Introduction to MPI

Daniel Langr, Pavel Tvrdík

Department of Computer Systems
Faculty of Information Technology
Czech Technical University in Prague

Parallel and Distributed Programming (MIE-PDP)
Summer Semester 2020/21, Lecture 06

https://courses.fit.cvut.cz/MIE-PDP

Lecture syllabus

Introduction

MPI — Message Passing Interface

- Standardized and portable system of **message passing** among **processes** of a parallel program/application.
- Processes can run on various computers interconnected with a communication network (model of a parallel computer with distributed memory = NUMA).
- **MPI Standard** defines the syntax and semantics of library functions for writing portable programs with message passing in languages C, C++, and Fortran (analogy to the OpenMP standard).

MPI library

- Implementation of the MPI standard.
- There exists a great number of MPI library, e.g., OpenMPI, MPICH, MVAPICH, IBM MPI, Cray MPI, Intel MPI, etc.

Source program codes: OpenMP vs MPI

OpenMP: directives and library functions

```
#include <omp.h>
int main() {     // file omp_test.cpp
   #pragma omp parallel
   { std::cout << "Thread  " << omp_get_thread_num()
               << " out of " << omp_get_num_threads() << std::endl;
} }
```

MPI: library functions only

```
#include <mpi.h>  // file mpi_test.cpp
int main(int argc, char* argv[]) {
 MPI_Init(&argc, &argv); // initialize the MPI library
 int proc_num, num_procs;
 // MPI_COMM_WORLD: communicator = all processes (see further)
 MPI_Comm_rank(MPI_COMM_WORLD, &proc_num); //process's rank
 MPI_Comm_size(MPI_COMM_WORLD, &num_procs); //the # of processes
 std::cout<<"Process "<<proc_num<<" out of "<<num_procs<<std::endl;
 MPI_Finalize(); // close the MPI library
}
```

Program compilation: OpenMP vs MPI

OpenMP: Direct support in compilers using switches

- GNU compiler:

```
g++ -fopenmp -o omp_test omp_test.cpp
```

- Intel C++ compiler:

```
icpc -qopenmp -o omp_test omp_test.cpp
```

MPI: compiler wrappers

- Special tools for compilation of the MPI library. For example, in OpenMPI:

```
mpic++ -o mpi_test mpi_test.cpp
```

- The wrapper performs linking of the MPI library, header files processing, etc.
- A special C/C++ compiler can be chosen explicitly, e.g., by setting the OpenMP system variable `OMPI_CXX`:

```
OMPI_CXX=g++ mpic++ -o mpi_test mpi_test.cpp
OMPI_CXX=icpc mpic++ -o mpi_test mpi_test.cpp
```

Program execution: OpenMP vs MPI I

OpenMP: direct execution of the compiled code

- The number of threads can be set up using, e.g., a system variable:

```
OMP_NUM_THREADS=4 ./omp_test
Thread 0 out of 4
Thread 2 out of 4
Thread 3 out of 4
Thread 1 out of 4
```

MPI: special launching tools

- For example, the `mpirun` program for the OpenMPI library. The number of processes is setup using a launcher switch:

```
mpirun -np 4 ./mpi_test
Process 1 out of 4
Process 3 out of 4
Process 0 out of 4
Process 2 out of 4
```

- Standard launchers are `mpirun` or `mpiexec`.

Program execution: OpenMP vs MPI II

OpenMP: direct execution of the compiled code

- Runtime creates the master thread as the instance of the executed code that is forked into several threads due to directive `#pragma omp parallel`.
- All threads run **on the same processor (= CPU) where the code was launched**.

MPI: special launching tools

- The `mpirun` tool launches the requested number of processes, each of them being an independent instance of the program.
- In general, these processes are launched **on different computers**, e.g., on cluster nodes.
- Distribution of processes to nodes is decided by the MPI library in collaboration with the scheduling system that controls the distributed execution of the MPI program on a cluster.

Program execution: OpenMP vs MPI III

- For example, the Salomon supercomputer in IT4I in Ostrava consists of 1008 computing **nodes**.
- Every node contains two 12-core processors and 128 GB operating memory.

OpenMP program

- OpenMP program can be executed on 24 cores at most:
  `OMP_NUM_THREADS=24 ./omp_program`
- Program has at most 128 GB available memory.

MPI program

- MPI program can utilize all $1008 \times 24 = 24192$ cores:
  `mpirun -np 24192 ./mpi_program`
- It has all 129 TB of memory available.
- MPI gives the possibility to solve much larger computing tasks.

Communication of processes/threads: OpenMP vs MPI I

- Threads in OpenMP communicate using Read/Write of shared memory.

An example of implicit linear reduction of variable i

```
int main() {
  int i = 0;
  int num_threads;
  #pragma omp parallel shared(i,num_threads)
  {//in the whole program, only 1 instance of variable i exists
     #pragma omp single
     num_threads = omp_get_num_threads();
     #pragma omp atomic update
     i++; // all threads modify the same memory location
  }
  assert(i == num_threads);//check done by the master thread only
}
```

Communication of processes/threads: OpenMP vs MPI II

- Since the reduction is a very common operation in parallel algorithms, it is supported in OpenMP.

An example of explicit reduction of variable i

```
int main() {
 int i = 0;
 int num_threads;
 #pragma omp parallel shared(num_threads) reduction(+:i)
 {//i is now a private copy of the shared variable in each thread
   #pragma omp single
   num_threads = omp_get_num_threads();
   i = 1; // each thread has access only to its private copy
 } //here the private copies into the shared variable are reduced
 assert(i == num_threads); //check done by the master thread only
}
```

Communication of processes/threads: OpenMP vs MPI III

- MPI processes **do not share** memory.
- Communication is performed exclusively with **message passing**, realized by corresponding MPI functions.
- **All variables are private**, each MPI process has its own instance.
- Reduction in MPI can be performed with, e.g., function `MPI_Allreduce`.
- After the function ends, **all processes** receive the reduced value.

Reduction in MPI

```
int main(int argc, char* argv[]) {
  MPI_Init(&argc, &argv);
  int num_procs;
  MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
  int c = 1; // private variable of the MPI process
  MPI_Allreduce(MPI_IN_PLACE, &c, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
  assert(c == num_procs); // this check is done in each process
  MPI_Finalize();
}
```

Shared variables in MPI programs: MPI+OpenMP

- Computing nodes of clusters (supercomputers) are almost exclusively composed of 1 or more multicore processors. Therefore, they have hybrid architectures with **distributed and shared** memory.
- How to compose parallel programs:
  1. **MPI Only** model: On each **core**/**processor**/**computing node**, 1 or several MPI processes are running that are **NOT forked** into threads. The shared memory within computing nodes is not utilized.
  2. **MPI and OpenMP hybrid** model: On each **computing node**/**processor**, one or several MPI processes are running and those **DO fork** using OpenMP into multiple threads running on **cores**.
  3. Hybrid models with 1 OpenMP thread/core provide higher performance than the MPI Only model with 1 process/core for most applications.
- Typical architectures:
  1. **1 MPI process per computing node**: the process forks into multiple threads corresponding to the cores of the node.
  2. **1 MPI process per processor (= socket)**: the process forks into multiple threads corresponding to the cores of the processor. Better access to data
     $\Rightarrow$ often higher performance.

MPI+OpenMP hybrid I

- Hybrid initialization is performed by function `MPI_Init_thread`.
- Its parameter defines the **requested measure of cooperation MPI with threads**:
  1. `MPI_THREAD_SINGLE`: MPI Only model, processes are not forked into threads.
  2. `MPI_THREAD_FUNNELED`: multithreaded processes are allowed with the limitation that only the master thread can call MPI functions. 1-port model.
  3. `MPI_THREAD_SERIALIZED`: multithreaded processes are allowed with the limitation that at a given time only one thread can call MPI functions (it requires critical sections). 1-port model.
  4. `MPI_THREAD_MULTIPLE`: multithreaded processes where all threads can call MPI functions without any constraints. All-port model.
- The `MPI_Init_thread` function returns the **provided measure of cooperation with the threads**.
- In general, various MPI libraries support only some variants.
- The above mentioned values of cooperation measures form a totally ordered set and can be compared.

MPI+OpenMP hybrid II

- An example of program `mpi_omp_test.cpp` that requires at least the support `MPI_THREAD_FUNNELED`.
- Then all threads write out their ranks within their processes in parallel.
- Otherwise, an error message is issued.

mpi_omp_test.cpp

```
int main(int argc, char* argv[]) {
 int provided, required = MPI_THREAD_FUNNELED;
 MPI_Init_thread(&argc, &argv, required, &provided);
 if (provided < required)
  throw std::runtime_error(
   "The MPI library does not provide required threading support");
 int proc_num;
 MPI_Comm_rank(MPI_COMM_WORLD, &proc_num);
 #pragma omp parallel
 std::cout << "Process " << proc_num
           << "Thread " << omp_get_num_thread() << std::endl;
 MPI_Finalize();
}
```

MPI+OpenMP hybrid III

- Compilation of the preceding program:

```
OMPI_CXX=g++ mpic++ -fopenmp -o mpi_omp_test mpi_omp_test.cpp
```

- An example of executing the preceding program:

```
OMP_NUM_THREADS=2 mpirun -np 2 ./mpi_omp_test
Process 0 thread 0
Process 0 thread 1
Process 1 thread 0
Process 1 thread 1
```

- Threads within the frame of MPI processes are **ranked locally** from 0.

History of MPI standards and libraries

- The 1st version **MPI 1.0** was published in June 1994.
- Current stable version of the standard is **MPI 3.1** (03/2017), it contains cca **450 functions**.
- However, common applications utilize only a small fraction of that.
- The latest work-in-progress version is **MPI 4.0**.
- See more at http://mpi-forum.org/.
- Various MPI libraries implement various versions of the MPI standards (or their parts).
- All current MPI libraries implement the basic MPI functionality important for this course.

Groups of MPI processes

- Every MPI process is always part of at least one **process group**.
- Within each group, the processes are numbered from 0 to (GroupSize -1).
- The basic implicit group contains **all** processes created during the initialization of the MPI program.
- **New** process groups can be formed.
- If a given process is a part of more groups, its **ranking** within the groups can differ.

MPI communicators I

- Every MPI communication function has as a parameter called **communicator**.
- The communicator determines **a group of processes within which the communication is performed**.
- **Intra-communicator** is associated with a specific process group and defines the communication within the group..
- `MPI_COMM_WORLD` is the implicit predefined intra-communicator for the group of **all** MPI processes of the program.
- **Inter-communicator** is associated with 2 different processes groups and determines the communication between these groups (will not be discussed further).

MPI communicators II

MPI_Comm_rank

- Function MPI_Comm_rank returns **process rank** within the group associated with the given intra-communicator.

MPI_Comm_size

- Function MPI_Comm_size returns the **number of processes** of the process group associated with the given intra-communicator.

- An example of determining the number of processes and local ranks within a given process group:

```
int proc_num, num_procs;
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
//num_procs == the value of parameter -np of the mpirun command
MPI_Comm_rank(MPI_COMM_WORLD, &proc_num);
// 0 <= proc_num < num_procs
```

MPI communication operations: Taxonomy

Point-to-point vs collective communication operation

- **Point-to-point** communication operations define communication between two MPI processes.
- **Collective** communication operations define communication among all MPI processes associated with the given communicator.

Blocking vs nonblocking communication operations

- **Blocking** communication operations: the corresponding MPI function is completed and returns after the communication operation satisfies a given condition.
- **Nonblocking** communication operation: the corresponding MPI function returns immediately and the real completion of the communication operation must be explicitly tested/forced.

Basic point-to-point communication between processes

- A **source process** calls the **MPI_Send** function whose one argument is the destination.
- A **destination process** calls the **MPI_Recv** function whose one argument is the source.
- An example of implementation of **One-to-All Broadcast** (OAB):
  - The process with rank 0 sends the same value to all remaining processes.
  - Linear time complexity.

```
int proc_num, num_procs; // process rank, the number of processes
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
MPI_Comm_rank(MPI_COMM_WORLD, &proc_num);
if (proc_num == 0) { // the process with rank 0
   int value = ... // computation of the broadcast value
   for (int dest = 1; dest < num_procs; dest++)
      MPI_Send(&value, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
}
else { // remaining processes
   int value;
   MPI_Recv(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Function MPI_Send by MPI standard

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);
```

- buf: a pointer to the sent data (e.g., a pointer to a variable or the first array element);
- count: the number of the sent items (e.g., 1 in case of a scalar variable or the number of array elements);
- datatype: data type of the sent data (see Slide 24);
- dest: the rank of the destination process within the given communicator;
- tag: data tag (see Slide 27);
- comm: a valid MPI communicator (e.g., MPI_COMM_WORLD).

Function MPI_Recv by the MPI standard

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm,
             MPI_Status *status);
```

- buf: a pointer to the reception buffer (e.g., a pointer to a variable or the first array element);
- count: the maximal number of received elements (e.g., 1 in case of a variable or the number of array elements);
- datatype: data type of the received data (see Slide 24);
- source: the rank of the source process within the given communicator;
- tag: data tag (see Slide 27);
- comm: a valid MPI communicator (e.g., MPI_COMM_WORLD);
- status: a pointer to the **status object**) (see Slide 28).

Point-to-point communication: Type of transferred data

- MPI functions must know the **type of transferred data**: specified by the `datatype` parameter of the `MPI_Datatype` type.
- MPI standard defines the values of this parameter for basic data types of C/C++, e.g.,: `MPI_CHAR`, `MPI_INT`, `MPI_UNSIGNED`, `MPI_LONG`, `MPI_UNSIGNED_LONG`, `MPI_INT16_T`, `MPI_UINT64_T`, `MPI_FLOAT`, `MPI_DOUBLE`, etc.

An example:

```
int i; uint64_t u; double d;
MPI_Send(&i, 1, MPI_INT, ...);
MPI_Send(&u, 1, MPI_UINT64_T, ...);
MPI_Recv(&d, 1, MPI_DOUBLE, ...);
```

- For structured data types, a corresponding value for the `datatype` parameter van be constructed using the `MPI_Type_create...()` function.
- For example, `MPI_Type_create_struct()` for structured data types.

Point-to-point communication: The amount of transferred data

- MPI functions can simultaneously transfer more data elements. They all must be just of the **same data type** and must be stored in **continuous part of the memory**.
- Specification of the `count` parameter.

An example

```
int i;
float f[50];
uint64_t* u = (uint64_t*)malloc(100 * sizeof(uint64_t));
std::vector<double> d(n); //vector guarantees storing continuously
.............. // initialization
MPI_Send(&i, 1, MPI_INT, ...);
MPI_Send(&f[0], 50, MPI_FLOAT, ...); //can be f instead of &f[0]
MPI_Send(u, 100, MPI_UINT64_T, ...);
MPI_Send(&d[0], n, MPI_DOUBLE, ...);
```

Point-to-point communication: Source and destination processes

- A source process determines the destination using parameter `dest`:

```
if (proc_num == 0) {
   for (int dest = 1; dest < num_procs; dest++)
      MPI_Send(&i, 1, MPI_INT, dest, ...);          }
```

- A destination process can receive data **only** from a **specific** source process:

```
else { // variant I.
   MPI_Recv(&i, 1, MPI_INT, 0, ...);                }
```

- or from an **arbitrary** source process using a special value `MPI_ANY_SOURCE` of parameter `dest`:

```
else { // variant II.
   MPI_Recv(&i, 1, MPI_INT, MPI_ANY_SOURCE, ...);    }
```

Point-to-point communication: Message tagging

- **Tags** enable to distinguish messages with distinct **semantic** meaning.
- Source process prescribes the tag values of sent messages:

```
static const int result_tag = 0;
static const int temporary_tag = 1;
MPI_Send(&result, 1, MPI_INT, dest, result_tag, ...);
MPI_Send(&temp, 1, MPI_INT, dest, temporary_tag, ...);
```

- The destination process can receive messages with a **specific tag**:

```
MPI_Recv(&result, 1, MPI_INT, source, result_tag, ...);
MPI_Recv(&temp, 1, MPI_INT, source, temporary_tag, ...);
```

- or with an **arbitrary tag** using a special parameter value `MPI_ANY_TAG`:

```
MPI_Recv(&result_or_temp, 1, MPI_INT, source, MPI_ANY_TAG, ...);
```

Point-to-point communication: Status object I

- Some MPI functions, e.g., MPI_Recv, return on demand a pointer to the **status object**, which is a variable of type MPI_Status:

```
MPI_Status status;
MPI_Recv(&result_or_temp, 1, MPI_INT, MPI_ANY_SOURCE,
         MPI_ANY_TAG, MPI_COMM_WORLD, &status);
```

- If we are not interested in the status, we can use the special argument MPI_STATUS_IGNORE:

```
MPI_Recv(..., MPI_STATUS_IGNORE);
```

- MPI_Status is a structure containing items:
  1. the rank of the source process MPI_SOURCE,
  2. the tag of the received message MPI_TAG,
  3. the number of received elements that can be obtained by function MPI_Get_count.

Point-to-point communication: Status object II

An example

```
static const int count = 1000;
std::vector<int> v(count);
MPI_Status status;
MPI_Recv(&v[0],
   count, // the maximal number of received data (the buffer size)
   MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

std::cout<<"Source process No.: "<<status.MPI_SOURCE<<std::endl;
std::cout<<"Message tag: "<<status.MPI_TAG<<std::endl;

int received;
MPI_Get_count(&status, MPI_INT, &received);
// received == the real number of received items (message size)
std::cout << "Elements received: " << received << std::endl;
v.resize(received); // v now contains only the received elements
```

An example of a Master-Slave program I

- Process 0 (Master, M) schedules the work to the Slave processes (S).
- When S completes its assignment, it will send M a message with the `tag_done` tag and waits for a new task assignment, i.e., the message with the `tag_work` tag:

```
while (true) { // do the work
  MPI_Send(..., 0, tag_done, MPI_COMM_WORLD);
  MPI_Recv(..., 0, tag_work, MPI_COMM_WORLD, MPI_STATUS_IGNORE);}
```

- M waits until some S completes its task (waiting for message with the `tag_done` tag). M does not know which S completes its work as the first one, it must therefore accept messages from all S processes:

```
MPI_Recv(..., MPI_ANY_SOURCE, tag_done, MPI_COMM_WORLD, &status);
```

- Then M sends S another work; the S's rank discovers from the status object:

```
MPI_Send(..., status.MPI_SOURCE, tag_work, MPI_COMM_WORLD);
```

An example Master-Slave program II

- The previous template does not address the **control** and **termination** of the parallel execution. Possible modifications:

Master process

- Initially, it distributes the work to all Ss.
- Then it waits until Ss complete their work assignments (`tag_done`). After reception:
  1. if there is more work to do, it will send it to the given S (message with the `tag_work` tag);
  2. otherwise it informs the given S about termination of the computation (message with the `tag_finished` tag).

Slave process

- In a loop, it receives the work assignments from M with any tag:
  1. if the tag is `tag_finished`, it is terminated.
  2. if the tag is `tag_work`, it performs the work and subsequently send the message with the `tag_done` tag.

The final Master-Slave program template

```
MPI_Status status;
if (proc_num == 0) { // master process:
   for (int dest = 1; dest < num_procs; dest++) // for all Ss
      MPI_Send(..., dest, tag_work, MPI_COMM_WORLD); //initial work distribution
   int working_slaves = num_procs - 1; // the number of slaves
   while (working_slaves > 0) { // the main loop
      MPI_Recv(..., MPI_ANY_SOURCE, tag_done, MPI_COMM_WORLD, &status);
      if (more_work_to_be_done) // if there is more work to do
         MPI_Send(..., status.MPI_SOURCE, tag_work, MPI_COMM_WORLD);
      else {
         MPI_Send(..., status.MPI_SOURCE, tag_finished, MPI_COMM_WORLD);
         working_slaves--;
}  }  } else { // slave processes:
   while (true) {
      MPI_Recv(..., 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      if (status.MPI_TAG == tag_finished) break; // end of computation
      else if (status.MPI_TAG == tag_work) {
         // do the work
         MPI_Send(..., 0, tag_done, MPI_COMM_WORLD);
}  }  }
```

Return values of the MPI functions and error handling I

Indication of successful completion of MPI functions:

- Every MPI function (except few exceptions) return
  - value `MPI_SUCCESS` in case of **successful completion**,
  - **error code** in case of **unsuccessful completion**.
- MPI defines error classes and there are functions that convert an error code into a text and find the class of an error code.

Implicit predefined error handlers:

- The handler of an error of an MPI function is **associated to its communicator**.
- The error handler implicitly associated to the `MPI_COMM_WORLD` communicator is `MPI_ERRORS_ARE_FATAL`.
- This handler **aborts** the whole MPI **program**, all executing processes.
- This is the default reaction on an MPI function error.
- So an MPI function that ended up with an error does not return at all.
- So if `MPI_ERRORS_ARE_FATAL` handler is used, it makes no sense to check the return values of MPI functions.

Return values of the MPI functions and error handling II

- Second predefined error handler is `MPI_ERRORS_RETURN`. It does not abort the MPI program, it just returns the error code of the MPI function.
- In addition to these 2 predefined error handlers, an MPI implementation can predefine further handlers and a user can define his own.
- BUT! After an error is detected, the state of MPI is undefined. So, even if using `MPI_ERRORS_RETURN` we get the error code, it does not mean that the MPI program can normally continue, it depends strongly on the implementation. Usually, we can just make the diagnosis and issue user-defined error messages.
- A predefined of user-define error handler can be attached to a communicator using function `MPI_Comm_set_errhandler`.

An example

```
MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
int error_code;
error_code = MPI_Send(..., MPI_COMM_WORLD);
if (error_code != MPI_SUCCESS) {
   ... } // user defined error handler
```

Communication modes of blocking operations I

- MPI_Send is so called **blocking operation**. It completes and returns if the input buffer can be modified (data is gone):

```
int c = 10;
MPI_Send(&c, 1, MPI_INT, ...);
c = 20; //the sent data are intact, the destination gets value 10
```

- MPI_Send uses so called **standard mode** and it is a **nonlocal** operation: It can be started whether or not a matching receive has been posted. It returns only after the source data were
  1. either **received** by the destination process,
  2. or **copied** into a system buffer for sending them later.
- **The choice upon the MPI library** and unknown to the user. So a user cannot assume anything and should leave it to the system.
- In the former case, MPI_Send will not complete until a matching receive has been posted, and the data has been moved to the receiver.
- In the latter case, the MPI_Send function returns after copying the send buffer to the system one, since MPI takes responsibility for data delivery.

Communication modes of blocking operations II

- **buffered mode** — function `MPI_Bsend` is a **local** operation: It can be started whether or not a matching receive has been posted. MPI **always stores** source data into a temporary system buffer that the user must allocate before that using function `MPI_Buffer_attach`. Function `MPI_Bsend` returns as soon as data are stored in the system buffer, no matter in which state is the receiver. So the memory requirements double. If the system buffer is not big enough, function `MPI_Bsend` returns an error code.
- **synchronous mode** — function `MPI_Ssend` is a **nonlocal** operation: It can be started whether or not a matching receive has been posted. It returns successfully only if a matching receive is posted and the receive operation has started to receive the source data. Hence, both sender and receiver wait one for another in whichever ordering (rendezvous).
- **ready mode** — function `MPI_Rsend` is a **nonlocal** operation: It can start the communication only if the matching receive is already posted. Then the source data are delivered to the receiver as in the synchronous mode. Otherwise, it returns with error code and its outcome is undefined.

Nonblocking communication operations

- Functions `MPI_Send`, `MPI_Bsend`, `MPI_Ssend`, and `MPI_Rsend` are blocking in the sense that after they return, the input buffer (parameter `buf`) can be safely modified (data are gone).
- Function `MPI_Recv` is blocking in the sense that after it returns, the received data are safely in its buffer (parameter `buf`) and can be read.

- **Nonblocking functions** `MPI_Isend`, `MPI_Ibsend`, `MPI_Issend`, and `MPI_Irsend` are **local** operations: they initiate the data sending and return immediately.
- `MPI_Irsend` can successfully start only if the matching receive is already posted, the other 3 can start whether or not a matching receive has been posted.
- The input data buffer **cannot be modified** until it is not explicitly **ensured** that the source data are really out of the source buffer.
- **Nonblocking function** `MPI_Irecv` just initiates data reception. The store buffer **can be read** only if it is explicitly checked later on that the received data is really stored in it.

Completion of nonblocking communication operations I

- **All nonblocking** functions have an extra parameter: a pointer to a variable of type `MPI_Request` that serves for **completion testing** of nonblocking operations.
- The **completion testing** is performed by the `MPI_Test` function.
- The **waiting for completion** is performed by the `MPI_Wait` function.

```
int c = 10;
MPI_Request request;
MPI_Isend(&c, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
//variable c can't be modified, the data being sent would change
```

```
int flag;
MPI_Status status;
MPI_Test(&request, &flag, &status);
// if flag == 1, variable c can be modified, otherwise not
```

```
MPI_Wait(&request, &status);
//variable c can be safely modified now
```

Completion of nonblocking communication operations II

- It follows that at a nonblocking receive we obtain the status object from the functions `MPI_Test` or `MPI_Wait`, **NOT** from the `MPI_Irecv` function itself!!!

```
int c;
MPI_Request request;
MPI_Irecv(&c, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
          MPI_COMM_WORLD, &request);
// variable c has not been defined yet
MPI_Status status;
MPI_Wait(&request, &status);
// until now variable c has the received value
std::cout << "Received value" " << c << std::endl;
std::cout << "Source process: " << status.MPI_SOURCE << std::endl;
```

Completion of nonblocking communication operations III

A complete example

```
if (proc_num == 0) {
  int c = 10; MPI_Request request;
  MPI_Isend(&c, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
  .... // variable c cannot be modified now
  int flag;
  MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
  if (flag) {.... // variable c can be unconditionally modified
  } else {
     MPI_Wait(&request, MPI_STATUS_IGNORE);
     .... // variable c can be unconditionally modified
  }
} else if (proc_num == 1) {
  int c; MPI_Request request;
  MPI_Irecv(&c, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &request);
  .... // variable c has not been defined yet
  MPI_Status status;
  MPI_Wait(&request, &status);
  .... // now variable c has the value of received data
  std::cout << "Received value" " << c << std::endl;
  std::cout << "Source process: " << status.MPI_SOURCE << std::endl;
}
```

Completion of nonblocking communication operations IV

- Functions `MPI_Test` and `MPI_Wait` test/wait on the completion of **one** nonblocking operation.
- Functions `MPI_Testany` and `MPI_Waitany` test/wait on the completion of **arbitrary** nonblocking operation **from a certain set** specified by the parameter of type `MPI_Request[]`.
- Functions `MPI_Testall` and `MPI_Waitall` test/wait on completion of **all** nonblocking operations **from a certain set** (specified equally).

An example

```
MPI_Request requests[3];
MPI_Irecv(..., &requests[0]);
MPI_Irecv(..., &requests[1]);
MPI_Irecv(..., &requests[2]);
MPI_Status statuses[3];
MPI_Waitall(3, requests, statuses);
```

Completion of nonblocking communication operations V

- Completion and Return from blocking functions `MPI_Send`, `MPI_Bsend`, `MPI_Ssend`, `MPI_Rsend`, and `MPI_Recv` depends on the status of the communication operation.
- Nonblocking functions `MPI_Isend`, `MPI_Ibsend`, `MPI_Issend`, `MPI_Rssend`, and `MPI_Irecv` return immediately **independently** on the status of the communication operation which is just initiated.
- Functions of type `MPI_Test` wait for the required state of the communication using the method of **active waiting/polling**, whereas `MPI_Wait`-like functions represent **blocking** methods.
- A blocking send can be matched with a nonblocking receive and vice versa
- The distinction between `MPI_Bsend` and `MPI_Ibsend` is just in that `MPI_Bsend` returns only if the source buffer is copied into the system one, whereas `MPI_Ibsend` returns immediately while the copying into the system buffer is still in progress.

More details here.

Function MPI_Sendrecv I

- It allows to **send and receive messages simultaneously**.
- The source and destination processes can (but need not) be distinct.

```
int MPI_Sendrecv(const void *sendbuf,
 int sendcount, MPI_Datatype sendtype, int dest,
 int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype,
 int source, int recvtag, MPI_Comm comm, MPI_Status *status)
```

An example: data exchange between processes 0 and 1

```
int value_to_recv, value_to_send = ...;
if (proc_num == 0)
   MPI_Sendrecv(&value_to_send, 1, MPI_INT, 1, 0,
                &value_to_recv, 1, MPI_INT, 1, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
else if (proc_num == 1)
   MPI_Sendrecv(&value_to_send, 1, MPI_INT, 0, 0,
                &value_to_recv, 1, MPI_INT, 0, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Function MPI_Sendrecv II

Simplified version of the previous solution

```
int value_to_recv, value_to_send = ...;
if (proc_num < 2)
   MPI_Sendrecv(&value_to_send, 1, MPI_INT, (proc_num + 1) % 2, 0,
                &value_to_recv, 1, MPI_INT, (proc_num + 1) % 2, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Yet another simplification using function MPI_Sendrecv_replace

```
int value = ...;
if (proc_num < 2)
   MPI_Sendrecv_replace(
      &value, 1, MPI_INT, // unified for sending and receiving
      (proc_num + 1) % 2, 0,
      (proc_num + 1) % 2, 0,
      MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

An example of the cyclic shift permutation

Cyclic shift permutation

Process `proc_num` sends data to process `(proc_num + 1)% num_procs`.

- Cyclic shift is a very common communication operation.

Incorrect code using `MPI_Send` and `MPI_Recv`

```
// all processes:
int next = (proc_num + 1) % num_procs;
int prev = (proc_num + num_procs - 1) % num_procs;
MPI_Send(&value_to_be_sent, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
MPI_Recv(&value_to_be_received, 1, MPI_INT, prev, ...);
```

- `MPI_Send` in the **standard mode** does not need to return if the data reception is not initiated. If this happens to **all** processes (the decision is upon the MPI library), a **deadlock** appears!!!
- If the synchronous mode `MPI_Ssend` were used instead of `MPI_Send`, then the deadlock appears **always**!

Correct solution of the cyclic shift I

- Let us start with a correct solution based on `MPI_Send` and `MPI_Recv`.
- Two phase protocol: First **even-ranked** processes send data to **odd-ranked** ones and then vice versa.

```
int next = (proc_num + 1) % num_procs;
int prev = (proc_num + num_procs - 1) % num_procs;
bool odd = proc_num % 2;
if (!odd) { // even-rank processes
 MPI_Send(&value_to_be_sent, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
 MPI_Recv(&value_to_be_received, 1, MPI_INT, prev, ...);
} else { // odd-rank processes
 MPI_Recv(&value_to_be_received, 1, MPI_INT, prev, ...);
 MPI_Send(&value_to_be_sent, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
}
```

Questions

- Does it work for odd number of processes even if then the first and last processes are even-ranked?
- How many rounds are needed for even and odd `num_procs`?

Correct solution of the cyclic shift II

- A solution based on `MPI_Bsend` (= *buffered mode*).
- The `MPI_Bsend` function **is guaranteed to return** even if the reception has not been initiated yet.

```
// All processes:
int next = (proc_num + 1) % num_procs;
int prev = (proc_num + num_procs - 1) % num_procs;
MPI_Buffer_attach(...);//system buffer allocation for MPI_Bsend
MPI_Bsend(&value_to_be_sent, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
MPI_Recv(&value_to_be_received, 1, MPI_INT, prev, ...);
MPI_Buffer_dettach(...); // system buffer deallocation
```

Notes:

- MPI must store the source data into a temporary system buffer. It can be inefficient for big data.
- If `MPI_Bsend` requires more memory than was preallocated, it returns an error code.
- Details about allocation and deallocation of a system buffer are skipped.

Correct solution of the cyclic shift III

- A solution based on **nonblocking** MPI_Isend that returns immediately.

```
// all processes:
int next = (proc_num + 1) % num_procs;
int prev = (proc_num + num_procs - 1) % num_procs;
MPI_Request request;
MPI_Isend(&value_to_be_sent, 1, MPI_INT, next, 0, MPI_COMM_WORLD,
                  &request);
MPI_Recv(&value_to_be_received, 1, MPI_INT, prev, ...);
MPI_Wait(&request, MPI_STATUS_IGNORE);
```

Notes:

- Except for MPI_Recv also MPI_Wait is blocking. It **can return** if the data reception is initiated.
- The code will remain correct even if MPI_Isend is replaced with MPI_Issend. In such a case, if MPI_Wait **returns** then the data reception was certainly initiated.

Correct solution of the cyclic shift IV

- A solution based on the `MPI_Sendrecv` function.
- Each process behaves as 2-port (i.e., it is able to receive data from the left and **simultaneously** to send data to the right).
- Practically, this is the simplest and most efficient solution.

```
// all processes:
int next = (proc_num + 1) % num_procs;
int prev = (proc_num + num_procs - 1) % num_procs;
MPI_Sendrecv(
   &value_to_be_sent, 1, MPI_INT, next, 0,
   &value_to_be_received, 1, MPI_INT, prev, 0,
   MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Cyclic shift permutation: Wrap-up

- We have seen 4 different solutions of the cyclic shift permutation on a virtual ring of processes.
- In MPI, many problems can be solved in **a number of** ways.
- The selection of the simplest or most efficient solution depends on the **programmer experience**, but also on the **architecture** of the target cluster/supercomputer.
- The fastest solution must often be found by **performance evaluation** and **profiling**.

Message arrival probing I

```
MPI_Iprobe(int source,int tag,MPI_Comm comm,int *flag,MPI_Status *status);
```

- Function `MPI_Iprobe` allows incoming messages to be checked for, without really receiving them. It is **nonblocking**, it returns immediately.
- It returns `flag == true` **if there is a message that can be received and matches** the pattern specified by arguments `source`, `tag`, `comm`.
- The call matches the same message that would have been received by a call to `MPI_Recv(...,source,tag,comm,status)` executed at the same point in the program, and returns in `status` the same value that would have been returned by `MPI_Recv()`.
- Otherwise the call returns `flag == false` and leaves `status` undefined.
- If it returns `flag == true`, then the content of the status object can be subsequently investigated as explained on Slide 28.
- A subsequent receive executed with the same `source`, `tag`, `comm` returned in status by `MPI_Iprobe` will receive the message that was matched by the probe **if no other intervening receive occurs after the probe and the send is not successfully canceled before the receive**.

Message arrival probing II

- If the receiving process is multithreaded, it is the user's responsibility to ensure that the last condition on the previous slide holds.
- It is not necessary to receive a message immediately after it has been probed for, and the same message may be probed for several times before it is really received.
- The source argument of `MPI_Iprobe` can be `MPI_ANY_SOURCE` and the tag argument can be `MPI_ANY_TAG` , so that one can probe for messages from an arbitrary source and/or with an arbitrary tag.

```
MPI_Probe(int source,int tag,MPI_Comm comm,MPI_Status *status);
```

- Function `MPI_Probe` behaves like `MPI_Iprobe` except that it is a **blocking** call that returns only after a matching message has been found.

Applications of message arrival probing functions

1. Arrival of „optional“ messages, e.g., premature termination of the computation in case that the optimal solution has been found by another process.
2. Reception of a message of unknown size: finding out the message size using `MPI_Probe`, buffer allocation, data reception using `MPI_Recv`.

An example

```
MPI_Status status; int flag;
MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &flag, &status);
if (flag) { // a message arrival is available
  int count;
  MPI_Get_count(&status, MPI_INT, &count); //find out the message size
  std::vector<int> buf(count); // receive buffer allocation (C++)
  MPI_Recv(&buf[0],count, MPI_INT, status.MPI_SOURCE, status.MPI_TAG,
      MPI_COMM_WORLD, MPI_STATUS_IGNORE); // reception of the message
  ...
```