EUROCC - National Competence Centres in the framework of EuroHPC

EuroHPC-04-2019: HPC Competence Centres

Training: Parallel Programming with MPI

Dr Hristo Iliev

15.02.2021

Quick Introduction to MPI

MPI (Message Passing Interface) is the *de facto* standard paradigm for programming computational systems with distributed memory. It is an open specification that replaces various incompatible vendor communication interfaces and so ensures source code portability to various distributed memory platforms. The MPI specification is being maintained and further developed by the MPI Forum, a not-for-profit organisation whose members are various HPC users, software and hardware vendors. The text of the specification can be downloaded for free from the web site of the forum or ordered in book form for the price of printing the copy. MPI is a language- and platform-neutral set of primitives for data exchange and parallel I/O. It also includes two standard language bindings (concrete specifications of how each primitive should be invoked in a given language) for C and Fortran. There are other, non-standard bindings such as Boost.MPI for C++ and mpi4py for Python, which are not part of the specification and therefore provide features that form sub- or supersets of the specification. For example, **Boost.MPI** has the ability to directly communicate complex C++ objects, which saves the programmer a lot of hassle, but it doesn't provide access to the parallel I/O and the single-sided operations, for which one has to still use the C biding. Unlike many other technical specifications, the MPI standard allows itself to be read by mere mortals and can be used as a reference.

There are many implementations of MPI, some of which are available for free and with open source code (e.g. MPICH, Open MPI, etc.), while others are available under commercial licences (e.g., Intel MPI, Microsoft HPC Pack) or available only on certain HPC systems (e.g., Cray MPI). While the different implementations are usually binary incompatible with one another, often even between different versions of the same implementation, they provide source-level compatibility. This enables a development scenario, in which one develops and debugs a given MPI program on a personal computer using a freely available universal MPI implementation such as Open MPI, tests the scalability of the program with an optimised MPI implementation on a small to medium sized compute cluster, and finally performs production runs on a large supercomputer using the vendor-provided MPI implementation.

Unlike some other techniques for parallel programming, such as OpenMP and CUDA, MPI is not a language extension and does not require special support from the compiler. Rather, it is an application programming interface (API) and is often implemented as a set of external libraries and supporting programs, usually called *MPI runtime*.

Hello, MPI!

The simplest MPI programs in C and Fortran look as follows. Some interesting points in the source code are marked and detailed further down.

```c
#include <stdio.h>
#include <mpi.h>                          // (1)

int main (void)
{
  int rank, size;

  MPI_Init(NULL, NULL);                   // (2)

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);  // (3)
  MPI_Comm_size(MPI_COMM_WORLD, &size);  // (4)

  printf("Hello MPI world from rank %d of %d\n", rank, size);

  MPI_Finalize();                         // (5)

  return 0;
}
```

```fortran
program hello_mpi
  use mpi                                          ! (1)
  implicit none
  integer :: rank, size, ierr

  call MPI_INIT(ierr)                              ! (2)

  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)  ! (3)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)  ! (4)

  print *, "Hello MPI world from rank ", rank, " of ", size

  call MPI_FINALIZE(ierr)                          ! (5)
end program hello_mpi
```

Both programs share a common structure and differ from the traditional "Hello World" examples in five key places:

1. The header file `mpi.h` gets included (in C) or the `mpi` module is used (in Fortran), which makes available all the subroutines and named constants provided by MPI.
2. Before doing anything related to MPI, each program **must** first call `MPI_Init`, which initialises the MPI runtime. Only a very limited number of MPI subroutines can be called before `MPI_Init`.
3. Query for the ID of the current rank in the world communicator. More on that in the next section.
4. Query for the size of the world communicator. More on that in the next section.
5. Each MPI program **must** call `MPI_Finalize`, which completes any pending operation and signals the MPI environment that it's free to terminate. Failure to call `MPI_Finalize` may have unpredictable results depending on the implementation. Only a very limited number of MPI subroutines can be called after `MPI_Finalize`.

Compiling MPI Programs

Conceptually, MPI programs are compiled no differently than any other program. First, the source code is transformed to an object code by a compiler, then the necessary libraries are linked in to create one or more executable files. At any of those steps, it is usually necessary to provide additional information to the compiler, such as the location of the MPI header files, the location of the MPI libraries, and the names and locations of various system and other libraries that MPI is dependent on. For example, with Open MPI compiling and linking a C source file to an executable looks like:

```
$ icc -o hello_world hello_world.c \
      -I/opt/openmpi/1.6.5/intel-13.1/include \
      -pthread -L/opt/openmpi/1.6.5/intel-13.1/lib \
      -lmpi -ldl -lm \
      -Wl,--export-dynamic -lrt -lnsl -lutil
```

Of all those options, only the first line contains actual input from the developer and everything else is boilerplate required by the MPI library. To simplify and unify that process, most MPI vendors provide special *compiler wrappers* that pass their command-line options together with any additionally needed ones to the system compiler and linker. Common names for these wrappers are:

- `mpicc` -- wrapper around the system C compiler
- `mpic++` -- wrapper around the system C++ compiler

- `mpicxx` or `mpiCC` -- alternative names for `mpic++`
- `mpif90` -- wrapper around the system Fortran 90+ compiler
- `mpifort` -- generic wrapper around the system Fortran compiler(s)

The actual names may vary between the implementations and one should consult the user manual for more information. Using the compiler wrapper `mpicc`, the example above simplifies to:

```
$ mpicc -o hello_world hello_world.c
```

Model of Execution of MPI

MPI programs execute as a number of entities, called *ranks*. The standard does not specify what exactly is a rank, but with most existing implementations each rank is a separate operating system (OS) process. The basic presumption is that ranks do not directly share memory and instead pass it around when needed in the form of messages. This resembles the model of virtual address space isolation provided by most modern OSes, in which model the different processes share no memory and need to use special OS primitives for data exchange.

In the most common case, and this is a presumption (but not a requirement!) that the MPI API is structured around, all ranks are instances of the same program, e.g., multiple processes started from the same executable file, working on different parts of the problem data. This is known as *SPMD (Single Program Multiple Data)* and is similar to the model of other parallel programming paradigms such as OpenMP and CUDA. SPMD is not a hard requirement in MPI and many implementations allow for ranks from several executable files to be mixed in a single MPI program, also known as *MPMD (Multiple Programs Multiple Data.)* Understanding the SPMD model of writing single-source distributed applications is the most important aspect when learning MPI.

Ranks are given a unique 0-based numeric ID, also called a *rank*, which allows the program to distinguish the copy that it is currently executing as and hence perform a different computation accordingly. Without such a unique ID, it will not be possible to distinguish between the copies and all of them will simply perform the same computation, which is rarely useful. MPI ranks form logical *groups* and the ID of a rank is actually the index of that rank in the corresponding group. MPI groups are quite ephemeral objects and one usually deals with *communicators* instead, which are contexts built around each group and are where communication in MPI takes place. Each rank can be a member of several communicators at the same time and have a different numeric ID in the group corresponding to each communicator. Therefore, in order to uniquely identify a rank, one needs to specify **both the rank ID and the communicator that it's valid in**.

Communicators are immutable objects and can only be created by splitting or joining groups of ranks from existing communicators. There are two pre-existing communicators that are created automatically when MPI is initialised:

* `MPI_COMM_WORLD` -- known as the *world communicator*, it comprises all MPI ranks that are initially part of the MPI program;
* `MPI_COMM_SELF` -- contains only the current rank; useful when spawning child MPI programs (MPI process control is outside the scope of this course.)

MPI ranks obtain their ID in a specific communicator using `MPI_Comm_rank`:

```
int rank;

ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
integer :: rank

call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
```

It is often necessary to know how many ranks are there in a given communicator. The query for that is `MPI_Comm_size`:

```
int size;

ierr = MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
integer :: size

call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
```

Running MPI Programs

Unlike traditional programs where running an executable boils down to typing its name or double-clicking its visual representation in a graphical user interface, running an MPI program is a much more involved process. MPI targets distributed architectures, which brings a set of additional questions.

- **Where is the program going to run?** Most distributed systems such as supercomputers or clusters are shared resources and one usually receives access to

dynamically allocated portions. It is necessary to obtain information about the parts that have been granted access to.
- **How does the distribution of executable file(s) happen?** Depending on the system organisation, it may be necessary to distribute one or more executable files to remote nodes. This is often avoided by using shared file systems.
- **How to run executable files on more than one host?** When launching an MPI program that spans more than one host, which is the typical scenario of using MPI, it is necessary to launch and control the execution of processes on remote hosts. It is also necessary to supply each process with information on how to reach the other processes.

Those problems are usually solved by the MPI implementation by providing a special *program launcher*, which takes care of discovering the allocated hosts, launching MPI ranks on local and remote hosts, controlling them, and performing input-output redirection as necessary. The MPI specification deliberately does not attempt to standardise the process as it depends heavily on the system architecture and the system software. Nevertheless, the specification recommends that the launcher, if any, should be named `mpiexec` and prescribes a tiny set of universal command-line options. Some implementations provide the launcher under different names, among those `mpirun` -- the historic name of the MPI launcher in many early implementations. Some distributed resource managers, for example SLURM, have the ability to directly launch MPI programs. Others provide mechanisms for the launcher from the MPI implementation to hook into the job execution process.

Point-to-point Communication

The most fundamental operation in MPI is passing a message from rank A to rank B, also known as point-to-point communication. Message passing in MPI is based on *explicit agreement* on both sides of the communication, which means that:

- rank A (hereby the sender) signals MPI that it wants to send a message to rank B in communicator *comm* by *posting a send* operation with B as destination, and
- rank B (hereby the receiver) signals MPI that it wants to receive a message from rank A in communicator *comm* by *posting a receive* operation with A as source.

Only when both sides post two matching operations does the communication complete logically.

The standard MPI send operation for sending an array of 100 double precision floating-point numbers looks like this in C:

```
ierr = MPI_Send(
    buf,               // (1) buffer location
    100,               // (2) number of elements in the buffer
    MPI_DOUBLE,        // (3) data type
    rank_of_B,         // (4) rank of the receiver
    0,                 // (5) message tag
    MPI_COMM_WORLD     // (6) communicator
);
```

The first triplet of arguments (1-3) occurs in many MPI calls and it specifies the location of the data, its size, and its data type. Most MPI operations are array-oriented and they allow for multiple elements of the same type of data to be sent at once. Since MPI is a library and not an extension of the language, it doesn't automatically know the type of the data in the buffer, therefore one has to provide that information explicitly by specifying the data type in the third argument. `MPI_DOUBLE` is a special predefined MPI data type *handle* (not a C type!) that corresponds to the C type `double`. There are other predefined data types for almost any primitive C type:

MPI data type handle	**C type**
`MPI_CHAR`	`char`
`MPI_SHORT`	`short`
`MPI_INT`	`int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_BYTE`	one byte (w/o conversion)

An exhaustive list of predefined data types can be found in the MPI specification. Data types allow MPI to treat in-memory values in an architecture-independent way and allow for heterogeneous computing by converting data on the fly while communicating. `MPI_BYTE` is a special type that is used to send data without any conversion.

The second triplet of arguments (4-6) specifies the receiver of the message, a message *tag*, and the communicator where the operation should take place. The tag is a simple integer value that is attached to each message and can be matched against by the receiver.

In Fortran, the send operation looks identically except for the additional `ierr` argument that receives the error code:

```
call MPI_Send(            &
    buf,                  & ! (1) buffer location
    100,                  & ! (2) number of elements in the buffer
    MPI_DOUBLE_PRECISION, & ! (3) data type
    rank_of_B,            & ! (4) rank of the receiver
    0,                    & ! (5) message tag
    MPI_COMM_WORLD,       & ! (6) communicator
    ierr                  & ! (7) error code (output)
)
```

Just like in C, one has to explicitly tell MPI what type of data is being sent by providing the send operation with an MPI datatype handle. There are many predefined data types for the most common Fortran types:

MPI data type handle	**Fortran type**
`MPI_CHARACTER`	`character`
`MPI_INTEGER`	`integer`
`MPI_REAL`	`real`
`MPI_REAL8`	`real*8, real(kind=8)`
`MPI_DOUBLE_PRECISION`	`double precision`
`MPI_BYTE`	one byte (w/o conversion)

There are special provisions in MPI for mapping Fortran kinds to MPI datatypes, which are outside the scope of this course. For more information, see the MPI specification.

The standard MPI receive operation looks very similar to the send operation:

```
MPI_Status status;

ierr = MPI_Recv(
    buf,                // (1) buffer location
    100,                // (2) buffer capacity in elements
    MPI_DOUBLE,         // (3) data type
    rank_of_A,          // (4) rank of the sender
    0,                  // (5) message tag
    MPI_COMM_WORLD,     // (6) communicator
    &status             // (7) status object (output)
);
```

```
integer, dimension(MPI_STATUS_SIZE) :: status

call MPI_Recv(                  &
    buf,                        & ! (1) buffer location
    100,                        & ! (2) buffer capacity
    MPI_DOUBLE_PRECISION,       & ! (3) data type
    rank_of_A,                  & ! (4) rank of the sender
    0,                          & ! (5) message tag
    MPI_COMM_WORLD,             & ! (6) communicator
    status,                     & ! (7) status object
    ierr                        & ! (8) error code
)
```

The structure of the arguments follows the same pattern. The first triplet (1-3) specifies the location of the buffer, its capacity, and the data type. The capacity specifies the **maximum size** of the message that can be received. If the message contains less elements, part of the buffer simply will not be filled. If the message contains more elements than the buffer capacity, the operation will fail with a message truncation error.

The second triplet (4-6) specifies the rank of the sender that we expect a message from, the tag of the message, and the communicator. Those arguments collectively form a filter that matches against the incoming messages. Only a message that matches all three fields will be received. It is possible to specify *wildcards* for any of the sender and the tag by passing in the following special values:

* `MPI_ANY_SOURCE` - special sender rank that matches any sender;

* `MPI_ANY_TAG` - special message tag that matches any tag value.

There are no wildcard values for the communicator argument.

The status argument provides the location of an instance of `MPI_Status`, a structure with several public fields in C or an integer array of size `MPI_STATUS_SIZE` in Fortran, which provides information about the received message:

* count of elements received from the message;
* rank of the sender -- `status.MPI_SOURCE` in C and `status(MPI_SOURCE)` in Fortran;
* message tag -- `status.MPI_TAG` in C and `status(MPI_TAG)` in Fortran.

The last two are useful when one specifies a wildcard for the sender's rank and/or the message tag, but also needs to know the actual values. The actual count of elements in the message can be obtained with `MPI_Get_count`:

```
int count;

ierr = MPI_Get_count(&status, MPI_DOUBLE, &count);
```

```
integer :: count

call MPI_Get_count(status, MPI_DOUBLE_PRECISION, count, ierr)
```

If the amount of bytes in the message is not enough to form an integral number of elements of the provided data type, `MPI_Get_count` sets `count` to `MPI_UNDEFINED`. This happens when there is a data type mismatch on both sides of the communication. For example, if one sends an `MPI_INT` element and tries to receive it as `MPI_DOUBLE`.

N.B. MPI does not require that conforming implementations perform explicit type checking on the data in the received message and most don't typecheck for better performance. It is therefore up to the programmer to ensure that messages are interpreted as intended. Although on platforms with 4-byte integers and 8-byte double precision floats it is technically possible to send a message containing two `MPI_INT`s and receive it as a single `MPI_DOUBLE`, this is not a correct MPI operation. Some implementations may provide special debug modes with type checking, and also special MPI correctness checking tools exist, such as MUST.

Although the predefined MPI datatypes cover only the primitive types of the corresponding language, one can build out of them compound data types and so it is possible to send and receive data with complex memory layout.

Send Modes

MPI provides several modes of sending messages. All modes relate to the completion of the send operation in relation to the matching receive operation. In the *synchronous mode*, the send operation does not complete before a matching receive has been posted and the message reception has started (but not necessarily completed.) Because the sender blocks until the receiver is ready to receive the message, this mode synchronises the execution of both ranks. In the *buffered mode*, the message is first copied to a local user-provided buffer and then delivered later on when the receiver is ready to start receiving it. In this mode, the message is delivered asynchronously and it doesn't have the synchronising effect of the synchronous mode. The third mode, known as the *standard mode*, is a combination of both modes that aims at delivering the best possible performance. For small enough messages, the standard mode is usually buffered, using a system buffer instead of a user-provided one, while for larger messages it is synchronous. The threshold for switching between the two modes is implementation-dependent.

The `MPI_Send` operation performs the standard mode send. There are specific calls for the other two modes. `MPI_Ssend` performs send in synchronous mode while `MPI_Bsend` performs send in buffered mode. All three functions take the exact same type and number of arguments and only differ in the send mode. The buffered mode is tricky since it requires that the user must first attach a large enough buffer to hold the messages and some metadata overhead. It is rarely needed in practice except in some special cases, therefore it is not considered here. Interested users are advised to consult the MPI specification.

Semantics of Point-to-Point Message Passing

Unlike some networking APIs, MPI sends and receives messages as atomic units of data, i.e. there is exactly one message sent with each invocation of `MPI_Send` and exactly one message is received with each invocation of `MPI_Recv`. It is not possible to send one large message and then receive it with a couple of small-buffer receive operations. Conversely, it is not possible to receive a couple of small messages together using a single large-buffer receive. Therefore, the number of send operations must exactly match the number of receive operations, both globally and between any pair of ranks.

MPI messages are non-overtaking, i.e. they are always received in the same order in which they were sent. This only applies to messages sent between a pair of ranks and in a specific communicator. In other words, MPI works a lot like a message queueing system. The receiver always sees the oldest message that matches the specified filter. No ordering guarantee is given for messages coming from different ranks or in different communicators.

N.B. MPI was originally designed to work with single-threaded ranks since distributed-memory parallelism was an alternative to shared-memory parallelism. With the two kinds getting more and more mixed together, some abilities to handle threading were

introduced in MPI. Notwithstanding, no guarantee is given for stable message ordering when several threads are calling `MPI_Send` or `MPI_Recv` simultaneously.

The standard send mode guarantees that once the MPI call returns, the message is either buffered locally or entirely in transit. A correct MPI program must not rely on it having one or another behaviour, i.e. one must not assume that the standard send is always buffering small messages or always synchronising messages of a certain size. This is important in order to avoid deadlocks.

Imagine the common operation of two ranks exchanging some data. Rank A sends some data to rank B and receives some data from it. Conversely, rank B sends some data to rank A and receives some data from it. One may be tempted to implement it as a sequence of `MPI_Send` and `MPI_Recv` as shown below:

```
int other_rank = get_other_rank();

MPI_Send(&to_send, 1, MPI_INT, other_rank, 0, MPI_COMM_WORLD);
MPI_Recv(&received, 1, MPI_INT, other_rank, 0, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
```

Implemented that way, the program may or may not work, depending on whether `MPI_Send` is buffering small messages or not. If the messages are buffered, then both ranks will reach the `MPI_Recv` calls, which will progress the sending of the buffered messages while waiting to receive. If the standard send does not buffer small messages, both sends will block waiting for the matching receive calls, which will never be reached. One way to tackle that problem is to swap the order of send and receive in one of the ranks:

`MPI_Send(&to_send, ...);` `MPI_Recv(&received, ...);`	`MPI_Recv(&received, ...);` `MPI_Send(&to_send, ...);`

While it works, it is not a scalable solution, especially if the number of ranks can vary. Another solution is to use the non-blocking operations discussed later on. The best solution is to use `MPI_Sendrecv`, which combines the functionality of `MPI_Send` and `MPI_Recv` in a single call and guarantees that it will not deadlock as long as each send is matched by a receive:

```
int other_rank = get_other_rank();

MPI_Sendrecv(&to_send, 1, MPI_INT, other_rank, 0,
             &received, 1, MPI_INT, other_rank, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
integer :: other_rank

other_rank = get_other_rank()

call MPI_Sendrecv(to_send, 1, MPI_INTEGER, other_rank, 0, &
                  received, 1, MPI_INTEGER, other_rank, 0, &
                  MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)
```

An easy way to test the correctness of an MPI program with respect to the send operation is to replace each occurrence of `MPI_Send` with `MPI_Ssend` -- a correctly written program will run slower but will not deadlock.

MPI Primer

The following primer is a program that sums the numbers from 1 to N (here N is set to 1000) and helps better grasp the way SPMD and MPI in particular work:

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

#define N 1000

int main (int argc, char **argv)
{
    MPI_Init(&argc, &argv);

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

    int first = 1 + rank * N / size;
    int last = (rank + 1) * N / size;
    int partial_sum = 0;
    for (int i = first; i <= last; i++)
        partial_sum += i;

    if (rank == 0)
    {
        int total_sum = partial_sum;
```

```
        for (int i = 1; i < size; i++)
        {
            MPI_Recv(&partial_sum, 1, MPI_INT, i, 0,
                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            total_sum += partial_sum;
        }
        printf("The sum from 1 to %d is %d\n", N, total_sum);
    }
    else
        MPI_Send(&partial_sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);

    MPI_Finalize();
    return 0;
}
```

After initialisation of MPI, the program finds out its rank in the world communication and the total number of ranks:

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

It then uses that information to compute a range of numbers to sum up:

```
int first = 1 + rank * N / size;
int last = (rank + 1) * N / size;
```

The numbers in the range are summed up in order to obtain a partial sum:

```
for (int i = first; i <= last; i++)
    partial_sum += i;
```

Although each MPI rank is created from the same executable file and therefore executes the exact same code, the value of `rank` returned by `MPI_Comm_rank` will be different in each rank and so will be the computed values of `first` and `last`. Thus, each rank will compute the partial sum of a different contiguous sub-range of numbers with the totality of ranks covering the entire range from 1 to N.

Once the partial sums are ready, ranks split in two groups that perform different tasks, an approach known as *functional parallelism*. Rank 0 begins a loop in which it collects the partial sums computed by itself and all the other ranks, adding the received values together into `total_sum`:

```
if (rank == 0)
{
    int total_sum = partial_sum;
    for (int i = 1; i < size; i++)
    {
        MPI_Recv(&partial_sum, 1, MPI_INT, i, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        total_sum += partial_sum;
    }
    printf("The sum from 1 to %d is %d\n", N, total_sum);
}
```

All other ranks just send their partial sum to rank 0:

```
else
    MPI_Send(&partial_sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
```

The accumulation of partial sums distributed among the ranks of the MPI program is such a common operation that MPI provides a special generic global reduction operation, which will be the subject of a later section.

For completeness, the same program in Fortran:

```
program sum_to_n
  use mpi
  implicit none

  integer, parameter :: N = 1000
  integer :: rank, size, ierr
  integer :: first, last, partial_sum, total_sum, i

  call MPI_Init(ierr)
```

```
  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
  call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)

  first = 1 + rank * N / size
  last = (rank + 1) * N / size
  partial_sum = 0
  do i = first, last
    partial_sum = partial_sum + i
  end do

  if (rank == 0) then
    total_sum = partial_sum
    do i = 1, size-1
      call MPI_Recv(partial_sum, 1, MPI_INTEGER, i, 0, &
                    MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)
      total_sum = total_sum + partial_sum
    end do
    print *, "The sum from 1 to ", N, " is ", total_sum
  else
    call MPI_Send(partial_sum, 1, MPI_INTEGER, 0, 0, &
                  MPI_COMM_WORLD, ierr)
  end if

  call MPI_Finalize(ierr)
end program
```

Error Handling in MPI

Most MPI calls in C are functions that return an integer error code while the same calls in Fortran are subroutines that return the error code in their last output argument. Upon success, the error code will be MPI_SUCCESS, otherwise it will have an implementation-specific error value. MPI specifies error handling in general and leaves much of the specifics to the implementations.

In reality, most MPI users never have to bother with handling errors and code such as the following one is often superficial:

```
if (MPI_Send(...) != MPI_SUCCESS)
{
   // handle error
}
```

The reason for that is the error handling policy in MPI. Before returning, all MPI operations call an error handler that checks the error code and either aborts the application or allows the call to return with an error code. The default error handler for communication calls is of the former type and the entire MPI program simply aborts upon error. Therefore, the returned error code is always `MPI_SUCCESS` and explicitly checking whether that is indeed the case is redundant and makes the code less readable. If one wants to instead receive the error code and continue the execution, the error handler must be replaced with a non-aborting one. This is an advanced topic that goes beyond the scope of the course. Suffice it to say that most MPI implementations are not fault-tolerant and cannot recover from communication errors. Returning after an error in that case is simply meant to give the application a chance to crash in a more graceful manner.

Note: Although the integer error code in Fortran is practically never used, one of the most common mistakes is to omit the argument which leads to all sorts of compilation (best case) or runtime (worst case) errors.

Probing for Messages

It is sometimes desirable to know more about a message before receiving it. For example, one may want to dynamically allocate a buffer that exactly accommodates an incoming message instead of using (and possibly underutilising) a large static buffer. For that purpose, MPI provides the `MPI_Probe` operation which waits until a matching message becomes available without actually receiving it. Once matched, the message may be examined and then received thanks to the non-overtaking semantics of message passing.

```
MPI_Status status;

MPI_Probe(source, tag, MPI_COMM_WORLD, &status);
```

```
integer, dimension(MPI_STATUS_SIZE) :: status

call MPI_Probe(source, tag, MPI_COMM_WORLD, status, ierr)
```

The arguments of `MPI_Probe` are exactly the same four arguments that specify the message source, the tag, the communicator, and the output status object in `MPI_Recv`.

Wildcards are accepted for the first two. A typical use of `MPI_Probe` is for dynamic allocation of receive buffers when messages of size not known in advance are handled:

```
MPI_Status status;
int count;
double *buf;

MPI_Probe(MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
MPI_Get_count(&status, MPI_DOUBLE, &count);
buf = malloc(sizeof *buf * count);
MPI_Recv(buf, count, MPI_DOUBLE, status.MPI_SOURCE, 0,
         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Here, besides peeking the message size in order to allocate a big enough buffer, the code demonstrates the use of wildcards in the probe-then-receive case. The receive operation uses the message source as reported in the status object by `MPI_Probe`. This is because ordering is guaranteed only for messages sent between the same pair of ranks. If the source was specified as `MPI_ANY_SOUCE` in `MPI_Recv` too, it is possible that instead a different message sent by a different source rank is received.

Collective Communication

In section MPI Primer, we saw an example of an operation in which all ranks participate and the point of which was to compute the sum of several values, distributed among all ranks:

```
if (rank == 0)
{
    int total_sum = partial_sum;
    for (int i = 1; i < size; i++)
    {
        MPI_Recv(&partial_sum, 1, MPI_INT, i, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        total_sum += partial_sum;
    }
}
else
    MPI_Send(&partial_sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
```

This is an example of a more general class of MPI operations where all ranks in a given communicator participate at the same time. Such operations are known as *collective communication operations*, or *collectives* for short. Such operations occur repeatedly in distributed algorithms, which is why MPI provides implementations for the most common of them.

The above code is an example of an operation known as *global reduction*. The same piece of code can be rewritten simply as:

```
MPI_Reduce(
    &partial_sum,    // (1) send buffer
    &total_sum,      // (2) receive buffer
    1, MPI_INT,      // (3) count and datatype of elements
    MPI_SUM,         // (4) reduction operation
    0,               // (5) root rank
    MPI_COMM_WORLD   // (6) communicator
);
```

```
call MPI_Reduce(    &
    partial_sum,    & ! (1) send buffer
    total_sum,      & ! (2) receive buffer
    1, MPI_INTEGER, & ! (3) count and datatype of elements
    MPI_SUM,        & ! (4) reduction operation
    0,              & ! (5) root rank
    MPI_COMM_WORLD, & ! (6) communicator
    ierr)
```

There are two classes of collectives in MPI. In the first class are operations for which the result is placed in a single rank only, called the *root* of the operation. `MPI_Reduce` is one such collective and the root is specified in argument (5) (**N.B.** all ranks **must** specify the exact same value there). The reduction operation specified by its handle (4) is applied element-wise to the data in the send buffers across all ranks in the communicator (6):

$$\textbf{total\_sum[i]} = \textbf{partial\_sum}^{0}\textbf{[i]} + \textbf{partial\_sum}^{1}\textbf{[i]} + \ldots + \textbf{partial\_sum}^{N-1}\textbf{[i]}$$

Here $\textbf{partial\_sum}^{k}\textbf{[i]}$ is the *i*-th element of array *partial_sum* stored in rank *k*. The resulting values are written in `total_sum` in the root rank. In all other ranks the value supplied in the receive buffer argument (2) is ignored.

MPI provides a selection of predefined reduction operations that work with most basic MPI datatypes. Some of them are:

Reduction Op Handle	Operation
MPI_SUM	sum
MPI_PROD	product
MPI_MIN	global min value
MPI_MAX	global max value
MPI_LAND	logical AND
MPI_BAND	bitwise AND

All predefined operations are assumed to be commutative. This allows MPI implementations to provide efficient algorithms for computing the reduced values, but it also means that the order of application of the operation is not guaranteed. This could lead to surprising results when the algebra of the underlying language type is non-commutative. The best known example of such algebra is that of the truncated floating-point numbers used in virtually any modern computing system. One is therefore strongly advised to read the seminal work What Every Computer Scientist Should Know about Floating-Point Arithmetic.

Another often needed operation is the broadcast in which the value stored at a given rank (the root) is copied into all other ranks:

```
MPI_Bcast(&buf, n, MPI_INT, root, MPI_COMM_WORLD);
```

```
call MPI_Bcast(buf, n, MPI_INTEGER, root, MPI_COMM_WORLD, ierr)
```

The effect of calling MPI_Bcast is:

$buf^{k}[i] = buf^{root}[i]$, for all $k \neq root$

In rank `root`, `buf` is an input argument that specifies the location of the value(s) to be broadcasted. In all other ranks, `buf` is an output argument that specifies the location where the values are to be stored. All ranks must specify the same root and the same buffer length.

The following two collectives are often used together. The first one, `MPI_Scatter`, splits a large piece of data stored in the root rank into smaller chunks and scatters them among all the ranks in a communicator including the root:

```
MPI_Scatter(
    &sendbuf,       // (1) send buffer
    sendcnt,        // (2) count of elements to send to each rank
    sendtype,       // (3) send data type
    &recvbuf,       // (4) receive buffer
    recvcnt,        // (5) count of elements to receive
    recvtype,       // (6) receive data type
    root,           // (7) root rank
    MPI_COMM_WORLD  // (8) communicator
);
```

```
call MPI_Scatter(   &
    sendbuf,        &! (1) send buffer
    sendcnt,        &! (2) count of elements to send to each rank
    sendtype,       &! (3) send data type
    recvbuf,        &! (4) receive buffer
    recvcnt,        &! (5) count of elements to receive
    recvtype,       &! (6) receive data type
    root,           &! (7) root rank
    MPI_COMM_WORLD, &! (8) communicator
    ierr)
```

The chunks are distributed following the index of the receiving rank, i.e. rank 0 receives the first chunk, rank 1 receives the second chunk, and so on. In pseudocode, the action of MPI_Scatter can be described as:

recvbuf^{0}[0 .. recvcnt-1] ⬅ sendbuf^{root}[0 .. sendcnt-1]
recvbuf^{1}[0 .. recvcnt-1] ⬅ sendbuf^{root}[sendcnt .. 2×sendcnt-1]
…
recvbuf^{N-1}[0 .. recvcnt-1] ⬅ sendbuf^{root}[(N-1)×sendcnt .. N×sendcnt-1]

The send buffer must be big enough to provide N×`sendcnt` elements where N is the number of ranks in the communicator. The triplet of arguments (1-3) specifying it are ignored at every other rank except the root. Unlike with the point-to-point operations, MPI requires that the amount of data sent to a rank is exactly equal to the size of the receive buffer. The data types on the send and on the receive sides do not have to be equal, but only congruent. Congruent data types are discussed in great detail in section MPI Datatypes. In many useful cases both data types are the same and then it follows that `sendcnt` must be equal to `recvcnt` as shown in the following example:

```
MPI_Scatter(  &bigbuf, 10, MPI_INT,
            &smallbuf, 10, MPI_INT,
            0, MPI_COMM_WORLD);
```

```
call MPI_Scatter(  bigbuf, 10, MPI_INTEGER, &
                 smallbuf, 10, MPI_INTEGER, &
                 0, MPI_COMM_WORLD, ierr)
```

MPI_Gather is the complimentary operation of MPI_Scatter and it collects back chunks of data from all ranks in the communicator into a single buffer at the root rank.

```
MPI_Gather(
    &sendbuf,        // (1) send buffer
    sendcnt,         // (2) count of elements to send
    sendtype,        // (3) send data type
    &recvbuf,        // (4) receive buffer
    recvcnt,    // (5) count of elements to receive from each rank
    recvtype,        // (6) receive data type
    root,            // (7) root rank
    MPI_COMM_WORLD  // (8) communicator
);
```

```
call MPI_Gather(    &
    sendbuf,         &! (1) send buffer
    sendcnt,         &! (2) count of elements to send
    sendtype,        &! (3) send data type
    recvbuf,         &! (4) receive buffer
    recvcnt,    &! (5) count of elements to receive from each rank
    recvtype,        &! (6) receive data type
    root,            &! (7) root rank
    MPI_COMM_WORLD, &! (8) communicator
    ierr)
```

The action of `MPI_Gather` can be formalised as:

sendbuf^{0}[0 .. sendcnt-1] ➡ recvbuf^{root}[0 .. recvcnt-1]
sendbuf^{1}[0 .. sendcnt-1] ➡ recvbuf^{root}[recvcnt .. 2×recvcnt-1]
…
sendbuf^{N-1}[0 .. sendcnt-1] ➡ recvbuf^{root}[(N-1)×recvcnt .. N×recvcnt-1]

Everything said about `MPI_Scatter` applies to `MPI_Gather`, only the roles of the send and the receive buffer are reversed. Each rank in the communicator supplies its chunk in sendbuf and the root rank (and only it) supplies a receive buffer that gathers all chunks. The receive buffer must be big enough to accommodate N×`recvcnt` elements.

The two operations are often used together. First, the root rank prepares the input data and scatters it, each rank then processes its own chunk, and finally the chunks are gathered back:

```
chunksize = sizeof(bigbuf) / num_ranks;

if (rank == 0)
{
    read_data(bigbuf);
}

MPI_Scatter(bigbuf, chunksize, dtype,  // ignored except at rank 0
            localbuf, chunksize, dtype,
            0, MPI_COMM_WORLD);

// . . .
// process localbuf
// . . .

MPI_Gather(localbuf, chunksize, dtype,
           bigbuf, chunksize, dtype,   // ignored except at rank 0
           0, MPI_COMM_WORLD);

if (rank == 0)
{
    write_data(bigbuf);
}
```

In this scenario, arguments (1-3) of `MPI_Scatter` become arguments (4-6) of `MPI_Gather` and vice versa.

A second class of collectives comprises versions of the above operations that do not have a designated root. Instead, the result of the operation is made available in all ranks. Those collectives have names that are prefixed with `MPI_All…` and the only other difference in their invocation is the lack of root argument. Both the send and the receive buffer arguments are significant in every rank of the communicator.

`MPI_Allreduce` performs global reduction and the result is available in all ranks. Below is a modified version of the reduction call show above:

```
MPI_Allreduce(
    &partial_sum,    // (1) send buffer
    &total_sum,      // (2) receive buffer
    1, MPI_INT,      // (3) count and datatype of elements
    MPI_SUM,         // (4) reduction operation
    MPI_COMM_WORLD   // (5) communicator
);
```

```
call MPI_Allreduce( &
    partial_sum,    & ! (1) send buffer
    total_sum,      & ! (2) receive buffer
    1, MPI_INTEGER, & ! (3) count and datatype of elements
    MPI_SUM,        & ! (4) reduction operation
    MPI_COMM_WORLD, & ! (6) communicator
    ierr)
```

`MPI_Allgather` works the same as `MPI_Gather`, but the chunks are gathered in all ranks in the communicator:

```
MPI_Allgather(
    &sendbuf,       // (1) send buffer
    sendcnt,        // (2) send chunk size
    sendtype,       // (3) send data type
    &recvbuf,       // (4) receive buffer
    recvcnt,        // (5) receive chunk size
    recvtype,       // (6) receive data type
    MPI_COMM_WORLD  // (7) communicator
);
```

```
```

```
call MPI_Allgather( &
    sendbuf,        &! (1) send buffer
    sendcnt,        &! (2) send chunk size
    sendtype,       &! (3) send data type
    recvbuf,        &! (4) receive buffer
    recvcnt,        &! (5) receive chunk size
    recvtype,       &! (6) receive data type
    MPI_COMM_WORLD, &! (7) communicator
    ierr)
```

Both all-collectives are semantically equivalent to a combination of the rooted collective followed by a broadcast of the result:

```
MPI_Reduce(&partial_sum, &total_sum, 1, MPI_INT, MPI_SUM,
           0, MPI_COMM_WORLD);
MPI_Bcast(&total_sum, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Due to the potential overlap in the inner workings of the two operations, the all-collectives are usually more efficient than the combination of a rooted collective and a broadcast.

There is no direct all-equivalent to `MPI_Scatter`, but `MPI_Alltoall` provides a combination of scatter and gather, which functions as a kind of global chunked transposition. Each rank performs a scatter of its send buffer to all other ranks following the workings of `MPI_Scatter`. The chunks scattered to a given rank are gathered into the receive buffer following the workings of `MPI_Gather`. Thus, rank 0 collects every first chunk from each rank. Rank 1 collects all second chunks, and so on. This is best illustrated graphically with four ranks:

Send buffers before the operation:

	0 .. sendcnt-1	sendcnt .. 2xsendcnt - 1	2xsendcnt .. 3xsendcnt - 1	3xsendcnt .. 4xsendcnt - 1
Rank 0	a0a0a0a0	a1a1a1a1	a2a2a2a2	a3a3a3a3
Rank 1	b0b0b0b0	b1b1b1b1	b2b2b2b2	b3b3b3b3
Rank 2	c0c0c0c0	c1c1c1c1	c2c2c2c2	c3c3c3c3
Rank 3	d0d0d0d0	d1d1d1d1	d2d2d2d2	d3d3d3d3

Receive buffers after the operation:

	0 .. recvcnt-1	recvcnt .. 2xrecvcnt - 1	2xrecvcnt .. 3xrecvcnt - 1	3xrecvcnt .. 4xrecvcnt - 1
Rank 0	a0a0a0a0	b0b0b0b0	c0c0c0c0	d0d0d0d0
Rank 1	a1a1a1a1	b1b1b1b1	c1c1c1c1	d1d1d1d1
Rank 2	a2a2a2a2	b2b2b2b2	c2c2c2c2	d2d2d2d2
Rank 3	a3a3a3a3	b3b3b3b3	c3c3c3c3	d3d3d3d3

Varying-Size Versions

All scatter, gather, and all-to-all collectives shown so far communicate chunks of the same size, which restricts the number of elements in the data buffers to multiples of the number of ranks in the communicator. More often than not, this is not the case. MPI provides varying-size versions of all such collectives. Those have a `-v` suffix and replace the count argument with two arrays: of offsets and counts. The varying-size scatter is:

```
MPI_Scatterv(
    &sendbuf,        // (1) send buffer
    &sendcnts,       // (2) array of send chunk sizes
    &displs,         // (3) array of send chunk displacements
    sendtype,        // (4) send data type
    &recvbuf,        // (5) receive buffer
    recvcnt,         // (6) count of elements to receive
    recvtype,        // (7) receive data type
    root,            // (8) root rank
    MPI_COMM_WORLD   // (9) communicator
);
```

```
call MPI_Scatterv(  &
    sendbuf,        &! (1) send buffer
    sendcnts,       &! (2) array of send chunk sizes
    displs,         &! (3) array of send chunk displacements
    sendtype,       &! (4) send data type
    recvbuf,        &! (5) receive buffer
    recvcnt,        &! (6) count of elements to receive
    recvtype,       &! (7) receive data type
    root,           &! (8) root rank
    MPI_COMM_WORLD, &! (9) communicator
    ierr)
```

`sendcnts` and `displs` are integer arrays of size equal the number of ranks in the communicator that provide the length and the displacement from the beginning of the buffer of the chunk for each rank. Unlike `MPI_Scatter`, `MPI_Scatterv` allows gaps in the send data by positioning the chunks accordingly. The requirement that the size of the data sent from the root to rank *i* is equal to the size of the receive buffer specified by rank *i* remains. `MPI_Gatherv` simply swaps the send and the receive arguments and the direction of the data flow. `MPI_Alltoallv` replaces both the send and the receive counts with pairs of arrays. There is an even more generic version of the all-to-all collective, `MPI_Alltoallw`, that also allows for each chunk to have its own data type. For brevity, the full signatures are not shown here and the readers are advised to consult the MPI specification or the manual pages of their MPI implementation.

Barrier Synchronisation

Unlike in the shared-memory programming paradigms, explicit synchronisation between MPI ranks is rarely necessary. In most cases, ranks synchronise pairwise through the point-to-point operations. In rare cases it is necessary to synchronise the execution of all ranks in a given communicator and `MPI_Barrier` does exactly that.

```
MPI_Barrier(MPI_COMM_WORLD);
```

```
call MPI_Barrier(MPI_COMM_WORLD, ierr)
```

If rank *k* enters `MPI_Barrier` at time instant $\mathbf{t}^k_{\mathbf{In}}$ and exits it at time instant $\mathbf{t}^k_{\mathbf{Out}}$ ($\mathbf{t}^k_{\mathbf{Out}} > \mathbf{t}^k_{\mathbf{In}}$), then `MPI_Barrier` guarantees that there is a point in time $\mathbf{t}$, for which $\mathbf{t}^k_{\mathbf{Out}} > \mathbf{t} > \mathbf{t}^k_{\mathbf{In}}$ for all ranks *k*. In other words, there is a point in time, in which all ranks are simultaneously inside the `MPI_Barrier` call. No guarantees are given to the exit times $\mathbf{t}^k_{\mathbf{Out}}$ though with many implementations those are usually pretty narrowly distributed.

A typical use case for the barrier synchronisation is benchmarking parts of the code. Many factors, including the initial launch of the processes that make up the MPI job, lead to the different MPI ranks becoming desynchronised in time. Benchmarking the code without first synchronising all ranks in time leads to spurious delays getting counted into the execution time. Also, many parallel algorithms are overly sensitive to inter-rank delays due to the phenomenon of *delay propagation*. Although practically never zero, the temporal desynchronisation after the barrier call is orders of magnitude smaller than before it.

Another use case is synchronisation during parallel I/O and the single-sided remote-memory operations of MPI, which bring MPI closer to the programming model of OpenMP and are not discussed in this course.

Again, explicit barrier synchronization is rarely needed in MPI programs and is more often than not abused unnecessarily with negative effects on the performance.

Non-blocking Operations

All operations discussed so far are blocking in that they do not return control back to the user program until MPI no longer needs access to the data buffers. While safe, this is often suboptimal because the user code is blocked from doing anything else while the communication is taking place. In many algorithms there are computational parts that can proceed independently of communication and it is often possible to have both running in parallel, an approach known as *communication-computation overlap*.

MPI makes possible the overlap of communication and computation through the means of non-blocking operations. Unlike the blocking operations, the non-blocking ones only initiate the operation and return immediately with a handle of the operation that can then be waiter upon or periodically tested for completion. Syntactically, the difference between the blocking and the non-blocking operations is that the former have their names prefixed with `I` (capital letter I for “initiation”) and an additional output argument that receives the handle of the operation. For example, the non-blocking standard send is:

```
MPI_Request req;

MPI_Isend(
    buf,               // (1) buffer location
    sendcnt,           // (2) number of elements in the buffer
    dtype,             // (3) data type
    rank_of_recv,      // (4) rank of the receiver
    tag,               // (5) message tag
    MPI_COMM_WORLD,    // (6) communicator
    &req               // (7) request handle
);
```

```
integer :: req

call MPI_Isend(        &
    buf,               & ! (1) buffer location
    sendcnt,           & ! (2) number of elements in the buffer
    dtype,             & ! (3) data type
    rank_of_recv,      & ! (4) rank of the receiver
    tag,               & ! (5) message tag
```

```
    MPI_COMM_WORLD,    & ! (6) communicator
    req,               & ! (7) request handle
    ierr               & ! (8) error code (output)
)
```

The request handle is an opaque value that is of type MPI_Request in C or simply an integer in Fortran. It identifies the operation and can be waited upon until the operation is complete with `MPI_Wait`:

```
MPI_Status status;

MPI_Wait(&req, &status);
```

```
integer, dimension(MPI_STATUS_SIZE) :: status

call MPI_Wait(req, status, ierr)
```

Once the operation identified by `req` completes, the request is set to `MPI_REQUEST_NULL` and the status object is filled with information about the completed operation. Again, the program is aborted in case of error and hence the status object carries no useful information about non-blocking send operations. If the status is of no interest, `MPI_STATUS_IGNORE` can be passed instead.

`MPI_Wait` itself is blocking in the sense that it does not return before the operation is complete. There is a non-blocking version that only tests the current status of the operation:

```
MPI_Status status;
int flag;

MPI_Test(&req, &flag, &status);
```

```
integer, dimension(MPI_STATUS_SIZE) :: status
integer :: flag

call MPI_Test(req, status, flag, ierr)
```

If the operation is complete, `flag` is set to a non-zero value, the request object is set to `MPI_REQUEST_NULL`, and the status object is filled with information. Otherwise, `flag` is set to zero and both the request and the status object are left untouched.

The non-blocking receive operation is `MPI_Irecv`:

```
MPI_Request req;

MPI_Irecv(
    buf,                // (1) buffer location
    recvcnt,            // (2) buffer capacity in elements
    dtype,              // (3) data type
    rank_of_sender,     // (4) rank of the sender
    tag,                // (5) message tag
    MPI_COMM_WORLD,     // (6) communicator
    &req                // (7) request handle
);
```

```
integer :: req

call MPI_Irecv(        &
    buf,               & ! (1) buffer location
    recvcnt,           & ! (2) buffer capacity in elements
    dtype,             & ! (3) data type
    rank_of_sender,    & ! (4) rank of the sender
    tag,               & ! (5) message tag
    MPI_COMM_WORLD,    & ! (6) communicator
    req,               & ! (7) request handle
    ierr)                ! (8) error code (output)
```

The non-blocking operations simply split the atomic blocking operations in two parts:

- Initiation
- Waiting/testing for completion

Nothing else distinguishes the non-blocking from the blocking operations. A blocking operation is semantically equivalent to a non-blocking operation followed by a wait operation:

```
MPI_Send(&buf, cnt, dtype, dst, 0, MPI_COMM_WORLD);
```

```
MPI_Request req;

MPI_Isend(&buf, cnt, dtype, dst, 0, MPI_COMM_WORLD, &req);
MPI_Wait(&req, MPI_STATUS_IGNORE);
```

Messages sent by non-blocking operations are exactly the same as those sent by blocking operations and the communication counterpart cannot distinguish between them. Therefore,

it is not necessary to match non-blocking operations with non-blocking operations on both sides of the communication. It is perfectly acceptable to have a non-blocking send matched by a blocking receive or vice versa. In fact, in some MPI implementations, the blocking operations are implemented internally as non-blocking ones followed immediately by a wait.

There are non-blocking equivalents of the collectives too. For example, the non-blocking broadcast operation is:

```
MPI_Request req;

MPI_Ibcast(&buf, cnt, dtype, root, MPI_COMM_WORLD, &req);
```

```
integer :: req

call MPI_Ibcast(buf, cnt, dtype, root, MPI_COMM_WORLD, req, ierr)
```

Unlike the point-to-point operations, **it is not possible to mix non-blocking and blocking calls for the same collective operation**.

It is possible to wait on or test several non-blocking operations at the same time. There are `-all`, `-some`, and `-any` versions of both `MPI_Wait` and `MPI_Test` that do exactly what their names imply. `MPI_Waitall` receives an array of non-blocking requests and waits until all of them complete, `MPI_Waitany` returns when one of the operations complete, while `MPI_Waitsome` returns when one or more requests complete. The same applies to the multiple requests versions of `MPI_Test`.

Important: One must always keep in mind that until a non-blocking operation is successfully completed by either waiting on its request or when the test for completion comes out true the send and/or receive buffer(s) provided to the operation must not be modified (in case it is an input buffer) or will not have a well-defined value (in case it is an output buffer.)

MPI provides non-blocking message probing, but the way it works is different from the rest of the non-blocking operations:

```
int flag;

MPI_Iprobe(sender, tag, MPI_COMM_WORLD, &flag);
```

```
integer :: flag

call MPI_Iprobe(sender, tag, MPI_COMM_WORLD, flag, ierr)
```

Unlike the rest of the I-operations, `MPI_Iprobe` does not return an operation request handle. Instead, it simply checks whether there is a message matching the receive criteria and sets the `flag` variable accordingly.

Non-blocking operations can be used to solve the potential deadlock problem discussed in the section about the semantics of point-to-point message passing. One could use non-blocking sends:

```
int other_rank = get_other_rank();
MPI_Request req;

MPI_Isend(&to_send, 1, MPI_INT, other_rank, 0, MPI_COMM_WORLD,
          &req);
MPI_Recv(&received, 1, MPI_INT, other_rank, 0, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
MPI_Wait(&req, MPI_STATUS_IGNORE);
```

Alternatively, one could first issue a non-blocking receive, followed by a blocking send and a wait. In fact, that is how `MPI_Sendrecv` is implemented in Open MPI.

MPI Datatypes

Being a library, MPI needs a way to understand the data it is moving between the ranks. In order to achieve that understanding, it has a very comprehensive system that allows the construction of type descriptors known as *MPI datatypes*, which are templates that tell MPI what are the underlying language types that comprise the data to be sent and its precise layout in memory.

Each MPI datatype has a *type map*, which is an ordered collection of pairs (tuples) of basic (language) type and integer displacement from the beginning of the data buffer: *{($type_0$, $disp_0$), ($type_1$, $disp_1$), …}*
Here, $type_i$ is a basic type, for example `int` or `float` in C. $disp_i$ is a signed integer displacement in bytes from the beginning of the data buffer. There can be both positive and negative displacements, which allows for the creation of some interesting data types. The ordered collection of basic types from each pair is known as the *type signature*. Two MPI datatypes are said to be *congruent* if their type maps only differ in the displacement but not in the basic types in the pairs, i.e., if the types share the same type signature. For example, the following two data types are congruent: *{(int, 0), (char, 4), (double, 8)}* and *{(int, 8), (char, -2), (double, 0)}*. The type signature of both types is *{int, char, double}*.

All predefined MPI datatypes that correspond to a language type have type maps that consist of a single entry -- the basic type at zero displacement. For example, the type map of

`MPI_INT` is *{(int, 0)}*. The smallest displacement in any pair is known as the *lower bound* (lb) of the data type. The largest displacement plus the size of the basic type and any alignment padding required by the language is called the *upper bound* (ub) of the data type:

lb(*type*) = min$_i$ *offset*$_i$
ub(*type*) = max$_i$ (*offset*$_i$ + sizeof(*type*$_i$)) + padding$_i$

The sum of the sizes of all language types in the type map is the *size* of the data type, while the difference between the upper and the lower bounds is its *extent*:

size(*type*) = sum$_i$ sizeof(*type*$_i$)
extent(*type*) = ub(*type*) - lb(type)

The size of a type is directly related to the amount of bytes it occupies in a message. The extent of a type is important when more than one element of that type is being sent or received. When MPI goes from one element in the data buffer to the next one, it uses a stride that is equal to the extent of the type. Since the offsets in the typemap can be arbitrary, it is possible to have holes in the memory layout and hence the extent can be larger than the size. This is not always the case though since MPI allows for arbitrary setting of the lower and upper bounds of a type, which can alter its apparent size and which allows for some really complex data manipulations.

Type congruence is an important concept in MPI. As was mentioned earlier, the types on both sides of a communication, be it a point-to-point one or a collective, must not necessarily be the same, but they have to be congruent.

Derived Datatypes

The basic data types are often not enough in many communication scenarios. MPI allows the construction of more complex data types from existing ones, a process that always starts with the basic MPI data types. There are several *type constructors* that combine existing types in different ways.

The simplest one is `MPI_Type_continuous`, which repeats elements of a data type into contiguous locations, i.e., a type that represents an array of `count` elements.

```
MPI_Datatype newtype;

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

```
integer :: newtype

call MPI_Type_contiguous(count, oldtype, newtype, ierr)
```

The type map of the new type is a repetition of the type map of `oldtype` with the displacements adjusted accordingly. When sending or receiving with count greater than one, MPI implicitly creates a contiguous data type. Therefore, the following code is perfectly valid:

```
MPI_Datatype ten_ints;

MPI_Type_contiguous(10, MPI_INT, &ten_ints);
MPI_Type_commit(&ten_ints);

if (rank == 0)
    MPI_Send(buf, 10, MPI_INT, 1, 0, MPI_COMM_WORLD);
else if (rank == 1)
    MPI_Recv(buf, 1, ten_ints, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
```

Another useful constructor is the vector constructor `MPI_Type_vector`, which creates a repetition of equally spaced blocks:

```
MPI_Type_vector(
    count,        // number of blocks
    blocklength,  // length of a single block in elements
    stride,       // distance between consecutive block starts
    oldtype,      // element type
    &newtype      // new type handle
);
```

```
call MPI_Type_vector( &
    count,       & ! number of blocks
    blocklength, & ! length of a single block in elements
    stride,      & ! distance between consecutive block starts
    oldtype,     & ! element type
    newtype,     & ! new type handle
    ierr)
```

Vector types are very useful for working with higher-dimensional data like multidimensional arrays. A two-dimensional C array `int arr[M][N]` has M rows of N elements each. C uses row-major storage, so the elements of a single array are continuously laid out in memory and a perfect candidate for the contiguous MPI type. If one wants to communicate a column instead, the data consists of `N` blocks of length one element that are spaced out by the length of a row (`M`), which is what the vector type is about. The following call creates the type represents a single column of `arr`:

```
MPI_Datatype columndt;

MPI_Type_vector(N, 1, M, MPI_INT, &columndt);
```

A similar case applies in Fortran, which uses column-major storage for multidimensional arrays, so if one wants to send a row from a two-dimensional array integer arr(M, N), then a vector type needs to be involved:

```
integer :: rowdt

call MPI_Type_vector(M, 1, N, MPI_INTEGER, rowdt, ierr)
```

A more general version of the vector constructor is `MPI_Type_indexed`, which creates a data type that represents a sequence of blocks, where each block can have a different number of elements and have a different displacement.

```
MPI_Type_indexed(
    count,          // number of blocks
    blocklenghts,   // array of block lengths
    displacements,  // array of block displacements
    oldtype,        // element type
    &newtype        // new type handle
);
```

```
call MPI_Type_indexed( &
    count,         & ! number of blocks
    blocklengths,  & ! array of block lengths
    displacements, & ! array of block displacements
    oldtype,       & ! elemen type
    newtype,       & ! new type handle
    ierr)
```

Both `blocklengths` and `displacements` are integer arrays of no less than `count` elements. The displacements are in elements from the beginning of the data buffer.

Finally, the most generic type constructor allows also for the elements of each block to be of a different type. This makes it possible to send and receive data of complex types such as C structures or Fortran structures and records. Appropriately, the constructor is named `MPI_Type_create_struct`.

```
MPI_Type_create_struct(
    count,          // number of blocks
    blocklengths,   // array of block lengths
    displacements,  // array of block displacements
    oldtypes,       // array of element types
    &newtype        // new type handle
);
```

```
call MPI_Type_create_struct( &
    count,         & ! number of blocks
    blocklengths,  & ! array of block lengths
    displacements, & ! array of block displacements
    oldtypes,      & ! array of element types
    newtype,       & ! new type handle
    ierr)
```

`blocklengths`, `displacements`, and `oldtypes` are all arrays of no less than `count` elements. `blocklengths` contains the length of each block in elements. `displacements` specifies the offset of each block in **bytes** from the beginning of the data buffer. `oldtypes` provides the data type for the elements of each block.

Resizing Structure Types

Structure types created this way are good for sending or receiving one item of that type. If an array of such elements is to be communicated, the type has to be massaged further a bit. The reason is that with struct types the alignment logic built into MPI often fails to adjust the type extent to exactly match the size of the corresponding compound language type. The provisions of MPI that allow modification of the lower and upper bounds of the data type have to be employed. The following code shows how to create an MPI type that corresponds to a C structure and how to resize it so that it can be used to communicate an array of such structures.

```
typedef struct _point
{
   double r[3];
   double v[3];
   double mass;
   char kind;
} point;

int lenghts[] = { 3, 3, 1, 1 };
MPI_Aint displacements[] = {
```


```
    offsetof(point, r),    offsetof(point, v),
    offsetof(point, mass), offsetof(point, kind)
};
MPI_Datatype types[] = {
    MPI_DOUBLE, MPI_DOUBLE, MPI_DOUBLE, MPI_CHAR
};

MPI_Datatype dt;
MPI_Type_create_struct(4, lengths, displacements, types, &dt);

// Resize the type such that its extent matches sizeof(point)
MPI_Datatype pointdt;
MPI_Type_create_resized(dt, (MPI_Aint)0, sizeof(point), &pointdt);
MPI_Type_commit(&pointdt);

MPI_Send(points, N, pointdt, …);
```

Type Registration

MPI data types created using the above constructors can be used for further constructing even more complex data types. But before they can be used for communication, they have to be registered by *committing* them with `MPI_Type_commit`.

```
MPI_Type_commit(&dtype);
```

```
call MPI_Type_commit(dtype, ierr)
```

Committing a data type allows MPI to perform various tasks such as optimising the internal type representation to make it more efficient for the underlying network equipment. Only types that are used in communication have to be committed.

Once a data type is no longer needed, it has to be freed with a call to `MPI_Type_free`.

```
MPI_Type_free(&dtype);
```

```
call MPI_Type_free(dtype, ierr)
```

When a data type is used to construct another data type, its type information is copied into the new data type and not simply referenced. This means that such data types can be freed immediately after the more complex type is created. One should never attempt to free a predefined data type such as `MPI_INT`. Those are taken care of by the MPI itself.

A Bit About mpi4py

Although not part of the MPI specification, the unofficial Python interface *mpi4py* is gaining popularity. Since it is based on the former C++ bindings (no longer part of the MPI standard), many of the topics discussed so far apply to it too. But due to the dynamic nature of Python, *mpi4py* goes a step further and provides services that go well beyond what the standard MPI bindings have.

The main difference between the standard MPI language bindings and *mpi4py* is that the latter takes a more object-oriented approach. For example, query operations that take a communicator such as `MPI_Comm_size` and `MPI_Comm_rank` are methods of the object that represents the communicator. The difference is best demonstrated with a C and an *mpi4py* version of the same program as shown together below:

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

int main (void)
{
  int rank, size;

  MPI_Init(NULL, NULL);

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

  printf("Hello MPI world from rank %d of %d\n", rank, size);

  MPI_Finalize();

  return 0;
}
```

```python
from mpi4py import MPI

rank = MPI.COMM_WORLD.Get_rank()
size = MPI.COMM_WORLD.Get_size()

print(f"Hello MPI world from rank {rank} of {size}")
```

The Python program is way shorter, mainly because *mpi4py* takes care of MPI initialisation and finalisation. MPI operations that happen in the context of some MPI object are all implemented as methods of the class representing that object. For example, all communication operations are methods of the class which represents MPI communicators. The naming differs slightly from the standard, e.g., `MPI_Comm_rank` becomes `comm.Get_rank()`.

An interesting feature of *mpi4py* is that each communication primitive comes in two distinct flavours. The first flavour corresponds to the operations defined in the standard and deals with buffer-like objects, mostly *numpy* arrays. The name of the corresponding method starts with a capital letter, e.g., `comm.Send()`, `comm.Irecv()`, and so on. The buffer is provided as a tuple or a list of 2 or 3 items: ***(buffer, count, MPI.SOMETYPE)*** or ***(buffer, MPI.SOMETYPE)*** In the latter case, the count is determined automatically by dividing the byte size of the buffer object by the extent of the MPI data type. The rank of the communication partner and the message tag are provided as keyword arguments and can be skipped in order to use their default values:

```
from mpi4py import MPI
import numpy as np

world = MPI.COMM_WORLD
rank = world.Get_rank()

if rank == 0:
    sbuf = np.arange(100, dtype='i')
    world.Send([sbuf, MPI.INT], dest=1)
elif rank == 1:
    rbuf = np.empty(100, dtype='i')
    world.Recv([rbuf, MPI.INT], source=0)
```

The other flavour has no analogous operations in MPI and deals with communication of generic Python objects. This is made possible by the standard library functionality of Python for object serialisation and deserialisation. The methods that take Python objects all start with a lowercase letter, e.g., `comm.send()`, `comm.irecv()`, and so on.

```
from mpi4py import MPI

world = MPI.COMM_WORLD
rank = world.Get_rank()

if rank == 0:
    sdata = {'foo': 1, 'bar': 42}
```

```
    world.send(sdata, dest=1)
elif rank == 1:
    rdata = world.recv(source=0)
```

The `comm.recv()` method directly returns the received data and error conditions are signalled by throwing exceptions.

All non-blocking operations return a request object with wait and test methods that again come in two flavours distinguished by the case of the first letter - uppercase `comm.Wait()` / `comm.Test()` when dealing with buffers and lowercase `comm.wait()` / `comm.test()` when dealing with Python objects. The latter methods return the received data when the operation is a non-blocking receive:

```
from mpi4py import MPI

world = MPI.COMM_WORLD
rank = world.Get_rank()

if rank == 0:
    sdata = {'foo': 1, 'bar': 42}
    req = world.isend(sdata, dest=1)
    req.wait()
elif rank == 1:
    req = world.irecv(source=0)
    rdata = req.wait()
```

Collectives also provide buffer and generic object versions distinguished by the case of the first letter in their method's name. The following sample code gathers the squared IDs of all ranks in the world communicator:

```
from mpi4py import MPI

rank = MPI.COMM_WORLD.Get_rank()
rank2 = rank**2
data = MPI.COMM_WORLD.gather(rank2, root=0)
if rank == 0:
    print(data)
```

Executor Pools

One of the coolest features of *mpi4py* is its interface that resembles the `multiprocessing` module from the standard Python library. More specifically, the MPI executor pool which allows distribution of work from one main process to a pool of worker processes with MPI used for communication between them. This is an extension of the executor pool provided by the multiprocessing module, which can run on more than one host. The following sample code demonstrates how to square each item of a list. This is a silly example and for such a simple operation no parallel speedup is to be expected, but it demonstrates the core principles.

```
from mpi4py.futures import MPIPoolExecutor

def main():
    data = range(100)
    with MPIPoolExecutor() as pool:
        data2 = pool.map(lambda x: x**2, data, chunksize=10)

if __name__ == '__main__':
    main()
```

Because the same script file is both run directly as the main process and imported by the workers in the pool, the code for the main must be separated so it doesn't execute when the script is imported as a module. A new executor pool is spawned by creating an instance of `MPIPoolExecutor`. The pool provides methods for queuing work items and for distributed application of transformations. In this particular example, the lambda function that squares its argument is applied to all elements of the data array in chunks of 10 elements.

The pool is either spawned as a child MPI job or uses a set of pre-launched scripts running code in the `mpi4py.futures` module. In the former case, the program is launched as such using Open MPI:

```
$ OMPI_UNIVERSE_SIZE=5 mpiexec -n 1 python main.py
```

This will result in a total of 5 MPI ranks, 4 of which will be started as part of the executor pool. As an alternative, the 5 ranks can be launched from the start, but have to be made to execute a special function in the mpi4py.futures module:

```
$ mpiexec -n 5 python -m mpi4py.futures main.py
```

The final effect is the same in both cases and the difference is mainly dictated by the ability of the MPI implementation to deal with process management.

Executor pools have no equivalent in standard MPI and implementing them requires a non-trivial amount of coding in C or Fortran.

Installing mpi4py

mpi4py is available on PyPI and can be installed with `pip`. Because there is not one MPI implementation and because the different implementations are not binary compatible, the package does not come precompiled and is instead compiled during installation using the available MPI implementation. On multiuser systems one has to either install mpi4py into the user library path with

```
$ pip install --user mpi4py
```

or (better) into a virtual environment or a Conda environment:

```
$ python -m venv env
$ . env/bin/activate
(env) $ pip install mpi4py
```

```
$ conda create --name mpi-dev python=3.8
$ conda activate mpi-dev
(mpi-dev) $ pip install mpi4py
```

mpi4py uses `mpicc` by default as the name of the MPI C compiler wrapper. The name can be overridden by setting the `MPICC` environment variable.