


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

03 Lecture
Derived Data Types, Communicators and Virtual Topologies

With the support of Microsoft

Sysoyev A.V.
Software department

Contents

- Derived Data Types in MPI
  - Type Map
  - The Methods of Constructing
  - Declaring and Deleting
  - Data Packing
- Groups of Processes and Communicators
  - Managing Groups
  - Managing Communicators
- Virtual Topologies
  - Cartesian Topologies (Grids)
  - Graph Topologies

DERIVED DATA TYPES IN MPI

- Type Map
- The Methods of Constructing
- Declaring and Deleting
- Data Packing

Derived Data Types in MPI…
Type Map

- In all the above considered examples 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 may 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 Type in MPI may be created to describe the placement of data in memory

Derived Data Types in MPI…
Type Map

- 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

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…
Type Map

Example

- Let the message include the following variable values

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

- Suppose we know the addresses of variables a, b, n in memory
- 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…
Type Map

- The following 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…
Type Map

- MPI provides the following functions for obtaining the values ofthe 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 **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 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_Datatype oldtype, MPI_Datatype *newtype);
```

- **count** – the number of blocks
- **blocklen** – the number of elements in each block
- **stride** – the number of elements between start of 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_Datatype oldtype,
  MPI_Datatype *newtype);
```

Derived Data Types in MPI…
The Vector Method

- The derived data types for the description of the subarray 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_Datatype oldtype, MPI_Datatype *newtype);
```

- **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 storage order for the subarray as well as the full array (**MPI_ORDER_C**, **MPI_ORDER_FORTRAN**)
- **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 offsets[], MPI_Datatype oldtype,
  MPI_Datatype *newtype);
```

- **count** – the number of blocks
- **blocklen** – the number of elements in each block
- **offsets** – the offset of each block from the start 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_indexed(int count, int blocklens[],
  MPI_Aint offsets[], MPI_Datatype 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 *n x n* size

```
int *blocklens, *offsets;
MPI_Datatype UTMatrixType;

// memory allocation for blocklens and offsets
for (i = 0, i < n; i++)
{
  blocklens[i] = n - i;
  offsets[i]   = i * n + i;
}
MPI_Type_indexed(n, blocklens, offsets, MPI_DOUBLE,
  &UTMatrixType);
```

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[],
  int offsets[], MPI_Datatype oldtypes[],
  MPI_Datatype *newtype);
```

- **count** – the number of blocks
- **blocklen** – the number of elements in each block
- **offsets** – the offset of each block from the start of the type (in number of the initial type elements)
- **oldtype** – the initial data type for each block separately
- **newtype** – the new determined data type

Derived Data Types in MPI…
Declaring and Deleting

- The created data type should be committed 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…
Data Packing

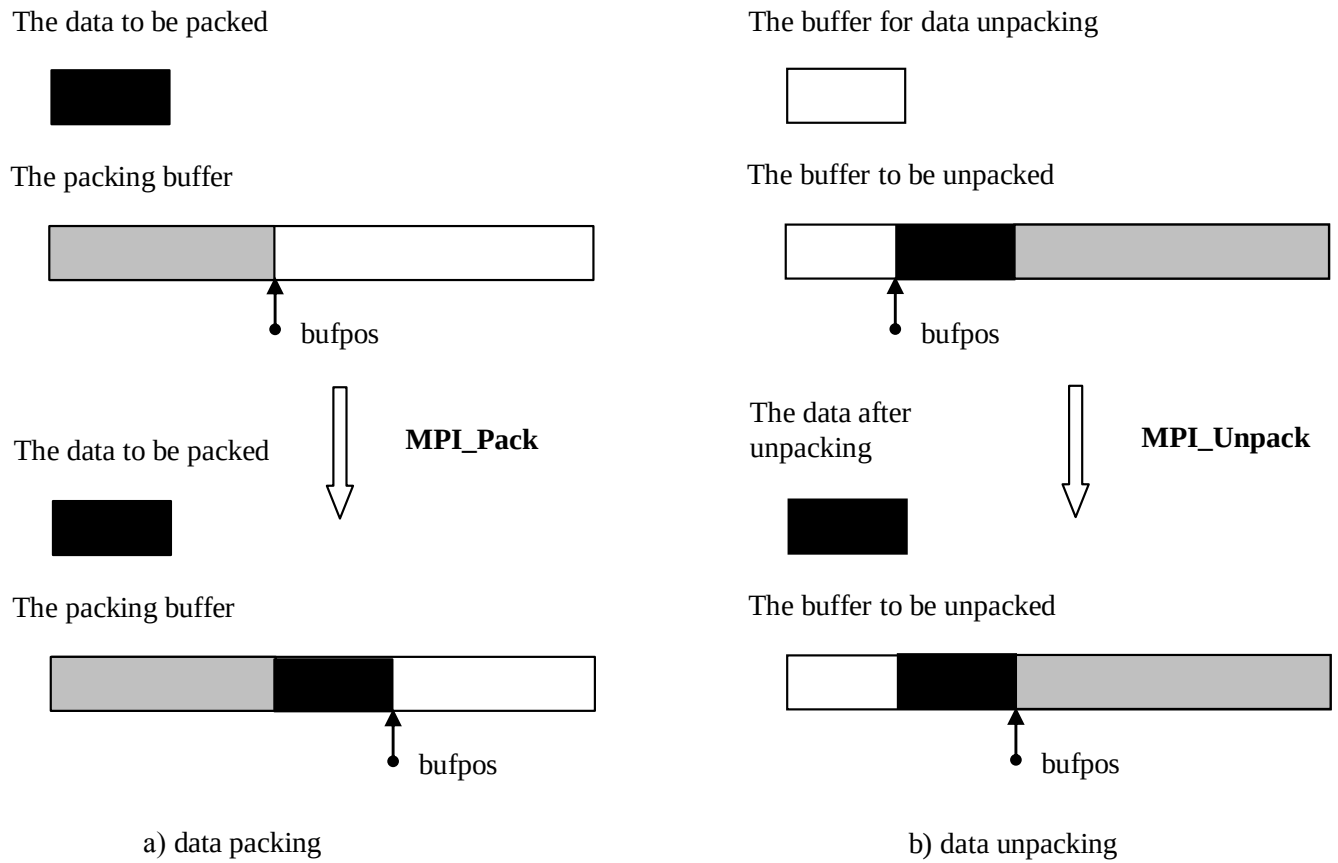
- An explicit method of assembling and disassembling the messages, which contain values of different types and are located in different memory locations

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

- **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…
Data Packing

- The Scheme of Data Packing and Unpacking



a) data packing

b) data unpacking

Derived Data Types in MPI…
Data Packing

- 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…
Data Packing

- 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; /* address 24 */
double b; /* address 40 */
int    n; /* address 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…
Data Packing

- 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
Data Packing

- 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

GROUPS OF PROCESSES AND COMMUNICATORS

Managing groups

Managing communicators

Groups of Processes and Communicators… Managing Groups

- Processes are united into groups. The group may contain all the processes of a parallel program or a part of the available processes only. The same process may belong to several groups
- The groups of processes are formed in order to create communicators on their basis
- The groups of processes may be defined on the basis of the available groups only. The group associated with some communicator may be excluding by means of the following function

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group);
```

Groups of Processes and Communicators…
Managing Groups

- New groups may be created on the basis of the existing groups
- It is possible to create a new group **`newgroup`** on the basis of the group **`oldgroup`**, which includes **`n`** processes
  - The ranks of the processes **to be included** in **`newgroup`** are enumerated in the array **`ranks`**

```
int MPI_Group_incl(MPI_Group oldgroup, int n, int ranks[],
  MPI_Group *newgroup);
```

  - The ranks of the processes that **have not to be included** in **`newgroup`** are enumerated in the array **`ranks`**

```
int MPI_Group_excl(MPI_Group oldgroup, int n, int ranks[],
  MPI_Group *newgroup);
```

Groups of Processes and Communicators… Managing Groups

- New groups may also be created by the following operations:
  - Creating a new group **newgroup** by **uniting** the groups **group1** and **group2**

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2,
  MPI_Group *newgroup);
```

  - Creating a new group **newgroup** from the **common** processes of the groups **group1** and **group2**

```
int MPI_Group_intersection(MPI_Group group1,
  MPI_Group group2, MPI_Group *newgroup);
```

  - Creating a new group **newgroup** by the **difference** of the groups **group1** and **group2**

```
int MPI_Group_difference(MPI_Group group1,
  MPI_Group group2, MPI_Group *newgroup);
```

Groups of Processes and Communicators… Managing Groups

- The following MPI functions provide obtaining information of the group of processes
  - Obtaining the number of processes in the group

```
int MPI_Group_size(MPI_Group group, int *size);
```

  - Obtaining the rank of the current process in the group

```
int MPI_Group_rank(MPI_Group group, int *rank);
```

- After the termination of its use, the group must be deleted

```
int MPI_Group_free(MPI_Group *group);
```

Groups of Processes and Communicators…
Managing Communicators

- A **communicator** in MPI is a specially designed control object, which unites in its contents a group of processes and a number of additional parameters (context), which are used in data communication operations
- This subsection discusses managing the **intracommunicators**, which are used for data communication operation within a group of processes

Groups of Processes and Communicators… Managing Communicators

- To create new communicators the two main methods are used
  - The duplication of the available communicator

```
int MPI_Comm_dup(MPI_Comm oldcom, MPI_comm *newcomm);
```

  - The creation of a new communicator from the subset of the processes of the available communicator

```
int MPI_comm_create(MPI_Comm oldcom, MPI_Group group,
  MPI_Comm *newcomm);
```

- The operation of creating communicators is collective and must be executed by all the initial communicator processes
- After the termination of its use, the communicator should be deleted

```
int MPI_Comm_free(MPI_Comm *comm);
```

Groups of Processes and Communicators…
Managing Communicators

- The following function provides a fast and useful method of simultaneous creation of several communicators

```
int MPI_Comm_split(MPI_Comm oldcomm, int split, int key,
  MPI_Comm *newcomm)
```

- **oldcomm** – the initial communicator,
- **split** – the number of the communicator, to which the process should belong
- **key** – the rank order of the process in the communicator being created
- **newcomm** – the communicator being created

- The function **MPI_Comm_split()** should be called in each process of the communicator **oldcomm**

Groups of Processes and Communicators
Managing Communicators

- The execution of the function **`MPI_Comm_split()`** leads to separating the processes into disjoint groups
- Each new group is formed from processes which have the same values of the parameter **`split`**
- On the basis of the created groups a set of communicators is created
- The order of enumeration for the process ranks is selected in such a way that it corresponds to the order of the values **`key`** (the process with the greater value **`key`** should have a higher rank)

VIRTUAL TOPOLOGIES

Cartesian Topologies (Grids)

Graph Topologies

Virtual Topologies…

- The **topology** of a computer system is the structure of the network nodes and communication links, which connect them. The topology may be presented as a graph, where the vertices are the system processors (processes), and the arcs correspond to the available communication links (channels)
- Point-to-point data communication operations may be executed for any processes of the same communicator. All the processes of the communicator participate in collective operations. In this respect, the **logical topology** of the communication links in a parallel program is a **complete graph**
- We may organize the logical presentation of any necessary virtual topology. For this purpose it is sufficient to form additional process addressing

Virtual Topologies…
Cartesian Topologies (Grids)

- **Cartesian** topologies assume the presentation of a set of processes as a rectangular grid and the use of Cartesian coordinate system for pointing to the processes
- The following function is used for creating the Cartesian topology

```
int MPI_Cart_create(MPI_Comm oldcomm, int ndims, int *dims,
  int *periods, int reorder, MPI_Comm *cartcomm);
```

- **oldcomm** – the initial communicator
- **ndims** – the Cartesian grid dimension
- **dims** – the array of **ndims** length, it defines the number of processes in each dimension of the grid
- **periods** – the array of **ndims** length, which defines whether the grid is periodical along each dimension
- **reorder** – the parameter for pointing out if the process ranks can be reordered
- **cartcomm** – the communicator being created with the Cartesian process topology

Virtual Topologies…
Cartesian Topologies (Grids)

- In order to determine the Cartesian process coordinates according to its rank, the following function can be used

```
int MPI_Card_coords(MPI_Comm comm, int rank, int ndims,
  int *coords);
```

- **comm** – the communicator with grid topology
- **rank** – the rank of the process, for which Cartesian coordinates are determined
- **ndims** – the Cartesian grid dimension
- **coords** – the Cartesian process coordinates calculated by the function

Virtual Topologies…
Cartesian Topologies (Grids)

- The reverse operation, i.e. determining the process rank according to its Cartesian coordinates, is provided by means of the following function

```
int MPI_Cart_rank(MPI_Comm comm, int coords[], int *rank);
```

- **comm** – the communicator with grid topology
- **coords** – the Cartesian coordinates of the process
- **rank** – the process rank calculated by the function

Virtual Topologies…
Cartesian Topologies (Grids)

- The procedure of splitting the grids into subgrids of smaller dimension, which is useful in many applications, is provided by the following function

```
int MPI_Card_sub(MPI_Comm comm, int subdims[],
  MPI_Comm *newcomm);
```

- **comm** – the initial communicator with grid topology
- **subdims** – the array for pointing the subgrid coordinates that can vary
- **newcomm** – the created communicator with the subgrid

- The function **MPI_Cart_sub()** defines, while it is being carried out, the communicators for each combination of the coordinates of the fixed dimensions of the initial grid

Virtual Topologies…
Cartesian Topologies (Grids)

- The additional function **`MPI_Cart_shift()`** provides the support of shift communications along a grid dimension
  - The **cyclic shift** on *k* elements along the grid dimension. The data from the process *i* is transmitted to the process (*i*+*k*) mod *n*, where *n* is the size of the dimension, along which the shift is performed
  - The **linear shift** on *k* positions along the grid dimension. In this variant of the operation the data from the processor *i* is transmitted to the processor *i*+*k* (if the latter is available)
- The function **`MPI_Cart_shift()`** only determines the rank of the processes, which are to exchange data in the course of shift operation. The execution of data transmission may be carried out, for instance, by means of the function **`MPI_Sendrecv()`**

Virtual Topologies…
Cartesian Topologies (Grids)

- The function **`MPI_Cart_shift()`** provides obtaining the ranks of the processes, which are to exchange the data with the current process (the process, which has called up the function **`MPI_Cart_shift()`**):

```
int MPI_Card_shift(MPI_Comm comm, int dir, int disp,
  int *source, int *dst);
```

- **`comm`** – the communicator with grid topology
- **`dir`** – the number of the dimension, along which the shift is performed
- **`disp`** – the shift value (<0 – the shift towards the beginning of the dimension)
- **`source`** – the rank of the process, from which the data should be obtained
- **`dst`** – the rank of the process, to which the data should be sent

Virtual Topologies…
Graph Topology

- To create a communicator with the graph topology the following function is intended in MPI

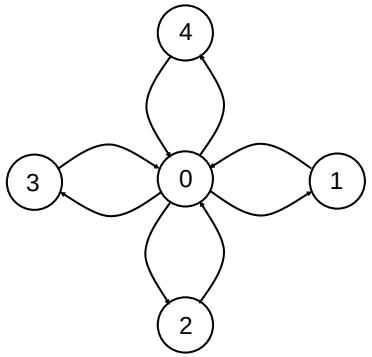
```
int MPI_Graph_create(MPI_Comm oldcomm, int nnodes,
  int index[], int edges[], int reorder,
  MPI_Comm *graphcomm);
```

- **oldcomm** – the initial communicator
- **nnodes** – the number of the graph vertices
- **index** – the number of the arcs proceeding from each vertex
- **edges** – the sequential list of the graph arcs
- **reorder** – the flag for pointing out if the process ranks can be reordered
- **graphcomm** – the created communicator with the graph type topology

Virtual Topologies…
Graph Topology

Example

- The number of processes is equal to 5, the graph vertices orders are (4,1,1,1,1), and the incidence matrix looks as follows



Processes	Communication Lines
0	1,2,3,4
1	0
2	0
3	0
4	0

- To create the topology with the graph of this type, it is necessary to perform the following program code

```
int index[] = { 4,1,1,1,1 };
int edges[] = { 1,2,3,4,0,0,0,0 };
MPI_Comm StarComm;
MPI_Graph_create(MPI_COMM_WORLD,5,index,edges,1,&StarComm);
```

Virtual Topologies…
Graph Topology

- The number of the neighboring processes, which contain the outgoing arcs from the current process, may be obtained by the following function

```
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank,
  int *nneighbors);
```

- Obtaining the ranks of the neighboring vertices is provided by the following function

```
int MPI_Graph_neighbors(MPI_Comm comm, int rank,
  int mneighbors, int *neighbors);
```

- (where **mneighbors** is the size of the array **neighbors**)

Summary

- The use of derived data types in MPI is discussed
- Group and communicator management are considered
- Virtual topologies are overviewed

Exercises

- Develop a sample program for each method of constructing the derived data types available in MPI
- 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
- Develop the derived data types for the rows, columns and diagonals of matrices
- Develop a sample program for the Cartesian topology
- Develop a sample program for a graph topology
- Develop subprograms for creating a set of additional virtual topologies (a star, a tree, etc.)

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.