


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

04 Lecture
MPI Extensions

With the support of Microsoft

Sysoyev A.V.
Software department

Contents

NONBLOCKING COLLECTIVE OPERATIONS

- Data Broadcasting
- Reduction Operations
- Scattering and Gathering
- All to All Communications
- Computation Synchronization

Nonblocking Collective Communications… General Description

- Nonblocking collective operations combine the potential benefits of nonblocking point-to-point operations with the optimized implementation and message scheduling provided by collective operations
- One way of doing this would be to perform a blocking collective operation in a separate thread
- Nonblocking collective communication often leads to better performance (avoids context switching, scheduler overheads, and thread management)
- Similarly to the blocking case, nonblocking collective operations are considered to be complete when the local part of the operation is finished

Nonblocking Collective Communications… General Description

- Completion does not indicate that other processes have completed or even started the operation (unless otherwise implied by the description of the operation)
- Completion of a particular nonblocking collective operation also does not indicate completion of any other posted nonblocking collective (or send-receive) operations, whether they are posted before or after the completed operation
- Users should be aware that MPI implementations are allowed, but not required (with exception of MPI_IBARRIER), to synchronize processes during the completion of a nonblocking collective operation

Nonblocking Collective Communications… General Description

- Unlike point-to-point operations, nonblocking collective operations do not match with blocking collective operations
- All processes must call collective operations (blocking and nonblocking) in the same order per communicator
- Once a process calls a collective operation, all other processes in the communicator must eventually call the same collective operation, and no other collective operation with the same communicator in between

Nonblocking Collective Communications…
Data Broadcasting

```
int MPI_Ibcast(void *buf, int count, MPI_Datatype type,
  int root, MPI_Comm comm, MPI_Request *request);
```

- **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
- **request** – the operation descriptor

- The function **MPI_Ibcast()** 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**

Nonblocking Collective Communications…
Data Reduction

- To “reduce” some data from all processes to chosen one

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

- **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
- **request** – the operation descriptor

Nonblocking Collective Communications… Scattering and Gathering

- To distribute some data from chosen process to all the processes

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

- **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
- **request –** the operation descriptor

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

Nonblocking Collective Communications… Scattering and Gathering

- Gathering data from all the processes to a process is reverse to data scattering

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

- **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
- **request –** the operation descriptor

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

Nonblocking 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_Iallgather()**

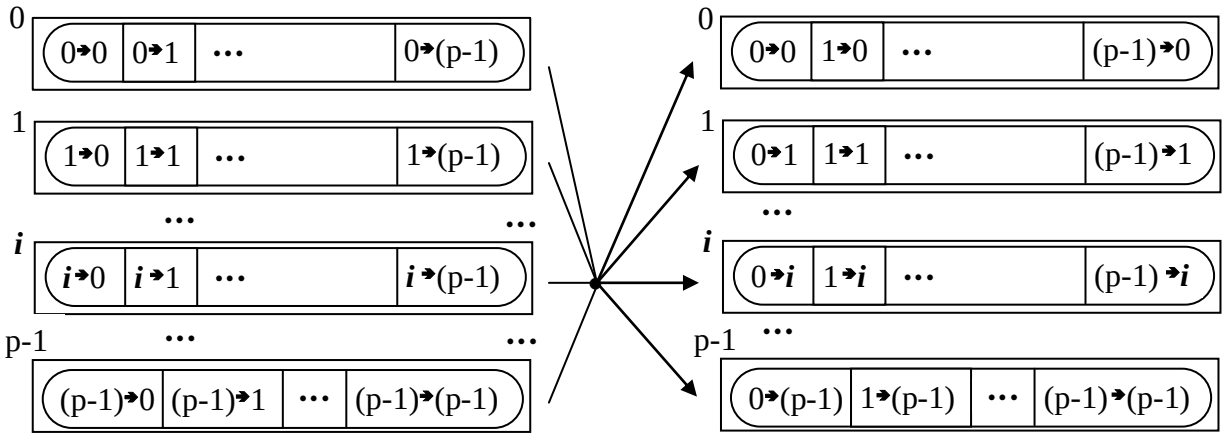
```
int MPI_Iallgather(
  void *sbuf, int scount, MPI_Datatype stype,
  void *rbuf, int rcount, MPI_Datatype rtype,
  MPI_Comm comm, MPI_Request *request);
```

- 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 function **MPI_Iallgatherv()**

Nonblocking Collective Communications… All to All Communications

- The total data exchange among processes

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



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_Ialltoallv()`**

Nonblocking Collective Communications…
All to All Communications

- To obtain the data reduction results on each of the communicator processes, it is necessary to use the function **`MPI_Iallreduce()`**

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

- For operations which are not truly associative, the result delivered upon completion of the nonblocking reduction (via **`MPI_Ireduce()`** or **`MPI_Iallreduce()`**) **may not exactly equal** the result delivered by the blocking reduction, even when specifying the same arguments in the same order

Nonblocking Collective Communications
Computation Synchronization

- **`MPI_IBarrier()`** is a nonblocking version of **`MPI_Barrier()`**

```
int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request);
```

- By calling **`MPI_IBarrier()`**, a process notifies that it has reached the barrier
- The call returns immediately, independent of whether other processes have called **`MPI_IBarrier()`**
- A nonblocking barrier can be used to hide latency
- Moving independent computations between the **`MPI_IBarrier()`** and the subsequent completion (**`MPI_Wait()`**, …) call can overlap the barrier latency

PROCESS CREATION AND MANAGEMENT

- General Description
- The Dynamic Process Model
- Process Management
- Establishing Connections

Process Creation and Management… General Description

- Important classes of MPI applications require process control
  - task farms
  - serial applications with parallel modules
  - problems that require a run-time assessment of the number and type of processes that should be started
- MPI provides a clean interface between an application and system software
- MPI guarantees communication determinism in the presence of dynamic processes
- MPI maintains a consistent concept of a communicator, regardless of how its members came into existence
- A communicator is never changed once created, and it is always created using deterministic collective operations

Process Creation and Management…
The Dynamic Process Model

- The dynamic process model allows for the creation and cooperative termination of processes after an MPI application has started
- It provides a mechanism to establish communication between the newly created processes and the existing MPI application
- It also provides a mechanism to establish communication between two existing MPI applications, even when one did not “start” the other

Process Creation and Management…
The Dynamic Process Model

Starting Processes

- MPI applications may start new processes through an interface to an external process manager
  - **`MPI_Comm_spawn()`** starts MPI processes and establishes communication with them, returning an intercommunicator
  - **`MPI_Comm_spawn_multiple()`** starts several different binaries (or the same binary with different arguments), placing them in the same MPI_COMM_WORLD and returning an intercommunicator
- A process is represented in MPI by a (group, rank) pair
- A (group, rank) pair species a unique process
- A process does not determine a unique (group, rank) pair, since a process may belong to several groups

Process Creation and Management…
Process Management

```
int MPI_Comm_spawn(const char *command, char *argv[],
  int maxprocs, MPI_Info info, int root, MPI_Comm comm,
  MPI_Comm *intercomm, int array_of_errcodes[]);
```

- **`MPI_Comm_spawn()`** tries to start **`maxprocs`** identical copies of the MPI program specified by **`command`**, establishing communication with them and returning an intercommunicator
- The spawned processes are referred to as children. The children have their own **`MPI_COMM_WORLD`**, which is separate from that of the parents
- **`MPI_Comm_spawn()`** is collective over **`comm`**
- The intercommunicator returned by **`MPI_Comm_spawn()`** contains the parent processes in the local group and the child processes in the remote group

Process Creation and Management…
Process Management

```
int MPI_Comm_spawn(const char *command, char *argv[],
  int maxprocs, MPI_Info info, int root, MPI_Comm comm,
  MPI_Comm *intercomm, int array_of_errcodes[]);
```

- **command** – name of program to be spawned (significant only at root)
- **argv** – arguments to command (significant only at root)
- **maxprocs** – maximum number of processes to start (significant only at root)
- **info** – a set of key-value pairs telling the runtime system where and how to start the processes (significant only at root)
- **root** – rank of process in which previous arguments are examined
- **comm** – intracommunicator containing group of spawning processes
- **intercomm** – intercommunicator between original group and the newly spawned group
- **array_of_errcodes** – one code per process (array of integer)

Process Creation and Management…
Process Management

```
int MPI_Comm_spawn_multiple(int count, char *commands[],
  char **argvs[], const int maxprocs[],
  const MPI_Info info[], int root, MPI_Comm comm,
  MPI_Comm *intercomm, int array_of_errcodes[]);
```

- **`MPI_Comm_spawn_multiple()`** is identical to **`MPI_Comm_spawn()`** except that there are multiple executable specifications
- The first argument, count, gives the number of specifications
- Each of the next four arguments are simply arrays of the corresponding arguments in **`MPI_Comm_spawn()`**
- All of the spawned processes have the same **`MPI_COMM_WORLD`**

Process Creation and Management…
Process Management

```
int MPI_Comm_spawn_multiple(int count, char *commands[],
  char **argvs[], const int maxprocs[],
  const MPI_Info infos[], int root, MPI_Comm comm,
  MPI_Comm *intercomm, int array_of_errcodes[]);
```

- **count** – number of commands
- **commands** – programs to be executed
- **argvs** – arguments for commands
- **maxprocs** – maximum number of processes to start for each command
- **infos** – info objects telling the runtime system where and how to start processes
- **root** – rank of process in which previous arguments are examined
- **comm** – intracommunicator containing group of spawning processes
- **intercomm** – intercommunicator between original group and the newly spawned group
- **array_of_errcodes** – one code per process (array of integer)

Process Creation and Management… Establishing Connections

- Some situations in which establishing connections are useful are:
  - Two parts of an application that are started independently need to communicate
  - A visualization tool wants to attach to a running process
  - A server wants to accept connections from multiple clients. Both clients and server may be parallel programs
- MPI must establish communication channels where there is no parent/child relationship

Process Creation and Management… Establishing Connections

- MPI must establish communication channels where there is no parent/child relationship
  - Establishing contact between two groups of processes that do not share an existing communicator is a collective but asymmetric process
  - One group of processes indicates its willingness to accept connections from other groups of processes
  - We will call this group the (parallel) server, even if this is not a client/server type of application
  - The other group connects to the server; we will call it the client

Process Creation and Management…
Establishing Connections

Server Routines

```
int MPI_Open_port(MPI_Info info, char *port_name);
```

- Establishes a network address, encoded in the **port_name** string, at which the server will be able to accept connections from clients

```
int MPI_Close_port(const char *port_name);
```

- Releases the network address represented by **port_name**

```
int MPI_Comm_accept(const char *port_name, MPI_Info info,
  int root, MPI_Comm comm, MPI_Comm *newcomm);
```

- Establishes communication with a client
- It is collective over the calling communicator. It returns an intercommunicator that allows communication with the client

Process Creation and Management…
Establishing Connections

Client Routine

```
int MPI_Comm_connect(const char *port_name, MPI_Info info,
  int root, MPI_Comm comm, MPI_Comm *newcomm);
```

- Establishes communication with a server specified by **`port_name`**.
- It is collective over the calling communicator and returns an intercommunicator in which the remote group participated in an **`MPI_Comm_accept()`**
- If the named port does not exist (or has been closed), **`MPI_Comm_connect()`** raises an error of class **`MPI_ERR_PORT`**

Process Creation and Management
Establishing Connections

Client Routine

```
int MPI_Comm_connect(const char *port_name, MPI_Info info,
  int root, MPI_Comm comm, MPI_Comm *newcomm);
```

- If the port exists, but does not have a pending **`MPI_Comm_accept()`**, the connection attempt will eventually time out after an implementation-defined time, or succeed when the server calls **`MPI_Comm_accept()`**
- In the case of a time out, **`MPI_Comm_connect()`** raises an error of class **`MPI_ERR_PORT`**

Summary

- The Nonblocking Collective Operations are discussed
- The Creation and Management of additional process in MPI program are considered

Exercises

- Develop a sample program for each method of nonblocking collective operations
- Develop a sample program using additional process in MPI program. Possible scheme to implement is “master-workers”

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.