Programming with MPI

Miscellaneous Guidelines

Nick Maclaren
Computing Service

nmm1@cam.ac.uk, ext. 34761

March 2010
Summary

This is a miscellaneous set of practical points
Over-simplifies some topics in extra lectures
Mostly not about MPI, but languages and systems

Done this way, because course has become too long

• Remember that everything here is a half truth
  Good as a guideline, but no more than that

• Remember extra lectures if any weird problems
  Or you use a facility in a non-trivial way
Composite Types

So far, mainly contiguous arrays of basic types
n-D arrays stored in array element order
Fortran 77, C and C++ are all similar

Advanced collectives allow one level of separation

- Fortran 90 arrays not always contiguous
  An N-D array may have N levels of separation
- C and C++ have structures and pointers
  And “objects” are often built using them
- Fortran 90 and C++ have “classes”
Shortcuts (Hacks)

In a simple case, you can put the code inline
Or pack multiple transfers into one function
• Do whichever is simplest and cleanest

1: Pack up your data for export
2: Do the actual data transfer
3: Unpack the data you have imported

OR

1: Transfer the first simple array
2: Transfer the second simple array
   ...

n: Rebuild them into a consistent structure
C++ PODs and C structs are easy. Use as array of `sizeof` bytes (type `MPI_BYTE`).

But you must follow these rules:

- Do it only when using the same `executable`.
- Do it only between `identical` types.
- Don’t do it if they contain `pointers`.
- Don’t do it if have any `environment data`.

And watch out for `variable sized structs`.

---

**C++ PODs**

PODs (Plain Old Data) are a feature of the C++ language that allows for efficient storage and transfer of data. They are essentially structs that do not contain any pointers or references, ensuring that they can be sent over a network or shared between processes without the overhead of copying pointers or referencing other data.

In the context of MPI (Message Passing Interface), using PODs allows for efficient communication between processes because the data can be sent as a single block of memory, without the need to serialize or deserialize complex data structures.

**Usage and Rules:**

- **Easy Use:**
  - PODs and similar C structs are easy to use in C++.
  - They can be used as an array of `sizeof bytes` when using the `MPI_BYTE` type.

- **Guidelines:**
  - **Do it only when using the same executable.**
  - **Do it only between identical types.**
  - **Don’t do it if they contain pointers.**
  - **Don’t do it if have any environment data.**

- **Variable Sized structs:**
  - Be aware of variable sized structs, as they can lead to unexpected behavior when using PODs in an MPI context.

---

**Programming with MPI – p. 5**
C, C++ and POSIX

Some C, C++ and POSIX features are toxic
Often cause chaos to almost all other interfaces
Can be used safely, but only by real experts

<stdio.h>, <setjmp.h> and C++ exceptions
POSIX threading, signal handling, scheduling
timer control, alarm, sleep, ...

More detail in extra lectures, but try to avoid them
Fortran Assumed Shape Arrays

Good Fortran 90 uses assumed shape arrays
MPI uses assumed size arrays (i.e. Fortran 77)

Generally requires a copy, on call and return
Ignore this if not a performance problem
See Fortran course for some more details

• Only real problem is with non-blocking transfers
Convert to Fortran 77 (e.g. explicit shape)
In a common parent of both send/receive and wait
Fortran Type Checking

A routine must use compatible arguments everywhere MPI buffers can be of any supported type So the compiler may object to your use of them

If compiler objects to buffer argument type use:

- Keep all calls in one module the same
  Fortran compilers rarely check over all program

- Or write trivial wrappers in external procedures
  E.g. My_Send_Integer and My_Send_Double
Fortran Derived Types

Fortran 2003 supports $\text{BIND}(C)$ for interoperability
$\text{BIND}(C)$ derived types are like C++ PODs

In general, don’t treat them like PODs
And never do if they contain allocatable arrays

- No option but to transfer them as components
  Tedium, messy, but not difficult

- Don’t assume $\text{SEQUENCE}$ means C–compatible
  Has its uses for MPI, but not within this course
Debugging vs Tuning

In practice, these overlap to a large extent

• Tuning MPI is more like tuning I/O than code

Many performance problems are logic errors
E.g. everything is waiting for one process

Many logic errors show up as poor performance

• So don’t consider these as completely separate
Partial Solution

• Design primarily for **debuggability**

**KISS** – Keep It Simple and Stupid

This course has covered many MPI–specific points

See also *How to Help Programs Debug Themselves*

• Do that, and you rarely need a **debugger**
  Diagnostic output is usually good enough

• Only **then** worry about **performance**
MPI Memory Optimisation

The examples waste most of their memory
Here are some guidelines for real programs:

• Don’t worry about small arrays etc.
  If they total less than 10%, so what?

• For big ones, allocate only what you need
  For example, for \texttt{gather} and \texttt{scatter}

• Reuse large buffers or free them after use
  Be careful about overlapping use, of course
MPI Performance

• Ultimately only \textit{elapsed time} matters
  The \textit{real time} of program, start to finish

• All other measurements are just \textit{tuning tools}

This actually simplifies things considerably

• You may want to analyse this by \textit{CPU count}
  Will tell you the \textit{scalability} of the code
Design For Performance (1)

Here is the way to do this

• Localise all major communication actions
  In a module, or whatever is appropriate for you
  Keep its code very clean and simple

• Don’t assume any particular implementation
  This applies primarily to the module interface
  Keep it generic, clean and simple

• Keep the module interfaces fairly high level
  E.g. a distributed matrix transpose
Design For Performance (2)

Use the highest level appropriate MPI facility
• E.g. use its collectives where possible
Collectives are easier to tune, surprisingly

Most MPI libraries have had extensive tuning
• It is a rare programmer who will do as well

mpi_timer implements MPI_Alltoall many ways
Usually, 1–2 are faster than built-in MPI_Alltoall
Not often the same ones, and often by under 2%
Design For Performance (3)

- Put enough **timing calls** into your module
  Summarise time spent in MPI and in computation

- Check for other **processes** or **threads**
  Only for ones **active** during MPI **transfers**

Now look at the timing to see if you have a problem

- If it **isn’t** (most likely), do **nothing**

- Try using only **some** of the **cores** for MPI
  It’s an easy change, but may not help
High-Level Approach (1)

Try to minimise inter-process communication
There are three main aspects to this:

- **Amount of data** transferred between processes
  Inter-process bandwidth is a limited resource

- **Number of transactions** involved in transfer
  The message-passing latency is significant

- **One process** needs data from **another**
  May require it to **wait**, wasting time
High-Level Approach (2)

Partitioning is critical to efficiency
That will be described in the next lecture

You can bundle multiple messages together
Sending one message has a lower overhead

You can minimise the amount of data you transfer
Only worthwhile if your messages are large

You can arrange all processors communicate at once
Can help a lot because of progress issues
Bundling

On a typical cluster or multi-core system:
Packets of less than 1 KB are inefficient
Packets of more than 10 KB are no problem

Avoid transferring a lot of small packets
⇒ Packing up multiple small transfers helps
But only if significant time spent in them

• Remember integers can be stored in doubles
Timer Synchronisation (1)

This means synchronisation across processes. I.e. are all results from MPI_Wtime consistent?

Almost always the case on SMP systems. Will often be the case even on clusters.

- Generally, try to avoid assuming or needing it. Rarely compare timestamps across processes.

- If you use only local intervals, you are OK. Time passes at the same rate on all processes.
Timer Synchronisation (2)

Beyond that is a job for real experts only

Parallel time is like relativistic time
Event ordering depends on the observer

There is a solution in directory Posixtime
Functions to return globally consistent time

I wrote this for a system with inconsistent clocks
Please ask about synchronisation if you need to
This means language, POSIX and Microsoft I/O

There are serious problems – not because of MPI Caused by the system environment it runs under

• Will cover most common configuration only

If it doesn’t apply, look at the extra lecture
Or ask your administrator to help you
There are two, very different, classes of file
• Normal named and scratch files
• stdin, stdout and stderr
Former local to process – latter global to program

Problems are caused by the system environment
E.g. clusters of distributed memory systems
Or shared file descriptors on SMP systems

• These issues are NOT specific to MPI
Other parallel interfaces have the same problems
Shared File Access (1)

- Assume all *processes* share a *filing system* directly, using *POSIX*, or indirectly, using *NFS* or with the *Microsoft* and other equivalents.

- And that all *processes* share a *working directory* with luck, that’s *controllable* or your *home directory*.

  The details are very system-dependent, as usual.

- Here are some rules on how to use *files* safely.
Shared File Access (2)

- Always use write-once or read-many
  That applies to the whole duration of the run

- All updates and accesses must be considered
  Including any that are done outside MPI

I.e. if a file is updated at any time in the run
  only one process opens it in the whole run

Any number of processes may read a file
  provided that no process updates it
Directories

- Regard a directory as a single file (it is)

If you change it in any way in any process
- Don’t access it from any other process
Creating a file in it counts as a change, of course

If you do, a parallel directory listing may fall over!
Listing a read–only directory is safe

- Can create and delete separate files fairly safely
[ But not under Microsoft DFS, I am afraid ]
Create and delete any single file in one process
Scratch Files

Don’t assume where scratch files go
That statement applies even on serial systems
It is even more complicated on parallel ones

It’s common to have shared working directories
But separate, distributed scratch directories

• Just a warning – clean code rarely has trouble
Standard Units

Issues arise from implementation details

- Almost always show up with output
  Probably just because almost all programs use it!

- It is an almost unbelievable can of worms
  Don’t even try to program round the problems
  Only solution is to bypass the issue entirely

- These issues are NOT specific to MPI
  Other parallel interfaces have the same problems
Avoiding the Mess

The “right” solution is also the simplest. Only root process does stdin/stdout I/O. See the extra I/O lecture for the full details on this.

It does all the reading from stdin.
It broadcasts or scatters it to the others.

It gathers all of the output from the others. And then it writes it to stdout.

This can also be done for file I/O.
Handling Standard I/O

You have learnt all of the techniques you need
Or look at the extra I/O lecture for details
It has quite a lot of worked examples

If root process both handles I/O and computation
I do not recommend doing it asynchronously
It’s extremely hard to make such code reliable

- Code the I/O transfers as a collective
  That’s not too difficult to debug and tune
• Just write to `stderr` or equivalent
  Fortran users may need to use FLUSH

It may well get mangled (reasons given above)
It may get lost on a crash or `MPI_Abort`
But it’s simple, and errors are rare, right?

Same applies to `stdout`, with some programs

• Beyond that, use a dedicated I/O process
  Just as we described for `stdout` above
Practicals

There are some **practicals** on **I/O** handling
Mainly **spooling** it through the **root** process

You have already learnt all of the **techniques** needed

- You are likely to **need** to be able to do this

Plus a trivial one on transferring **structures**