same size to all the processes. When the message sizes for different processes may be different, the execution of data scattering is provided by means of the function *MPI_Scatterv*



a) Before the operation b) After the operation

Collective Communication Operations…

Gathering Data from All the Processes to a Process…

- 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),
```

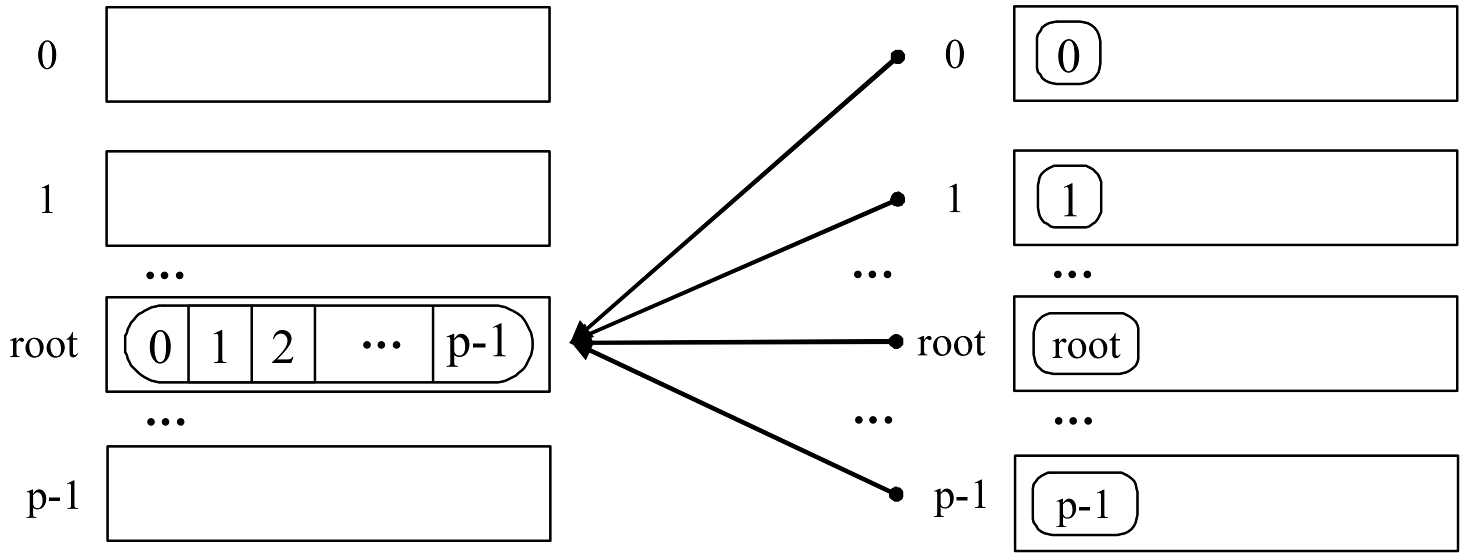
where

- **`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 communication is executed.

Collective Communication Operations…

Gathering Data from All the Processes to a Process…

- The call of the function *MPI_Garter* for gathering the data must be provided in each communicator process



a) After the operation b) Before the operation

Collective Communication Operations…

Gathering Data from All the Processes to a Process

- 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 Communication Operations…

All to All Communications...

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

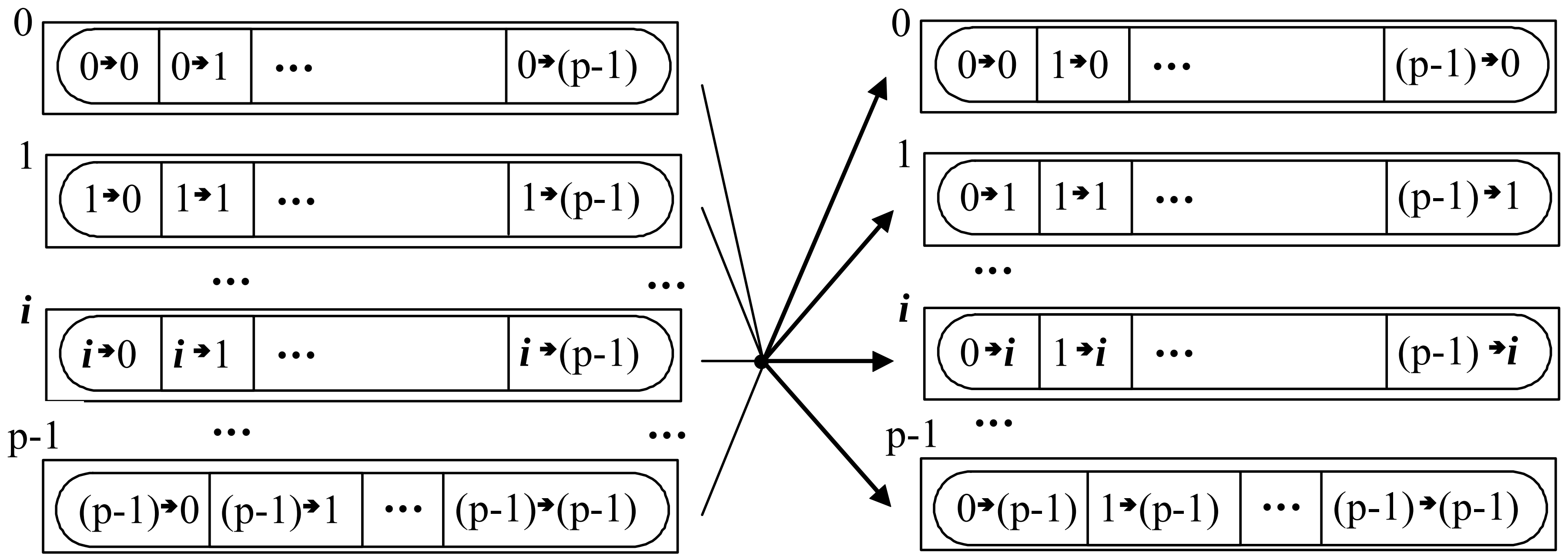
```
int MPI_Alltoall(void *sbuf,int scount,MPI_Datatype stype,
  void *rbuf,int rcount, MPI_Datatype rtype,MPI_Comm comm),
where
```

- **`sbuf`, `scount`, `stype`** - the parameters of the transmitted messages,
- **`rbuf`, `rcount`, `rtype`** - the parameters of the received messages,
- **`comm`** - the communicator, within of which the data transmission is executed

- The function MPI_Alltoall should be called in each communicator process during the execution of all to all data communication 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 Communication Operations…

All to All Communications



a) Before the operation b) After the operation

Collective Communication Operations…

Reduction Operations…

- 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);
```

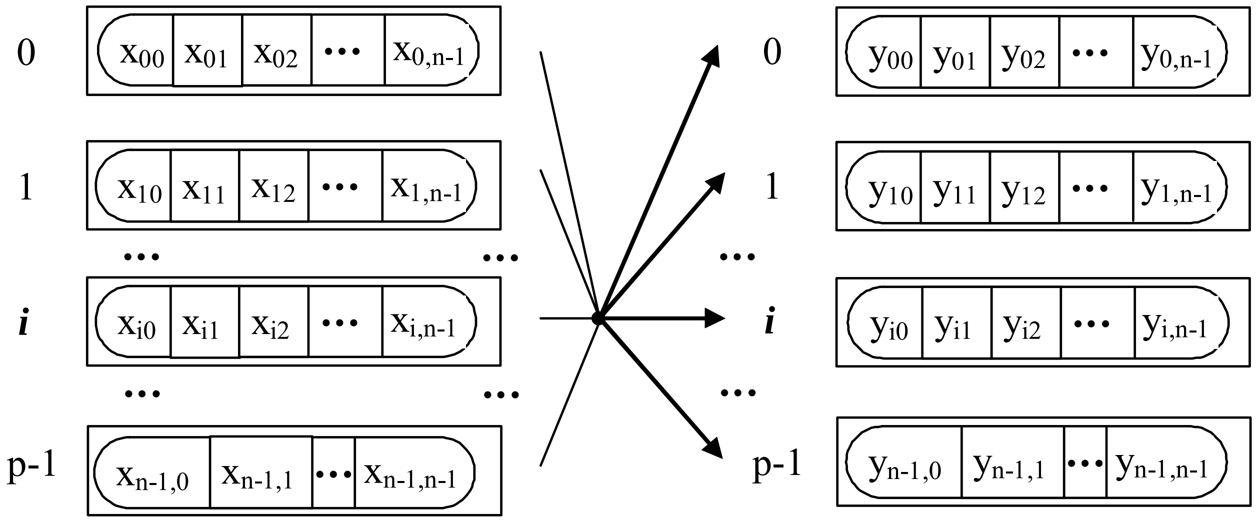
- The possibility to control the distribution of the data among the processes is provided by the function *MPI_Reduce_scatter*
- One more variant of the operation of gathering and processing data, which provides obtaining all the partial results of reduction, may be achieved by means of the following function:

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

Collective Communication Operations

Reduction Operations

- The general scheme of the function *MPI_Scan* execution is the following:



0
1
i
p-1
x_{00} x_{01} x_{02} … $x_{0,n-1}$
x_{10} x_{11} x_{12} … $x_{1,n-1}$
x_{i0} x_{i1} x_{i2} … $x_{i,n-1}$
$x_{n-1,0}$ $x_{n-1,1}$ … $x_{n-1,n-1}$
y_{00} y_{01} y_{02} … $y_{0,n-1}$
y_{10} y_{11} y_{12} … $y_{1,n-1}$
y_{i0} y_{i1} y_{i2} … $y_{i,n-1}$
$y_{n-1,0}$ $y_{n-1,1}$ … $y_{n-1,n-1}$

a) Before the operation b) After the operation

- The received message elements are the results of processing the corresponding transmitted message elements. To obtain the results at the process with the rank *i, 0≤i<p*, the data from the processes with the ranks smaller or equal to *i* are used

Derived Data Types in MPI…

- In all the above considered examples of the data communication functions it was assumed, that the messages are a certain continuous vector of the elements of the type predetermined in MPI
- The data necessary to be transmitted may not be located close to each other and contain the values of different types:
  - The data may be transmitted using several messages (this method will not be efficient because of accumulating the latencies of the number of executed data communication operations),
  - The data necessary to be transmitted can be packed into the format of a continuous vector (in that case there are some excessive operations of copying the data)

Derived Data Types in MPI…

- The *derived data type* in MPI is the description of a set of the values of the predetermined MPI types, the described values are not necessarily located continuously in the memory:
  - The type is set in MPI by means of the *type map* in the form of the sequential descriptions of values included into the type, each separate value is described by pointing to the type and the offset of the location address from a certain origin address, i.e. :

TypeMap = $\{(type_0, disp_0), (type_1, disp_1), \dots , (type_{n-1}, disp_{n-1})\}$

  - The part of the type map, which contains only the types of values, is called in MPI a type *signature*:

TypeSignature = $\{type_0, type_1, \dots , type_{n-1}\}$

Derived Data Types in MPI…

- ***Example***:
  - Let the message include the following variable values:

```
double a; /* address 24 */
double b; /* address 40 */
int    n; /* address 48 */
```

  - Then the derived type for the description of the data should have the map of the following form:

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

Derived Data Types in MPI…

- The following number of new concepts is used in MPI for the derived data types:
  - The *lower boundary* of type:

$$lb\ (TypeMap) = \min_j (disp_j),$$

  - The *upper boundary* of type:

$$ub\ (TypeMap) = \max_j (disp_j + sizeof(type_j)) + \Delta,$$

  - The *extent* of type (the extent is the memory size in bytes, which should be allocated for a derived type element):

$$extent\ (TypeMap) = ub\ (TypeMap) - lb\ (TypeMap),$$

  - The *size* of the data type is the number of bytes that is required to place a single value of this data type.

The difference between the values of the extent and the size is in the approximation value needed for the address alignment

Derived Data Types in MPI…

- MPI provides the following functions for obtaining the values of the extent and the type size:

```
int MPI_Type_extent ( MPI_Datatype type, MPI_Aint *extent );
int MPI_Type_size   ( MPI_Datatype type, MPI_Aint *size );
```

- The lower and the upper boundaries of the types may be determined by means of the following functions:

```
int MPI_Type_lb ( MPI_Datatype type, MPI_Aint *disp );
int MPI_Type_ub ( MPI_Datatype type, MPI_Aint *disp );
```

- The function of getting the address of the variable is essential in constructing the derived types:

```
int MPI_Address ( void *location, MPI_Aint *address );
```

Derived Data Types in MPI…

- **The Methods of Constructing the Derived Data Types:**
  - The ***continuous*** method makes possible to define a continuous set of the elements of some data type as a new derived type,
  - The ***vector* method** provides creating a new derived type as a set of elements of some available type. Between the elements there may be regular memory intervals. The size of the intervals is determined in the number of the elements of the initial type, while in case of the *h-vector* method this size has to be set in bytes,
  - The ***index* method** differs from the vector method as the intervals between the elements of the type are irregular,
  - The ***structural* method** provides the most general description of the derived type by pointing directly to the type map of the created data type

Derived Data Types in MPI…

- **The Continuous Method:**

```
int MPI_Type_contiguous(int count,MPI_Data_type oldtype,
                      MPI_Datatype *newtype);
```

– As it follows from the description, the new type *newtype* is composed as the *count* elements of the initial type *oldtype*. For instance, if the type map of the data type *oldtype* has the form

```
  { (MPI_INT,0),(MPI_DOUBLE,8) },
```

then the call of the function *MPI_Type_contiguous* with the following parameters

```
MPI_Type_contiguous (2, oldtype, &newtype);
```

will cause the creation of the new data type, the type map of which looks as follows:

```
{ (MPI_INT,0),(MPI_DOUBLE,8),(MPI_INT,16),(MPI_DOUBLE,24) }.
```

Derived Data Types in MPI…

- **The Vector Method…**
  - The new derived type is constructed in case of the vector method as a number of blocks of the initial type elements. The blocks are separated by the regular interval:

```
int MPI_Type_vector ( int count, int blocklen, int stride,
  MPI_Data_type oldtype, MPI_Datatype *newtype ),
where
- count     –  the number of blocks,
- blocklen  –  the size of each block,
- stride    –  the number of elements, located between
               the two neighboring blocks,
- oldtype   -  the initial data type,
- newtype   -  the new determined data type.
```

Derived Data Types in MPI…

- **The Vector Method…**
  - If the interval size are determined in bytes instead of the initial type elements, to construct the derived data type one can use the following function:

```
int MPI_Type_hvector ( int count, int blocklen,
 MPI_Aint stride, MPI_Data_type oldtype, MPI_Datatype *newtype );
```

Derived Data Types in MPI…

- **The Vector Method**
  – The derived data types for the description of the subarrays of the multidimensional arrays can also be created by the function:

```
int MPI_Type_create_subarray ( int ndims, int *sizes,
      int *subsizes, int *starts, int order,
      MPI_Data_type oldtype, MPI_Datatype *newtype ),
where
```

- **ndims**	–	the array dimension,
- **sizes**	–	the number of elements in each dimension of the initial array,
- **subsizes**	–	the number of elements in each dimension of the determined subarray,
- **starts**	–	the indices of the initial elements in each dimension of the determined subarray,
- **order**	-	the parameter for pointing to the necessity of re-ordering,
- **oldtype**	-	the data type of the initial array elements,
- **newtype**	-	the new data type for the description of the subarray.

Derived Data Types in MPI…

- **The Index Method…**
  – The new determined data type is created as a set of blocks of different sizes of the initial type elements. The memory locations of the blocks are set by the offset with respect to the origin of the type:

```
int MPI_Type_indexed ( int count, int blocklens[],
 int indices[], MPI_Data_type oldtype, MPI_Datatype *newtype ),
where
- count     – the number of blocks,
- blocklens – the number of elements in each block,
- indices   – the offset of each block with respect to the origin of the type
              (in number of the initial type elements),
- oldtype   - the initial data type,
- newtype   - the new determined data type.
```

Derived Data Types in MPI…

- **The Index Method**…
  - If the block offsets are defined in bytes instead of the initial type elements, to construct the derived data type one can use the following function:

```
int MPI_Type_hindexed ( int count, int blocklens[],
  MPI_Aint indices[], MPI_Data_type oldtype,
  MPI_Datatype *newtype);
```

Derived Data Types in MPI…

- **The Index Method:**
  - ***Example:*** Constructing a type for the description of the upper triangle matrix of *nxn size*:

```
// constructing a type for the description of the upper triangle matrix
for ( i=0, i<n; i++ ) {
  blocklens[i] = n - i;
  indices[i]   = i * n + i;
}
MPI_Type_indexed ( n, blocklens, indices, &UTMatrixType,
                 &ElemType );
```

Derived Data Types in MPI…

- **The Structural Method:**
  - This method is the most general constructing method for creating the derived data type, when the corresponding type map is set explicitly:

```
int MPI_Type_struct ( int count, int blocklens[],
  MPI_Aint indices[], MPI_Data_type oldtypes[],
  MPI_Datatype *newtype ),
where
 - count       –  the number of blocks,
 - blocklens   –  the number of elements in each blocks,
 - indices     –  the offset of each block with respect to the origin of
                  the type in bytes,
 - oldtypes    -  the initial data types for each  block separately,
 - newtype     -  the new determined data type.
```

Derived Data Types in MPI…

- **Derived Data Type Declaring and Deleting:**
  - The created data type should be *commited* before being used by means of the following function:

```
int MPI_Type_commit ( MPI_Datatype *type );
```

  - After the termination of its use, the derived type must be *annulled* by means of the following function:

```
int MPI_Type_free ( MPI_Datatype *type );
```

Derived Data Types in MPI…

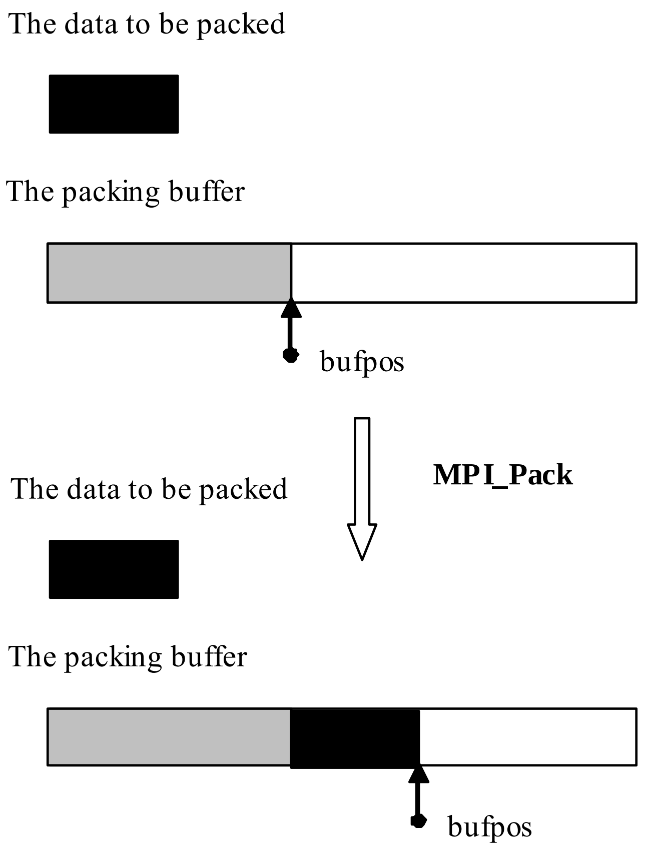
- **Forming Messages by Means of Data Packing and Unpacking…**
  - An explicit method of assembling and disassembling the messages, which contain values of different types and are located in different memory locations:

```
int MPI_Pack ( void *data, int count, MPI_Datatype type,
  void *buf, int bufsize, int *bufpos, MPI_Comm comm),
where
```

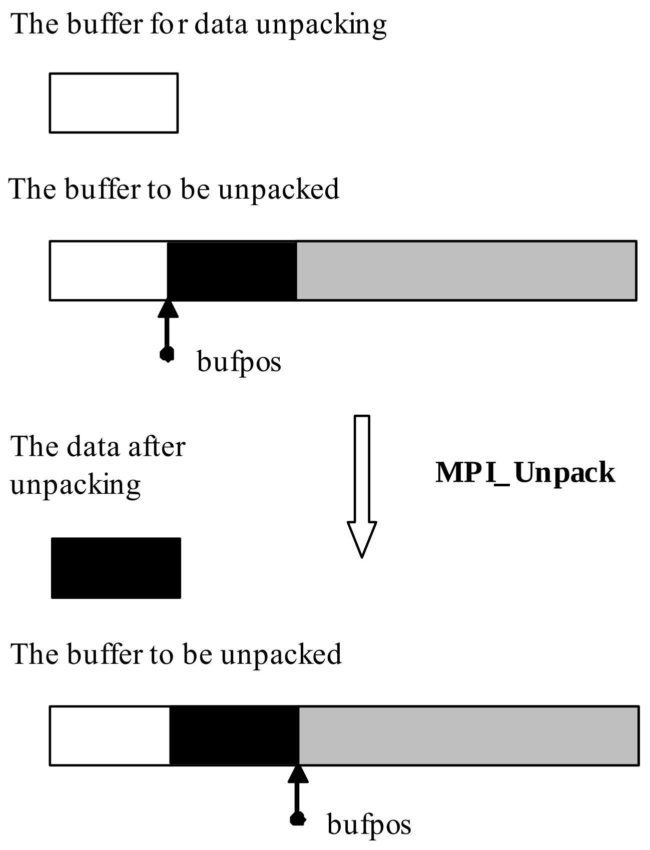
-	**data**	– the memory buffer with the elements to be packed,
-	**count**	– the number of elements in the buffer,
-	**type**	– the data type for the elements to be packed,
-	**buf**	- the memory buffer for packing,
-	**buflen**	– the buffer size in bytes,
-	**bufpos**	– the position for the beginning of buffering (in bytes from the origin address of the buffer),
-	**comm**	- the communicator for the packed message.

Derived Data Types in MPI…

❑ The Scheme of Data Packing and Unpacking…



a) data packing


The buffer for data unpacking
The buffer to be unpacked
bufpos
The data after unpacking
MPI_Unpack
The buffer to be unpacked
bufpos

b) data unpacking

Derived Data Types in MPI…

- **Forming Messages by Means of Data Packing and Unpacking…**
  – To determine the buffer size necessary for packing, it is possible to use the following function:

```
MPI_Pack_size (int count, MPI_Datatype type,
               MPI_Comm comm, int *size);
```

  – To send the packed data the prepared buffer must be used in the function *MPI_Send* with the type *MPI_PACKED*,
  – After the receiving the message with the type *MPI_PACKED,* the data may be unpacked by means of the following function:

```
int MPI_Unpack (void *buf, int bufsize, int *bufpos,
 void *data, int count, MPI_Datatype type, MPI_Comm comm);
```

Derived Data Types in MPI…

- **Forming Messages by Means of Data Packing and Unpacking…**
  - The function *MPI_Pack* is called sequentially for packing all the necessary data. Thus, if message is a set of variables ***a***, ***b*** and ***n***

```
double a /* адрес 24 */; double b /* адрес 40 */; int    n /* адрес 48 */;
```

it is necessary to carry out the following operations in order to pack the data:

```
bufpos = 0;
MPI_Pack(a,1,MPI_DOUBLE,buf,buflen,&bufpos,comm);
MPI_Pack(b,1,MPI_DOUBLE,buf,buflen,&bufpos,comm);
MPI_Pack(n,1,MPI_INT,buf,buflen,&bufpos,comm);
```

Derived Data Types in MPI…

- **Forming Messages by Means of Data Packing and Unpacking…**
  – To unpack the data It is necessary to carry out the following:

```
bufpos = 0;
MPI_Unpack(buf,buflen,&bufpos,a,1,MPI_DOUBLE,comm);
MPI_Unpack(buf,buflen,&bufpos,b,1,MPI_DOUBLE,comm);
MPI_Unpack(buf,buflen,&bufpos,n,1,MPI_INT,comm);
```

Derived Data Types in MPI

- **Forming Messages by Means of Data Packing and Unpacking:**
  - This approach causes the additional operations of packing and unpacking the data,
  - This method may be justified, if the message sizes are comparatively small and the message is packed/unpacked sufficiently rarely,
  - Packing and unpacking may prove to be useful, if buffers are explicitly used for the buffered data communication method

Summary

- 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
- Non-blocking data communications between the processes is discussed for every operation
- Collective data communication operations are considered
- The use of derived data types in MPI is discussed

Discussions

- Adequacy of the supported data communication operations in MPI
- Recommendations of usage for different methods of constricting the derived data types

Exercises…

- **Data Transmission between Two Processes:**
  1. Derive new variants of the previously developed programs with different modes of data communications. Compare the execution time of data communication operations in cases of different modes.
  2. Derive new variants of the previously developed programs using the non-blocking method of data communication operations. Estimate the necessary amount of the computational operations, which is needed to execute data communication and computations in parallel. Develop the program, which has no computation delays caused by waiting for the transmitted data.
  3. Develop a program, where two processes repeatedly exchange messages of *N* byte length and use the operation of simultaneous data sending and receiving. Compare the results of the computational experiments.

Exercises…

- **Collective Data Transmission Operations:**
  4. Develop a sample program for each collective operation available in MPI.
  5. 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.
  6. Develop a program, carry out the experiments and compare the results for different algorithms of data gathering, processing and broadcasting (the function *MPI_Allreduce*).

Exercises

- **The Derived Data Types in MPI…**

7. Develop a sample program for each method of constructing the derived data types available in MPI.
8. Develop a sample program using data packing and unpacking functions. Carry out the experiments and compare the results to the results obtained in case of the use of the derived data types.
9. Develop the derived data types for the rows, columns and diagonals of matrices.

References…

- 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

References…

- **Group, W., Lusk, E., Skjellum, A.** (1994). Using MPI. Portable Parallel Programming with the Message-Passing Interface. –MIT Press.
- **Group, W., Lusk, E., Skjellum, A.** (1999a). Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface (Scientific and Engineering Computation). - MIT Press.
- **Group, W., Lusk, E., Thakur, R.** (1999b). Using MPI-2: Advanced Features of the Message Passing Interface (Scientific and Engineering Computation). - MIT Press.

References

- **Pacheco, P.** (1996). Parallel Programming with MPI. - Morgan Kaufmann.
- **Quinn, M. J.** (2004). Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill.
- **Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.** (1996). MPI: The Complete Reference. - MIT Press, Boston, 1996.

Next Section

- Parallel Programming with MPI…

Author’s Team

Gergel V.P., Professor, Doctor of Science in Engineering, Course Author

Grishagin V.A., Associate Professor, Candidate of Science in Mathematics

Abrosimova O.N., Assistant Professor (chapter 10)

Kurylev A.L., Assistant Professor (learning labs 4,5)

Labutin D.Y., Assistant Professor (ParaLab system)

Sysoev A.V., Assistant Professor (chapter 1)

Gergel A.V., Post-Graduate Student (chapter 12, learning lab 6)

Labutina A.A., Post-Graduate Student (chapters 7,8,9, learning labs 1,2,3, ParaLab system)

Senin A.V., Post-Graduate Student (chapter 11, learning labs on Microsoft Compute Cluster)

Liverko S.V., Student (ParaLab system)

About the project

The purpose of the project is to develop the set of educational materials for the teaching course “Multiprocessor computational systems and parallel programming”. This course is designed for the consideration of the parallel computation problems, which are stipulated in the recommendations of IEEE-CS and ACM Computing Curricula 2001. The educational materials can be used for teaching/training specialists in the fields of informatics, computer engineering and information technologies. The curriculum consists of **the training course “Introduction to the methods of parallel programming”** and **the computer laboratory training “The methods and technologies of parallel program development”**. Such educational materials makes possible to seamlessly combine both the fundamental education in computer science and the practical training in the methods of developing the software for solving complicated time-consuming computational problems using the high performance computational systems.

The project was carried out in Nizhny Novgorod State University, the Software Department of the Computing Mathematics and Cybernetics Faculty (http://www.software.unn.ac.ru). The project was implemented with the support of Microsoft Corporation.