Message-Passing and MPI Programming
Using MPI

N.M. Maclaren

nmm1@cam.ac.uk

July 2010

2.1 Working With MPI

There are a huge number of minor points that need mentioning, including all of the ' housekeeping' facilities. You do not need to remember all of the details, initially, but try to remember which facilities are included and refer back to this document when doing the practicals. It is a **lot** easier than it looks at first!

By default, all actual errors are fatal, and MPI will produce some kind of an error message. With luck, the whole program will then stop – if you are unlucky, some processes may hang and you will have to kill them by hand. You can ask to do your own error handling, and that is described later.

You should use **one** interface: i.e. Fortran or C. The MPI Forum now supports C++ programmers only by allowing MPI's C interface to be called, possibly by using '`extern "C"`', and this is what this course now teaches. Calling MPI using both its Fortran or C interfaces in the same program is possible, but it is advanced use and is not covered by this course.

In the extra (online) materials, there are files containing proformas for all functions used in the examples or in the practicals; anything merely mentioned but not described is omitted, for clarity. The files are `Interfaces/Fortran` and `Interfaces/C`. The course does not give the syntax in detail, so check those files when doing the practicals.

2.2 MPI's Fortran Interface

If possible, include the statement `USE mpi` at the start of the main program, any module and any external subroutine or function. If not, use `INCLUDE 'mpif.h'` after all "`USE`" statements and "`IMPLICIT`" in the same places. Note that the first is "`mpi`" and the second "`mpif.h`". If both of these fail, it usually means that you have a usage or installation problem, such as not having set search paths correctly.

All MPI names start with `MPI_`. Do not declare your own names starting `MPI_` or `PMPI_`; names starting `PMPI_` are used for profiling.

Boolean values (i.e. ones that are true or false) are `LOGICAL`.

Process numbers, error codes etc. are `INTEGER`.

Element counts etc. are also plain `INTEGER` – this is not a problem on any current system.

Almost all MPI constants are Fortran constants (`PARAMETER`); the only exception mentioned in this course is `MPI_IN_PLACE`.

Arrays start at one, where it matters.

Type-generic ("choice arguments" in MPI's terms) arguments are a kludge – MPI relies on Fortran not checking the types. The course will describe some of the issues later. MPI 3 and Fortran TS 29113 fixes this issue properly. For now, just pass arrays of any type – if the compiler objects, ask a Fortran expert for help.

Handles (e.g. communicators) are opaque types; those are ones you cannot break apart and look inside. In Fortran, they are undocumented and unpredictable `INTEGER` values. You can test them for (in)equality and assign them using Fortran's built-in operations, but call the appropriate MPI functions for **all** other operations. Another way of viewing this is that MPI returns such values as `INTEGER` tokens; if two such values match, they are the same token, but nothing more is specified about their values.

Almost all MPI functions are subroutines, and the final argument returns an `INTEGER` error code. Success returns `MPI_SUCCESS`, which is always zero; failure codes are implementation dependent. Their results are returned through arguments. There are only a very few exceptions, and the only one that most people will use is `MPI_Wtime`.

As people will know, Fortran's default `REAL` is a disaster for most scientific programming, and `DOUBLE PRECISION` is tedious and out-of-date. You should start all procedures, modules etc. with something like:

```
USE double
USE mpi
IMPLICIT NONE
```

There is a suitable file to create the `double` module in `Programs/double.f90`; you should ask for help if you do not know how to use it.

2.3 MPI's C Interface

You need to include the statement `#include "mpi.h"`. For C, that is all you need to do, and it *may* work for C++.

If it does not work in a C++ program, the simplest solution is to try:

```
extern "C" {
    #include "mpi.h"
}
```

Another approach is to put your MPI code into a separate file of C source (typically ending `.c`), write that file in pure C, and compile it using a C compiler that is compatible with your C++ one. You can then use `extern "C"` to use that file from your C++.

All C names start with `MPI_`. Do not declare your own names starting `MPI_` or `PMPI_`; names starting `PMPI_` are used for profiling.

Boolean values (i.e. ones that are true or false) are `int`, as usual.

Process numbers, error codes etc. are `int`.

Element counts etc. are also plain `int` – this is not a problem on any current system.

Type-generic arguments ("choice arguments" in MPI's terms) are `void *` pointers.

Almost all MPI constants are C *initialization expressions*, but not usually *preprocessor constants* or *integer constants*, so they cannot be used in `case`, array sizes etc. Only the maximum sizes are *preprocessor* constants.

Arrays start at zero, where it matters.

Handles (e.g. communicators) are opaque types; their names are set up by `typedef` and are scalars. You can test them for (in)equality and assign them using C's built-in operations, but call the appropriate MPI functions for **all** other operations. The main such opaque types are `MPI_Comm`, `MPI_Datatype`, `MPI_Errhandler`, `MPI_Group`, `MPI_Op`, `MPI_Request` and `MPI_Status`. Another way of viewing this is that MPI returns such values as tokens; if two such values match, they are the same token, but nothing more is specified about their types or values.

Almost all MPI functions have an `int` result type, and return an error code. You can ignore it, as usual in C, if you are using default error handling. Success returns `MPI_SUCCESS`, which is always zero; failure codes are implementation dependent. Their results are returned through pointer arguments. There are only a very few exceptions, and the only one that most people will use is `MPI_Wtime`.

2.4 MPI's C++ Interface

MPI 2.0 introduced a C++ interface in 1997, which significantly better in a great many respects; it was a "proper" C++ one, not just a hacked C one, and that caused mainentance problems. For that reason, MPI 2.2 deprecated it in 2009, and MPI 3.0 deleted in in 2012. Its recommendation is to use the C interface from C++, and this is what this course teaches.

2.5 MPI Setup

For now, we will ignore error handling. **All** processes must start by calling `MPI_Init` and, normally, all finish by calling `MPI_Finalize`. These are effectively collectives, and you should call both of them at predictable times, or risk confusion. You **must not** restart MPI after `MPI_Finalize` – i.e. `MPI_Init` must be called exactly once.

Fortran:

Fortran argument decoding is done behind the scenes, so the following is all you need.

```
USE double
USE mpi
IMPLICIT NONE
INTEGER :: error

CALL MPI_Init ( error )
< do the actual work >
CALL MPI_Finalize ( error )
END
```

If that does not work, see the installation notes, or ask for help.

C:

`MPI_Init` takes the **addresses** of `main`'s arguments, not the arguments themselves. You **must** call it before decoding them, because some implementations change them in `MPI_Init`.

```
#include "mpi.h"

int main (int argc , char * argv [] ) {
    MPI_Init ( & argc , & argv ) ;
    < do the actual work >
    MPI_Finalize ( ) ;
    return 0 ;
}
```

Aside: Examples

All of the examples will omit the following statements, for brevity:

Fortran:

```
USE double
USE mpi
IMPLICIT NONE
```

C:

```
#include "mpi.h"
```

Include them in any "module" where you use MPI (where "module" includes Fortran external procedures and C/C++ files). You are **strongly** advised not to rely on implicit declaration – it often works in one implementation, and fails on another.

2.6 MPI State and Constants

MPI 1.2 and up provide version number information; it is rarely needed, except when investigating errors. There are constants `MPI_VERSION` and `MPI_SUBVERSION`. These are set to `1` and `3` for MPI 1.3 or `2` and `2` for the current version, MPI 2.2. There is also a function `MPI_Get_version`, which can be called even before `MPI_Init`.

You can test the state of MPI in a process – this is normally needed only when writing library code. `MPI_Initialized` returns whether MPI has been initialised, and `MPI_Finalized` tests whether it has been finalised.

Fortran:

```
LOGICAL :: started , stopped
INTEGER :: error
CALL MPI_Initialized ( started , error )
CALL MPI_Finalized ( stopped , error )
```

C:

```
int started , stopped , error ;
error = MPI_Initialized ( & started ) ;
error = MPI_Finalized ( & stopped ) ;
```

The global communicator is predefined: `MPI_COMM_WORLD`. It includes all usable processes – e.g. the `<n>` set up by "`mpiexec -n <n>`". Many applications use only this communicator, almost all of this course does, too. There is one lecture on communicators.

The *rank* is the process's index within the context of a communicator (i.e. a process may have different ranks in different communicators). It is an integer from `0` to `<n>-1`, in all languages, including Fortran. There is one predefined rank constant: `MPI_PROC_NULL`, meaning "no such process". Do not assume either that this is negative or that it is not! We shall describe the use of it when it becomes relevant.

2.7 Information Calls

`MPI_Comm_size` returns the number of processes, and `MPI_Comm_rank` returns the local process number (i.e. the rank).

Fortran:

```
INTEGER :: nprocs , myrank , error
CALL MPI_Comm_size ( MPI_COMM_WORLD , nprocs , error )
CALL MPI_Comm_rank ( MPI_COMM_WORLD , myrank , error )
```

C:

```
int nprocs , myrank , error ;
error = MPI_Comm_size ( MPI_COMM_WORLD , & nprocs ) ;
error = MPI_Comm_rank ( MPI_COMM_WORLD , & myrank ) ;
```

You can query the local processor name, and this stores it in a character array of length `MPI_MAX_PROCESSOR_NAME`. This applies to C as well as Fortran – it does **not** return a C string.

Fortran:

```
CHARACTER ( LEN = MPI_MAX_PROCESSOR_NAME ) :: procname
INTEGER :: namelen , error
CALL MPI_Get_processor_name ( procname , namelen , error )
```

C:

```
char procname [ MPI_MAX_PROCESSOR_NAME + 1 ] ;
int namelen , error ;
```

```
error = MPI_Get_processor_name ( procname ,  & namelen ) ;
procname [ namelen ] = '\0' ;
```

MPI_Wtime gives the elapsed time (i.e. the "wall-clock time"), in seconds since an unspecified starting point. The starting point is fixed for a process and does not change while the process is running. I have seen the start of process, the system boot time, the Unix epoch and 00:00 Jan. 1st 1900; always use the difference between values and not the actual values. MPI_Wtick is similar but gives the timer resolution (i.e. precision); few people bother with it, but it is there if you want it.

Fortran:

```
REAL(KIND=KIND(0.0D0)) :: now
now = MPI_Wtime ( )
```

C:

```
double now ;
now = MPI_Wtime ( ) ;
```

You can use the information calls anywhere following the call to MPI_Init and preceding the call to MPI_Finalize. They are all purely local operations, so use them as often as you need them. MPI_Comm_size will give the same result on all processes, but all of the others may give different results on each process. That includes MPI_Wtime's starting point as well as the value returned from MPI_Wtick.

2.8 Other Important Utilities

MPI_Barrier synchronises all processes. They all wait until they have all entered the call, and then they all start up again, and continue executing independently. This is the **only** collective that synchronises in that way; we will come back to synchronisation later.

Fortran:

```
INTEGER :: error
CALL MPI_Barrier ( MPI_COMM_WORLD , error )
```

C:

```
int error ;
error = MPI_Barrier ( MPI_COMM_WORLD ) ;
```

MPI_Abort is the emergency stop; you should always call it on MPI_COMM_WORLD, though MPI does not require that. It is not a collective but should stop all processes – and, on most systems, it usually does. Outstanding file output is often lost, and it is far better to stop normally, if at all possible (i.e. all processes should call MPI_Finalize and exit normally). MPI_Abort is the **emergency** stop!

Fortran:

```
INTEGER :: error
CALL MPI_Abort ( MPI_COMM_WORLD , <failure code> , error )
```

C:

```
int error ;
error = MPI_Abort ( MPI_COMM_WORLD , <failure code> ) ;
```

2.9 Practical Use of MPI

I/O in parallel programs is **always** tricky, and it is worse in MPI, because of MPI's portability. Each type of parallel system has different oddities, and implementations are incredibly variable. For now, you should just write to `stdout` or `stderr` (and the default output unit in Fortran, of course); it will work well enough for the examples. Lines may be interleaved with each other in strange ways, but ignore that. We will come back to using I/O later.

You can actually do quite a lot with just the MPI facilities taught so far. The practical exercises start by asking you to write a trivial test program, and then writing a command spawner. The latter is very useful, and there are several around – some practical uses of MPI really **are** that simple! If you have trouble with this, it will be be in using your language, not in MPI – if that is the case, just skip the exercise.

Compiling and running is all very implementation-dependent, of course, but something like this works on most systems:

- Compile and link using `mpif90`, `mpicc` or `mpiCC`, as appropriate.
- Run using "`mpiexec -n <n> <program> [args ...]`", where `<n>` is the number of processes to use.

When using a job scheduler (i.e. queuing system), you may need to put the latter in a script. As a reminder, this course will use MPI only in SPMD mode.