Message-Passing and MPI Programming
Working With MPI

N.M. Maclaren
Computing Service
nmm1@cam.ac.uk
ext. 34761

July 2010

1.11 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, C or C++. You can also program in C++, and call MPI's C interface, of course. Calling MPI using more than one of its Fortran, C or C++ interface 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`, `Interfaces/C` and `Interfaces/C++`. The course does not give the syntax in detail, so check those files when doing the practicals.

1.12 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`); there is one exception in MPI-1 (`MPI_BOTTOM`) and a few more in MPI-2 (e.g. `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. 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 three exceptions in MPI-1, of which `MPI_Wtime` is by far the most important, and only a couple more in MPI-2.

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.

1.13 MPI's C and C++ Interfaces

This section covers the properties that are common to C and C++.

You need to include the statement `#include "mpi.h"`.

All C and a few 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.

1.14 MPI's C Interface

This is also usable from C++, of course.

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 three exceptions in MPI-1, of which `MPI_Wtime` is by far the most important, and only a couple more in MPI-2.

1.15 MPI's C++ Interface

MPI 2.0 introduced a C++ interface in 1997, which significantly better in a great many respects, but MPI 2.2 deprecated it in 2009, and the MPI Forum's current recommendation is to use the C interface. Currently, MPI 3.0 is being worked on. This course will continue to teach both interfaces until the future becomes clearer.

The C++ interface is a "proper" C++ one, and is not just a hacked C one. Almost all names omit the `MPI_` prefix and are declared in the namespace `MPI`; e.g. `MPI_Init` becomes `MPI::Init` and `MPI_TYPE_INT` becomes `MPI::TYPE_INT`. The very few (and rarely used) exceptions are mentioned as they are described.

There have been a few name changes, mostly because MPI-2 has cleaned up its naming conventions – the course will mention them when they are described. The new names will often work in C and Fortran; the old ones are deprecated, and are not in C++. There are some other systematic changes for C++, though; e.g. `Get_` is added before information calls; e.g. the C call `MPI_Comm_size(MPI_COMM_WORLD)` is `MPI::COMM_WORLD.Get_size()` in C++.

Namespace `PMPI` is used for profiling.

Most of the specification of C++'s interface is almost identical to that of its C interface (e.g. the definitions of most constants), and this course will not repeat the information that is unchanged. There are only three major differences, mentioned shortly; minor differences will be described when needed.

This course describes **only** what it needs, and this is particularly relevant to C++; there is a lot in the standard that it does not mention, because it is needed only if you start to use MPI's facilities in complicated ways.

MPI handles are classes in C++ – i.e. C opaque types become C++ classes. Almost all MPI functions are member functions of some suitable class (usually of a communicator class), and a typical use is `MPI::COMM_WORLD.Send(...)`. Classes have a proper C++

class interface, and you must read the details for more than trivial use, especially for creation, deletion, copying and comparison. However, none of those are needed for simple use of MPI.

In C++, `Comm` (the communicator class) is purely a base class, and you always declare one of its derived classes. The only one relevant to this course is `Intracomm`. Some methods are only in `Intracomm` and not `Comm`, though many are moved to `Comm` in MPI-2; you do not need to understand why for this course. So we shall always use `Intracomm` in C++ everywhere that C uses the opaque type `Comm`; C++ inheritance means that will work.

The next big difference is in error handling, which has consequential changes on the interface. Functions do not return error values, but throw a C++ exception instead. There is a new class `Exception` for those.

Functions that deliver a scalar result return that value as the function result. Other functions become void functions in C++. This takes a little getting used to.

The last big difference is the use of references. Essentially all output arguments become references; here, MPI's C++ interface is more like its Fortran one than its C one. For example, `MPI_Init(&argc,&argv)` becomes `MPI_Init(argc,argv)`. That does not apply to array and pointer arguments; e.g. all ones that are transfer buffers stay the same;

1.16 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 ;
}
```

C++:

The arguments are passed to `MPI::Init` as references, not pointers, including the array of argument strings.

```
using namespace std ;
#include "mpi.h"

int main (int argc , char * argv [] ) {
    MPI::Init ( argc , argv ) ;
    MPI::Finalize ( ) ;
    return 0 ;
}
```

Any other valid use of namespace `MPI` is fine; e.g. you could add "`using namespace MPI ;`" and omit all of the `MPI::`. The standard C++ rules for such things apply.

You can also call `MPI::Init` with no arguments, but you might have problems on some implementations, as it is unclear how that works.

Aside: Examples

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

Fortran:

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

C:

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

C++:

```
using namespace std ;
#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 on one implementation, and fails on another.

1.17 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` in **all** of the languages (including C++). 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` (`MPI::Get_version` in C++), 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; the latter is only in MPI-2.

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 ) ;
```

C++:

```
int started , stopped ;
started = MPI::Is_initialized ( ) ;
stopped = MPI::Is_finalized ( ) ;
```

Note that C++ uses different names to Fortran and C (i.e. the addition of `Is_`).

The global communicator is predefined: `MPI_COMM_WORLD` (`MPI::PROC_NULL` in C++). 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` (`MPI::PROC_NULL` in C++), 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.

1.18 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 ) ;
```

C++:

```
int nprocs , myrank ;
nprocs = MPI::COMM_WORLD . Get_size ( ) ;
myrank = MPI::COMM_WORLD . Get_rank ( ) ;
```

Note that C++ uses different names to Fortran and C (i.e. the addition of `Get_`).

You can query the local processor name, and this stores it in a character array of length `MPI_MAX_PROCESSOR_NAME`. This applies to all languages, including C and C++ – it does **not** return a C string or a C++ `basicstring` instance – and, again, C++ passes it as a reference.

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 ] ;
int namelen , error ;
error = MPI_Get_processor_name ( procname ,  & namelen ) ;
```

C++:

```
char procname [ MPI::MAX_PROCESSOR_NAME ] ;
int namelen ;
MPI::Get_processor_name ( procname , namelen ) ;
```

`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 ( ) ;
```

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`.

1.19 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 ) ;
```

C++:

```
MPI::COMM_WORLD . Barrier ( ) ;
```

`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> ) ;
```

C++:

```
MPI::COMM_WORLD . Abort ( <failure code> ) ;
```

1.20 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. 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 examples 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!

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.