

# Programming with MPI

## *Using MPI*

Nick Maclaren

Computing Service

**nmm1@cam.ac.uk, ext. 34761**

May 2008

# Warning

This lecture covers a huge number of minor points  
Including **all** of the ‘**housekeeping**’ facilities

Don't try to remember all details, initially

- Try to remember **which facilities** are included

Refer back to this when doing the practicals

It's a **lot** easier than it looks at first

# Using MPI

- By default, all actual **errors** are fatal  
MPI will produce some kind of an **error message**  
With luck, the **whole program** will then stop

Can ask to do your own **error handling** – see later

- Use **one** interface: **Fortran**, **C** or **C++**  
**C** can be used from **C++**, but don't mix them

Yes, you can mix them – but it's advanced use

# Function Declarations

There are **proformas** for all functions used  
Anything merely mentioned is omitted, for clarity

Interfaces/Fortran

Interfaces/C

Interfaces/C++

The examples don't give the syntax in detail  
Check those files when doing the practicals

# MPI's Fortran Interface (1)

- If possible, include the statement: `USE mpi`
- If not, use: `INCLUDE 'mpif.h'`  
after all `“USE”`s and `“IMPLICIT”`

Note the first is `“mpi”` and the second `“mpif.h”`  
If both fail, usually a usage / installation problem

All MPI names start with `MPI_`

- Don't declare `names` starting `MPI_` or `PMPI_`  
Names `PMPI_` are used for `profiling`

# MPI's Fortran Interface (2)

Boolean values (true/false) are **LOGICAL**

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

Element counts etc. are also plain **INTEGER**

This isn't a problem on any current system

Almost all MPI constants are Fortran constants

One **MPI-1** exception: **MPI\_BOTTOM**

A few more in **MPI-2**, e.g. **MPI\_IN\_PLACE**

Arrays start at **one**, where it matters

# MPI's Fortran Interface (3)

Type-generic arguments are a kludge  
MPI relies on Fortran not noticing them  
Will describe the issues later

For now, just pass arrays of any type  
If the compiler objects, ask for help  
Some guidelines on how in a later lecture

# MPI's Fortran Interface (4)

Handles (e.g. **communicators**) are **opaque** types

[ One you can't break apart and look inside ]

Undocumented and unpredictable **INTEGER** values

Use built-in **equality comparison** and **assignment**

Call MPI functions for **all** other operations

I.e. MPI returns **INTEGER** values as **tokens**

If their **values** match, they are the same **token**

# MPI's Fortran Interface (5)

- Almost all MPI functions are **SUBROUTINES**  
The **final argument** returns an **INTEGER** error code

Success returns **MPI\_SUCCESS** (always zero)

Failure codes are **implementation dependent**

Only **three MPI-1** exceptions: mainly **MPI\_Wtime**

There are only a couple more in **MPI-2**

All **results** are returned through **arguments**

# MPI's Fortran Interface (6)

As people will know, default **REAL** is a disaster  
**DOUBLE PRECISION** is tedious and out-of-date

Start all **procedures**, **modules** etc. with

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

There is a suitable file **Programs/double.f90**  
Ask for help if you don't know how to use it

# MPI's C/C++ Interface (1)

- This is also usable from C++, of course  
C++ people need to listen to this section, too  
I will cover the common C/C++ aspects here

Include the statement: `#include "mpi.h"`

All MPI names start with `MPI_`

- Don't declare `names` starting `MPI_` or `PMPI_`  
Names `PMPI_` are used for `profiling`

## MPI's C/C++ Interface (2)

Boolean values (true/false) are `int`, as usual

Process numbers, error codes etc. are `int`

Element counts etc. are also plain `int`

This isn't a problem on any current system

Type-generic arguments are `void *`

These are called “choice” arguments by MPI

# MPI's C/C++ Interface (3)

Almost all MPI constants are C initialization constants  
NOT usually preprocessor or integer constants

- Cannot use in case, array sizes etc.

Only maximum sizes are preprocessor constants

Arrays start at zero, where it matters

# MPI's C Interface (1)

The next two slides apply only to **C**, not **C++**

**Handles** (e.g. **communicators**) are **opaque types**

Names are set up by **typedef** and are **scalars**

Use built-in **equality comparison** and **assignment**

Call MPI functions for **all** other operations

The main such **opaque types** are:

**MPI\_Comm**, **MPI\_Datatype**, **MPI\_Errhandler**,  
**MPI\_Group**, **MPI\_Op**, **MPI\_Request**,  
**MPI\_Status**

# MPI's C Interface (2)

- Almost all MPI functions return an **error code**

This is the **function result** as an **int**

Can ignore it, if using **default** error handling

Success returns **MPI\_SUCCESS** (must be zero)

Failure codes are **implementation dependent**

Only **three MPI-1** exceptions: mainly **MPI\_Wtime**

There are only a couple more in **MPI-2**

- All **results** are returned through **arguments**

# MPI and C++

MPI 2.0 introduced a C++ interface in 1997

It's significantly better in a great many respects

However, MPI 2.2 deprecated it in 2009

Its recommendation is to use the C interface

Currently, MPI 3.0 is being worked on

- This course will teach both interfaces

It's unclear what is going to happen in this area

# MPI's C++ Interface (1)

A “proper” C++ interface, not just a hacked C one

Include the statement: `#include "mpi.h"`

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`

# MPI's C++ Interface (2)

Some name changes – will mention when needed  
Mostly because **MPI-2** has cleaned up its naming

The new names will often work in **C** and **Fortran**  
The old ones are **deprecated**, and are not in **C++**

Some other systematic changes, though

E.g. **Get\_** is added before information calls

**C:** `MPI_Comm_size(MPI_COMM_WORLD)`

**C++:** `MPI::COMM_WORLD . Get_size()`

Namespace **PMPI** is used for **profiling**

# MPI's C++ Interface (3)

- Most is almost identical to the C interface  
E.g. the definitions of most constants
- I won't repeat the information that is unchanged

Only three major differences (in a moment)  
Minor differences will be described when needed

- This course describes **only** what it needs

# MPI's C++ Interface (4)

MPI handles are classes in C++

I.e. C opaque types become C++ classes

Almost all MPI functions are member functions

E.g. `MPI_Send` becomes `MPI::Comm . Send`

A typical use is `MPI::COMM_WORLD.Send`

- Classes have a proper C++ interface

You must read the details, for more than trivial use

Esp. creation, deletion, copying and comparison

# MPI's C++ Interface (5)

In C++, Comm is purely a base class

You always declare one of its derived classes

- The only one relevant here is Intracomm

Some methods are only in Intracomm

Though many are moved to Comm in MPI-2

[ Don't bother understanding why, for now ]

- So we shall always use Intracomm in C++

Everywhere that C uses the opaque type Comm

C++ inheritance means that will work

# MPI's C++ Interface (6)

The next big difference is in error handling  
That has consequential changes on the **interface**

**Functions** do not return **error values**  
Instead, they throw a **C++ exception**  
There is a new **class Exception**

Functions that deliver a **scalar** result  
return that value as the **function result**  
**Others** become **void functions** in **C++**

# MPI's C++ Interface (7)

The last big difference is the use of **references**

Essentially all **output arguments** become **references**

Here, MPI's **C++** is more like **Fortran** than **C**

**MPI\_Init(&argc,&argv) ⇒ MPI\_Init(argc,argv)**

That doesn't apply to **array** and **pointer** arguments

E.g. all ones that are **transfer buffers** stay the same

# More on Interfaces

- That is all you need for now

We will return to language interfaces later

- Advanced language facilities to avoid
- Interfaces for advanced MPI programming
- Performance and optimisation issues

# Starting and Stopping

- For now, we will ignore error handling

All processes must start by calling `MPI_Init`  
And, normally, finish by calling `MPI_Finalize`

- These are effectively **collectives**  
Call both at predictable times, or risk confusion
- You **can't** restart MPI after `MPI_Finalize`  
`MPI_Init` must be called **exactly** once

# Fortran Startup/Stopping

Fortran argument decoding is behind the scenes

```
USE double
```

```
USE mpi
```

```
IMPLICIT NONE
```

```
INTEGER :: error
```

```
CALL MPI_Init ( error )
```

```
CALL MPI_Finalize ( error )
```

```
END
```

If that doesn't work, see the MPI documentation

- Though you will probably need to ask for help

# C Startup/Stopping

`MPI_Init` takes the **addresses** of **main**'s arguments

- You **must** call it before **decoding** them

Some **implementations** change them in `MPI_Init`

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

```
int main (int argc , char * argv [] ) {  
    MPI_Init ( & argc , & argv ) ;  
    MPI_Finalize ( ) ;  
    return 0 ;  
}
```

# C++ Startup/Stopping (1)

```
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 OK  
E.g. you could add “`using namespace MPI ;`”  
and omit all of the `MPI::`

# C++ Startup/Stopping (2)

The arguments are **references** not **pointers**

You can also call **MPI::Init** with no **arguments**

- Watch out with that – I am not sure how it works

# Aside: Examples

I will omit the following statements, for brevity:

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

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

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

Include them in any “module” where you use MPI

**Don't** rely on implicit declaration

# Version Numbers

MPI 1.2 and up provide **version number** information

- Not needed for simple use, as in this course  
All versions of MPI are essentially compatible

Constants **MPI\_VERSION**, **MPI\_SUBVERSION**

in **ALL** of **Fortran**, **C** and **C++**

Set to **1, 3** for **MPI 1.3** or **2, 2** for current **MPI-2**

There is also a **function** **MPI\_Get\_version**

**MPI::Get\_version** in **C++**

Which can be called even before **MPI\_Init**

# Testing MPI's State (1)

You can test the state of MPI **on a process**

- This is **needed** only when writing **library code**  
[ **MPI\_Finalized** is only in **MPI-2** ]

**Fortran** example:

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

# Testing MPI's State (2)

C example:

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

C++ example:

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

Note C++ uses different names to Fortran and C

# Global Communicator

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**

# Process Rank

The **rank** is the **process**'s index  
always within the context of a **communicator**

A **rank** is an integer from **0** to **<n>-1**

Yes, this applies to **Fortran**, too

There is one predefined **rank** constant:

**MPI\_PROC\_NULL** – no such **process**

**MPI::PROC\_NULL** in **C++**

Don't assume this is negative – or that it isn't

We shall describe the use of it when relevant

# Information Calls (1)

`MPI_Comm_size` returns the number of processes

`MPI_Comm_rank` returns the local process number

Fortran example:

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

Remember `&` means continuation in Fortran

# Information Calls (2)

C example:

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

C++ example:

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

Note the addition of `Get_` in C++

## Information Calls (3)

You can query the local **processor name**

A **string** of length **MPI\_MAX\_PROCESSOR\_NAME**

**Fortran** example:

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

# Information Calls (4)

C example:

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

C++ example:

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

The second argument is a **reference** not a **pointer**

# Information Calls (5)

**MPI\_Wtime** gives **elapsed time** (“wall-clock time”)  
**Seconds** since an unspecified **starting point**

The **starting point** is fixed for a **process**  
Doesn't change while the **process** is running

I have seen **start of process**, **system boot time**,  
**Unix epoch** and **00:00 Jan. 1st 1900**

**MPI\_Wtick** similar but gives timer **resolution**  
Few people bother – but it's there if you want it

# Information Calls (6)

## Fortran:

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

## C:

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

## C++:

```
double now ;  
now = MPI::Wtime ( ) ;
```

# Information Calls (7)

Anywhere from **MPI\_Init** to **MPI\_Finalize**

They are all purely **local** operations

Use them as often as you need them

**MPI\_Comm\_size** same result on all **processes**

- Others may give **different** ones on each **process**
- That includes **MPI\_Wtime**'s starting point  
As well as the value returned from **MPI\_Wtick**

# Barrier Synchronisation (1)

**MPI\_Barrier** synchronises all **processes**

They all wait until they have all entered the call

Then they all start up again, independently

- The **only** collective that **synchronises**

We will come back to this later

# Barrier Synchronisation (2)

Fortran example:

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

C example:

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

C++ example:

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

# Abandoning All Hope (1)

`MPI_Abort` is the emergency stop

- Always call it on `MPI_COMM_WORLD`

Not a **collective** but should stop all **processes**  
and, on **most** systems, it **usually** does ...

- Outstanding **file output** is often lost

Far better to stop normally, if at all possible

I.e. all **processes** call `MPI_Finalize` and exit

- `MPI_Abort` is the **emergency** stop

# Abandoning All Hope (2)

## 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> ) ;
```

# Lookahead to I/O

I/O in parallel programs is **always** tricky  
It's worse in MPI, because of MPI's portability  
Each **type of parallelism** has different oddities

- For now, just write to **stdout** or **stderr**  
And the default output in **Fortran**, of course  
It will work well enough for the examples
- We will come back to using I/O later

# First Practical

You can actually do quite a lot with just these

Start by writing a trivial test program

Then writing a **command spawner**

This is very useful, and there are several around

- Yes, some practical uses **ARE** that simple!

Use any language you like, that can call MPI

Examples will be in **Fortran**, **C** and (bad) **C++**

# Compiling and Running

This is all very **implementation-dependent**, of course  
But, on most systems, do something like this:  
Compile and link using **mpif90**, **mpicc**, **mpiCC**

Run using “**mpiexec -n <n> <program> [args ...]**”  
**<n>** is the number of **processes** to use

When using a **job scheduler (queuing system)**  
you may need to put the latter in a script

- This course will use MPI only in **SPMD** mode

# PWF Usage

Unfortunately, the PWF uses single core systems  
All the examples will work, but very slowly

I have set up OpenMPI, which has a few bugs  
You need to ignore a few warnings – but only those

# Ignorable Warnings

## Fortran:

Warning: Procedure '...' called with an implicit interface at (1)

For most of the **MPI** calls – but only those

## C++:

/usr/local/OPENMPI/include/openmpi/ompi/mpi/cxx/comm\_inln.h:...

warning: unused parameter '...'

Regrettably, there are quite a lot of these

## C:

/usr/local/OPENMPI/include/mpi.h:220: warning:

ISO C90 does not support 'long long'

# Instructions

If running **Microsoft Windows**, **CTRL-ALT-DEL**

Select **Restart** and then **Linux**

Log into **Linux** and start a shell and an editor

Create programs called **prog.f90**, **prog.c**, **prog.cpp**.

- Run by typing commands like  
**mpif90 prog.f90**, **mpicc prog.c**, **mpiCC prog.cpp**  
**mpiexec -n 4 a.out**
- Analyse what went wrong
- Fix bugs and retry