IN3200/IN4200: Chapter 9
Distributed-memory parallel programming with MPI

Textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

Objective

- The concept of explicit “message passing”
- Introduction to **MPI** (the Message Passing Interface)

Message passing

- Shared-memory programming (such as OpenMP) does not work for distributed-memory parallel computers
    - There is no way for one processor to directly access the address space of another process
- Explicit "**message passing**" is required on distributed-memory systems
    - This can also be a programming model for shared-memory or hybrid systems

Basic features of message-passing programming

- The same program runs on all processes (Single Program Multiple Data, or **SPMD**)—no difference from OpenMP programming in this regard
- The work of each process is implementation in a sequential language (such as C)
  - Data exchange (sending and receiving messages) is done via calls to an appropriate library
- All variables in a process are **local** to this process (nothing is shared)

"Messages" carry data to be exchanged between processes

Information needed about a message:

- Which process is sending the message?
- Where is the data on the sending process?
- What kind of data is being sent?
- How much data constitutes the message?
- Which process is going to receive the message?
- Where should the data be placed on the receiving process?
- What amount of data is the receiving process prepared to accept?

MPI (Message Passing Interface)

MPI is a *library standard* for programming distributed memory

- MPI implementation(s) available on almost every major parallel platform (also on shared-memory machines)
- Portability, good performance & functionality
- Collaborative computing by a group of individual processes
- Each process has its own local memory
- Explicit message passing enables information exchange and collaboration between processes

MPI C language binding

```
#include <mpi.h>

rv = MPI_Xxxxx(parameter, ... )
```

Beware of the **case-sensitive** naming pattern.

The return value (rv) transports information about the success of the MPI operation. (**MPI_SUCCESS** is returned if the MPI routine completed successfully.)

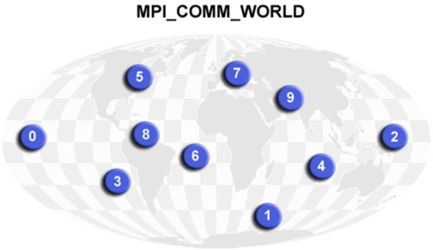
MPI_Init

The `MPI_Init` function initializes the parallel environment.

```
int MPI_Init( int *argc, char ***argv )
```

The `MPI_Init` function takes pointers to the `main()` function's input arguments so that the library can evaluate and remove any additional command line arguments that may have been added by the MPI startup process.

MPI communicator



https://computing.llnl.gov/tutorials/mpi/

- An MPI communicator is a "communication universe" for a group of processes
- `MPI_COMM_WORLD` – name of the default MPI communicator, i.e., the collection of all processes (started by an execution)
- Each process in a communicator is identified by its unique rank
- Almost every MPI command needs to provide a communicator as input argument

MPI process rank

- Each process has a unique rank, i.e. an integer identifier, within a communicator
- The rank value is between 0 and #procs-1
- The rank value is used to distinguish one process from another
- Commands MPI_Comm_size & MPI_Comm_rank are very useful

```
int size, my_rank;
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
```

MPI_Finalize

An MPI parallel program is shut down by a call to `MPI_Finalize()`.

Note that no MPI process except rank 0 is guaranteed to execute any code beyond `MPI_Finalize()`.

“Hello world” in MPI

```
#include <stdio.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  printf("Hello world, I’ve rank %d out of %d procs.\n",
         my_rank,size);
  MPI_Finalize ();
  return 0;
}
```

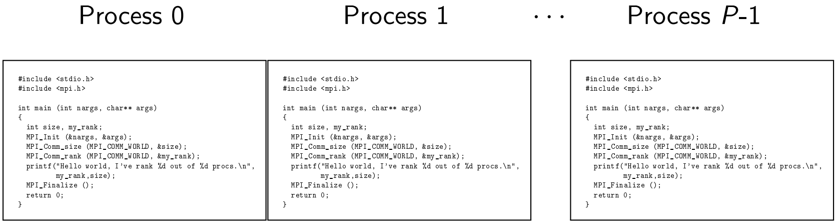
Example running result of "Hello world"

- Example of compilation: `mpicc hello_world_mpi.c`
- Example of parallel execution: `mpirun -np 4 ./a.out`
- Example running result (note: no deterministic order of the output):

```
Hello world, I've rank 2 out of 4 procs.
Hello world, I've rank 1 out of 4 procs.
Hello world, I've rank 3 out of 4 procs.
Hello world, I've rank 0 out of 4 procs.
```

Note: Compilation and execution can vary from system to system.

An abstract "picture" of parallel execution



Process 0

```
#include <stdio.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  printf("Hello world, I've rank %d out of %d procs.\n",
         my_rank,size);
  MPI_Finalize ();
  return 0;
}
```

Process 1

```
#include <stdio.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  printf("Hello world, I've rank %d out of %d procs.\n",
         my_rank,size);
  MPI_Finalize ();
  return 0;
}
```

$\cdots$

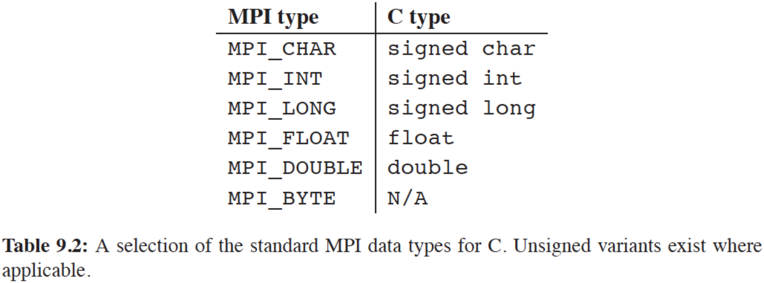
Process P-1

```
#include <stdio.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  printf("Hello world, I've rank %d out of %d procs.\n",
         my_rank,size);
  MPI_Finalize ();
  return 0;
}
```

MPI message

An MPI message is simply an array of elements of a particular MPI data type.

Data types can either be standard types (pre-defined) or *derived types*, which must be defined by appropriate MPI calls.



MPI type	**C type**
MPI_CHAR	signed char
MPI_INT	signed int
MPI_LONG	signed long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_BYTE	N/A

Table 9.2: A selection of the standard MPI data types for C. Unsigned variants exist where applicable.

Point-to-point communication

- One “sender” and one “receiver”: point-to-point communication
- Both the sender and receiver are identified by their ranks (within an MPI communicator)
- Each point-to-point message can carry an additional integer label, the so-called **tag**

The simplest MPI send command

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

This *blocking* send function returns when the data has been delivered to the system and **the message buffer can be safely modified**. The message may not have been received by the destination process.

The simplest MPI receive command

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

- This *blocking* receive function waits until a matching message is received from the system so that the buffer contains the incoming message.
- Match of data type, source process (or `MPI_ANY_SOURCE`), message tag (or `MPI_ANY_TAG`).
- Receiving fewer `datatype` elements than `count` is ok, but receiving more elements is an error.

MPI_Status

The MPI_Recv function has an additional output argument: the status object, which can be used to determine "unknown" parameters (if any).

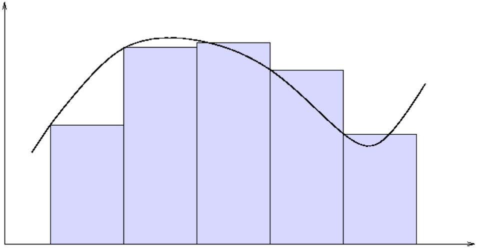
For example, the source or tag of a received message may not be known if wildcard values were used in the receive function.

To query the information stored in a status object:

```
status.MPI_SOURCE

status.MPI_TAG

MPI_Get_count (MPI_Status *status,
               MPI_Datatype datatype,
               int *count);
```

Example of MPI parallelization; numerical integration

A quick recap of numerical integration (already discussed in Chapter 6):



http://spiff.rit.edu/classes/phys317/lectures/num_integ2/num_integ2.html

The basic idea:

- Divide the integral domain into many small intervals
- Evaluate the function inside each small interval
- Sum up all the evaluated results (multiplied with the interval width)

MPI parallelization; overall idea

- Each MPI process is assigned a “subdomain’ to work on
- Each MPI process then computes its subdomain result
- Thereafter, all the subdomain results need to be summed up
- One possible strategy is to let each process (except rank 0) send its subdomain result to process rank 0, which sums up all the subdomain results

MPI implementation

```
int rank, size;
double a=0.0, b=1.0, mya, myb, psum;
MPI_Status status;

MPI_Init (&nargs, &args);
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);

mya = a + rank*(b-a)/size;
myb = mya + (b-a)/size;
psum = integrate(mya,myb); // Each integrates over its "subdomain"

if (rank==0) {
  double res = psum;
  for (i=1; i<size; i++) {
    MPI_Recv(&psum,1,MPI_DOUBLE,i,0,MPI_COMM_WORLD,&status);
    res += psum;
  }
  printf("Result: %g\n", res);
}
else {
  MPI_Send(&psum,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
}
MPI_Finalize();
```

Additional remarks

The preceding MPI example can be improved in several ways:

- Rank 0 receives in total `size-1` messages, one from each of the other processes. The order of receiving the messages is fixed, probably `not` the same as the incoming order of the messages. Using `MPI_ANY_SOURCE` (wildcard) instead of a prescribed sender rank is more appropriate.
- Rank 0 calls `MPI_Recv` after its own calculation (`integrate(mya,myb)`). If some of the other processes finish their computation earlier, communication cannot proceed, and it cannot be overlapped with computation on rank 0. (Non-blocking point-to-point communication will be more appropriate, see Section 9.2.4.)
- Rank 0 can easily become a "bottleneck", because it is responsible for receiving all the partial results. (Use of collective communication, such as that specifically designed for "reduction", will be more appropriate, see Section 9.2.3.)

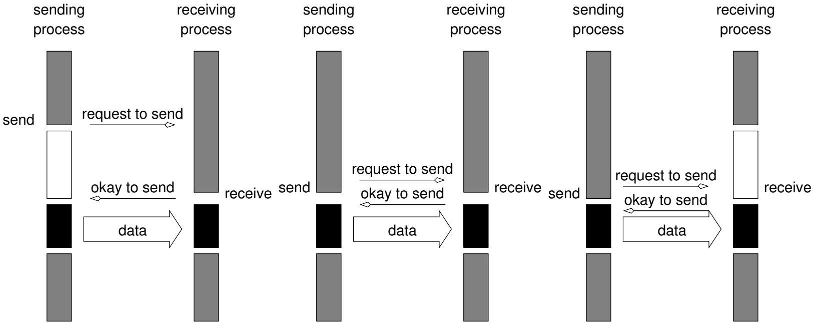
“Uncertainties/risks” with MPI_Send

While MPI_Send is easy to use, one should be aware that the MPI standard allows for a considerable amount of freedom in its actual implementation.

- Internally it may work completely **synchronously**, meaning that the call can not return until a message transfer has at least started after a “handshake” with the receiver.
- However, it may also copy the message to an intermediate system buffer and return right away, leaving the “handshake” and data transmission to another mechanism, like a background thread.
- It may even change its behavior depending on any explicit or hidden parameters.

Scenarios of "handshake"

Non-System-Buffered Blocking Message Passing Operations



(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at receiver

Source: A. Grama, A. Gupta, G. Karypis, and V. Kumar. **Introduction to Parallel Computing**. Addison Wesley, 2003

One scenario for deadlock

Deadlocks may occur if the possible synchronousness of MPI_Send is not taken into account. A typical communication pattern where this may become crucial is a “ring shift”:

```
int rank, size, left, right,in_buf[N], out_buf[N];
MPI_Status status;

// .......

MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);

right = rank==size-1 ? 0 : rank+1;
left = rank==0 ? size-1 : rank-1;

MPI_Send(out_buf,N,MPI_INT,right,0,MPI_COMM_WORLD);
MPI_Recv(in_buf,N,MPI_INT,left,0,MPI_COMM_WORLD,&status);
```

Depiction of the “ring shift” pattern

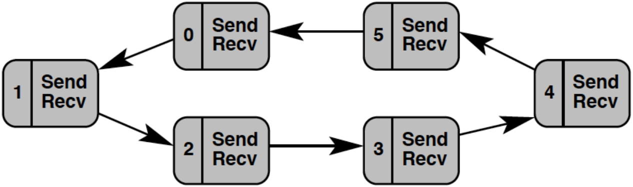


Figure 9.1: A ring shift communication pattern. If sends and receives are performed in the order shown, a deadlock can occur because `MPI_Send()` may be synchronous.

- If MPI_Send is **synchronous** (and there is no system buffering), all processes call it first and then wait forever for a matching receive to be posted—deadlock.
- However, the ring shift **may** run without problems if the messages are sufficiently short. In fact, most MPI implementations provide a (small) internal buffer for short messages and switch to synchronous mode when the buffer is full or too small.

One simple solution

```
int rank, size, left, right,in_buf[N], out_buf[N];
MPI_Status status;

// .......

MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);

right = rank==size-1 ? 0 : rank+1;
left = rank==0 ? size-1 : rank-1;

if (rank%2) {
  MPI_Recv(in_buf,N,MPI_INT,left,0,MPI_COMM_WORLD,&status);
  MPI_Send(out_buf,N,MPI_INT,right,0,MPI_COMM_WORLD);
}
else {
  MPI_Send(out_buf,N,MPI_INT,right,0,MPI_COMM_WORLD);
  MPI_Recv(in_buf,N,MPI_INT,left,0,MPI_COMM_WORLD,&status);
}
```

Depiction of the simple solution

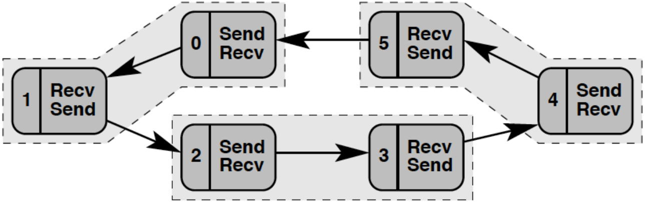


Figure 9.2: A possible solution for the deadlock problem with the ring shift: By changing the order of `MPI_Send()` and `MPI_Recv()` on all odd-numbered ranks, pairs of processes can communicate without deadlocks because there is now a matching receive for every send operation (dashed boxes).

Special MPI functions for “ring shifts”

```
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)
```

```
int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype,
                       int dest, int sendtag, int source, int recvtag,
                       MPI_Comm comm, MPI_Status *status)
```

Both routines use blocking communication, are guaranteed to not be subject to the deadlock effects that may occur with separate send and receive.

Collective communication in MPI

Recall the numerical integration example: summing up the partial sums is a *reduction* operation.

MPI has mechanisms that make reductions much simpler and in most cases more efficient than looping over all ranks and collecting results.

Since a reduction is a procedure involves all ranks in a communicator, it belongs to the so-called **collective** communication operations in MPI.

A collective MPI routine must be called by all ranks in a communicator.

MPI_Barrier

MPI_Barrier is the simplest collective in MPI, it does not actually perform any real data transfer.

```
int MPI_Barrier(MPI_Comm comm)
```

The barrier *synchronizes* the members of the communicator, i.e., all processes must call it before they are allowed to return to the user code.

Don't over-use MPI_Barrier! There are other MPI routines that allow for implicit or explicit synchronization with finer control.

MPI_Bcast

```
int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root,
               MPI_Comm comm )
```

MPI_Bcast sends a message from one process (the "root") to all others in the communicator.

The **buffer** argument to MPI_Bcast() is a send buffer on the root and a receive buffer on any other process.

Depiction of MPI_Bcast

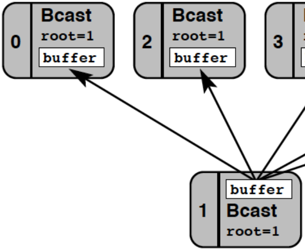


Figure 9.3: An MPI broadcast: The “root” process (rank 1 in this example) sends the same message to all others. Every rank in the communicator must call `MPI_Bcast()` with the same `root` argument.

MPI_Gather & MPI_Scatter

Examples of more advanced MPI collective calls that are concerned with global data distribution:

- MPI_Gather() collects the send buffer contents of all processes and concatenates them in rank order into the receive buffer of the root process
- MPI_Scatter() does the reverse, distributing equal-sized chunks of the root's send buffer
- MPI_Gatherv() and MPI_Scatterv() support arbitrary per-rank chunk sizes

MPI_Gather & MPI_Scatter syntax

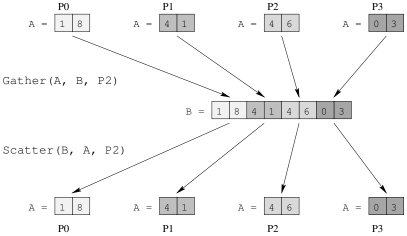
```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
               void *recvbuf, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```

```
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
               void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
               MPI_Comm comm)
```

The syntax of MPI_Gatherv() and MPI_Scatterv() is more complex, not listed here.

Depiction of MPI_Gather & MPI_Scatter

(a) Equal-Size Gather and Scatter Operations



P0 A = 1 8

P1 A = 4 1

P2 A = 4 6

P3 A = 0 3

Gather(A, B, P2)

B = 1 8 4 1 4 6 0 3

Scatter(B, A, P2)

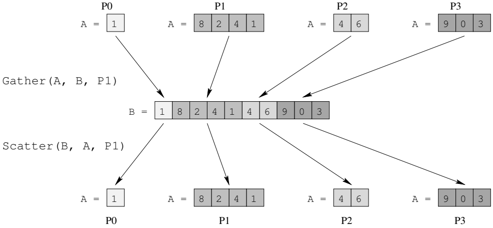
A = 1 8 P0

A = 4 1 P1

A = 4 6 P2

A = 0 3 P3

(b) Unequal-Size Gather and Scatter Operations



P0 A = 1

P1 A = 8 2 4 1

P2 A = 4 6

P3 A = 9 0 3

Gather(A, B, P1)

B = 1 8 2 4 1 4 6 9 0 3

Scatter(B, A, P1)

A = 1 P0

A = 8 2 4 1 P1

A = 4 6 P2

A = 9 0 3 P3

MPI_Reduce

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

MPI_Reduce() combines the contents of the sendbuf array on all processes, element-wise, using an operator encoded by the op argument, and stores the result in recvbuf on root

Examples of predefined operators: MPI_MAX, MPI_MIN, MPI_SUM and MPI_PROD

Depiction of MPI_Reduce

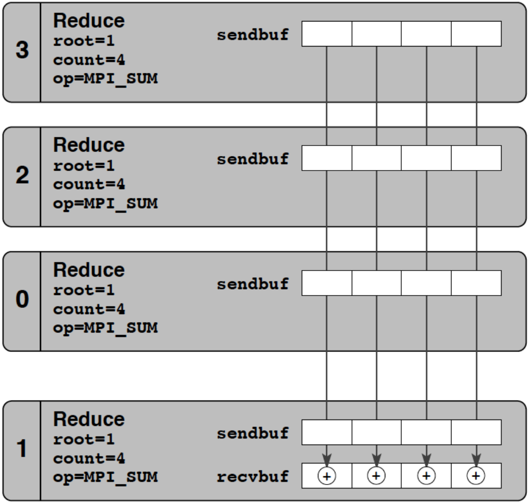


Figure 9.4: A reduction on an array of length `count` (a sum in this example) is performed by `MPI_Reduce()`. Every process must provide a send buffer. The receive buffer argument is only used on the root process. The local copy on root can be prevented by specifying `MPI_IN_PLACE` instead of a send buffer address.

Rewrite of the “numerical integration” example

```
int rank, size;
double a=0.0, b=1.0, mya, myb, psum, res=0.;
MPI_Status status;

MPI_Init (&nargs, &args);
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);

mya = a + rank*(b-a)/size;
myb = mya + (b-a)/size;
psum = integrate(mya,myb); // Each integrates over its "subdomain"

MPI_Reduce(&psum,&res,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
if (rank==0)
  printf("Result: %g\n", res);

MPI_Finalize();
```

MPI_Allreduce

```
int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,
                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

The result of the reduction operation (`recvbuf`) is available on all MPI ranks.

Nonblocking point-to-point communication

Point-to-point communication can be performed with nonblocking semantics.

- A nonblocking point-to-point call merely **initiates** a message transmission and returns very quickly to the user code.
- In an efficient implementation, waiting for data to arrive and the actual data transfer occur in the background, leaving resources free for computation.
- In other words, nonblocking MPI is a way in which communication may be overlapped with computation if implemented efficiently.
- The message buffer **must not be used** as long as the user program has not been notified that it is safe to do so (which can be checked by suitable MPI calls).
- Nonblocking and blocking MPI calls are mutually compatible, i.e., a message sent via a blocking send can be matched by a nonblocking receive.

MPI_Isend

```
int MPI_Isend(const void *buf, int count, MPI_Datatype datatype,
              int dest, int tag,
              MPI_Comm comm, MPI_Request *request)
```

- As opposed to the blocking send (MPI_Send()), MPI_Isend() has an additional output argument, the *request* handle (of type `struct MPI_Request`).
- It serves as an identifier by which the program can later refer to the “pending” communication request.

MPI_Irecv

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag,
              MPI_Comm comm, MPI_Request * request)
```

- The `status` object known from MPI_Recv() is **not** needed for MPI_Irecv.

MPI_Test & MPI_Wait

Checking a pending communication for completion can be done via the MPI_Test() and MPI_Wait() functions. The former only tests for completion and returns a flag, while the latter blocks until the buffer can be used.

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

The status object contains useful information only if the pending communication is a completed receive (i.e., in the case of MPI_Test() the value of flag must be true).

MPI_Waitall

In the case of multiple non-blocking communication operations (multiple requests pending), it is more convenient to use the MPI_Waitall function:

```
int MPI_Waitall(int count, MPI_Request array_of_requests[],
                MPI_Status array_of_statuses[])
```

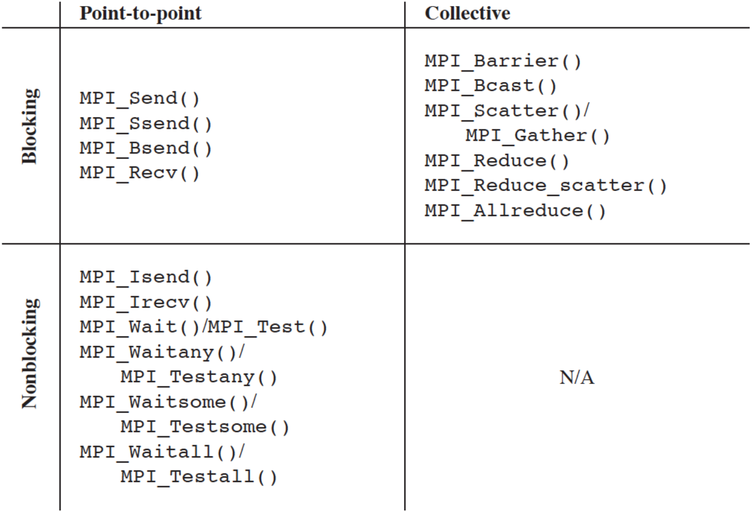
The “numerical integration” example, yet again

```
// ...
double *tmp;
MPI_Request *request_array;
MPI_Status *status_array;

if (rank==0) {
  // allocate arrays of tmp, request_array & status_array...
  for (i=1; i<size; i++)
    MPI_Irecv(&tmp[i-1],1,MPI_DOUBLE,i,0,MPI_COMM_WORLD,&request_array[i-1]);
}

mya = a + rank*(b-a)/size;
myb = mya + (b-a)/size;
psum = integrate(mya,myb); // Each integrates over its "subdomain"

if (rank==0) {
  double res = psum;
  MPI_Waitall (size-1,request_array,status_array);
  for (i=1; i<size; i++)
    res += tmp[i-1];
  printf("Result: %g\n", res);
}
else {
  MPI_Send(&psum,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
}
```

Benefits of nonblocking communication

- Nonblocking communication provides an obvious way to overlap communication, i.e., overhead, with useful work.
- The possible performance advantage, however, depends on many factors, and may even be nonexistent.
- But even if there is no real overlap, multiple outstanding nonblocking requests may improve performance because the MPI library can decide which of them gets serviced first.
- Nonblocking communication helps to avoid deadlock.

A short summary



	Point-to-point	Collective

Blocking	MPI_Send()
	MPI_Ssend()
	MPI_Bsend()
	MPI_Recv()
		MPI_Barrier()
		MPI_Bcast()
		MPI_Scatter()/ MPI_Gather()
		MPI_Reduce()
		MPI_Reduce_scatter()
		MPI_Allreduce()

Nonblocking	MPI_Isend()
	MPI_Irecv()
	MPI_Wait()/MPI_Test()
	MPI_Waitany()/ MPI_Testany()
	MPI_Waitsome()/ MPI_Testsome()
	MPI_Waitall()/ MPI_Testall()
		N/A

Table 9.3: MPI's communication modes and a nonexhaustive overview of the corresponding subroutines.

Virtual topologies

- MPI suits very well for implementing domain decomposition (Section 5.2.1) on distributed-memory parallel computers.
- However, setting up the process grid and keeping track of which ranks have to exchange halo data is nontrivial.
- MPI contains some functionality to support this recurring task in the form of *virtual topologies*.
- These provide a convenient process naming scheme, which fits the required communication pattern.

Cartesian topologies

Example: a 2D global Cartesian mesh of size 3000 × 4000. Suppose we want to use $3 \times 4 = 12$ MPI processes to divide the global mesh, each holding a piece of 1000 × 1000.

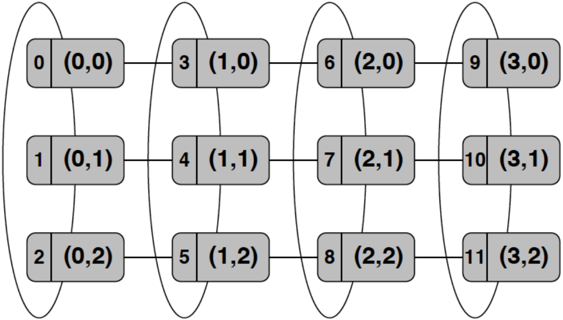


Figure 9.6: Two-dimensional Cartesian topology: 12 processes form a 3×4 grid, which is periodic in the second dimension but not in the first. The mapping between MPI ranks and Cartesian coordinates is shown.

Cartesian topologies (2)

- As shown in the preceding figure, each process can either be identified by its rank or its Cartesian coordinates.
- Each process has a number of neighbors, which depends on the grid's dimensionality. (In our example, the number of dimensions is two, which leads to at most four neighbors per process.)
- MPI can help with establishing the mapping between ranks and Cartesian coordinates in the process grid.

MPI_Cart_create

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],
                    const int periods[], int reorder, MPI_Comm * comm_cart)
```

- A new, “Cartesian” communicator comm_cart is generated, which can be used later to refer to the topology.
- The periods array specifies which Cartesian directions are periodic, and the reorder parameter allows, if true, for rank reordering so that the rank of a process in communicators comm_old and comm_cart may differ.
- Here, MPI merely keeps track of the topology information.

MPI_Cart_coords & MPI_Cart_rank

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])
```

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

- These are two "service" functions responsible for the translation between Cartesian process coordinates and an MPI rank.
- MPI_Cart_coords() calculates the Cartesian coordinates for a given rank.
- The reverse mapping, i.e., from Cartesian coordinates to an MPI rank, is performed by MPI_Cart_rank().

Example of 2D Cartesian grid

```
int rank, size;
MPI_Comm comm;
int dim[2], period[2], reorder;
int coord[2], id;
// ....

dim[0]=4; dim[1]=3;
period[0]=0; period[1]=1;
reorder=1;

MPI_Cart_create(MPI_COMM_WORLD, 2, dim, period, reorder, &comm);

if (rank == 5) {
    MPI_Cart_coords(comm, rank, 2, coord);
    printf("Rank %d coordinates are %d %d\n", rank, coord[0], coord[1]);
}

if(rank==0) {
    coord[0]=3; coord[1]=1;
    MPI_Cart_rank(comm, coord, &id);
    printf("The processor at position (%d, %d) has rank %d\n", coord[0], coord[1], id);
}
```

MPI_Cart_shift

A regular task with domain decomposition is to find out who the next neighbors of a certain process are along a certain Cartesian dimension.

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source,
                   int *rank_dest)
```

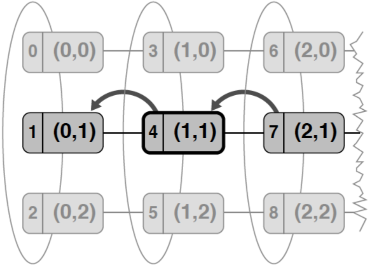


Figure 9.7: Example for the result of `MPI_Cart_shift()` on a part of the Cartesian topology from Figure 9.6. Executed by rank 4 with `direction`=0 and `disp`=−1, the function returns `rank_source`=7 and `rank_dest`=1.