

Programming with MPI

Debugging, Performance and Tuning

Nick Maclaren

nmm1@cam.ac.uk

March 2008

Available Implementations

Two open source versions – MPICH and OpenMPI
Most vendors have own, inc. Intel and Microsoft

Wide range of tuning and debugging tools
Mostly commercial, but not all

Or can use built-in profiling interface
Easy to use and can help with debugging

- Not ideal, but consensus is pretty good
- This lecture is just the general principles

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

Classes of Problem (1)

- Most common is breach of **language standard**
Parallelism exposes aspects that you never realised

Generally, **debuggers** and other tools don't help
The aspects are usually subtle ones of **semantics**
Most books and Web pages are **very** misleading

This is why my courses often seem too pedantic
I warn about issues that you hope you don't see
Contact your supervisor for advice

Classes of Problem (2)

- Second most common is **logic errors**

You wrote what you meant, but that doesn't work

E.g. distributing **data/work** between **processes**

Debuggers and other tools help only a little

You need to find **how** things went wrong, and **why**

Recommendations in this course are for safety

They should help to minimise these

- But this class of problem is unavoidable

Classes of Problem (3)

- Least common is MPI coding errors
E.g. a receive with no matching send

Parallel debuggers help a lot with this

But do what I say, and such bugs should be rare

Most programmers don't use parallel debuggers
Some others find them very helpful



My Hobby-Horse (1)

A good language would prevent 90% of errors
though only a few logic errors, of course

A restricted subset of MPI would allow checking
Would then be easy to detect many common errors
It would only help with the ones entirely in MPI

Most modern languages are complete ****

As far as error detection and prevention go
Ada is the main exception, possibly Python
Fortran 2003 goes a little way towards that

My Hobby-Horse (2)

We can agree that **flexibility** and **features** are good

Modern dogma is that **restrictions** are always bad
and **languages** should define only correct code
and **performance** always trumps **correctness**

- I am a heretic — that is totally false

Checked restrictions are the programmer's friend
Longer to get **running**, but quicker to get **working**

Which is why I say to impose your own **restrictions**

My Hobby-Horse (3)

Programming MPI shows this **very** clearly

Very hard to debug even a known correct algorithm
by doing it using general **point-to-point**

Back off, constrain the design (e.g. with barriers)
and it's hard to tell where the problem was

I believe that a **language** could do this for you
Would be completely different from current ones
Hoare's BSP uses this approach

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**

Parallel Debuggers (1)

These exist, and some people like them
None worth using are available for free
Please tell me if you find an exception

I have never more than dabbled with them
Totalview is the best-known one
Intel is also reported to be good

There are others, especially from **HPC** vendors

Parallel Debuggers (2)

These **must** be integrated with

- The **compiler** for your language
- The MPI **implementation**
- The **job scheduler**

That is not easy to arrange

Unless you are a vendor that sells all of them!

Many vendors sign up to **Etnus** (**Totalview**)

Parallel Tools

I haven't looked at many of these
[Intel](#) bought out [Pallas](#) ([Vampir](#))

Some [open-source](#) ([free](#)) ones might be OK
They don't have the same problems as [debuggers](#)

I have written a (not very good) one
It's quite easy to do, in many cases

Interactive Serial Debuggers

These are, by and large, useless for MPI

- Often difficult to run them on MPI **processes**
Usually needs administrator-level hacking

- Often interfere with each other, badly
May cause MPI to lock up solid or fail
Debugger may display wrong results, or crash

Non-blocking transfers are a major problem
Asynchronous progress is even worse

Debugging From Dumps (1)

This is usually much more successful

- Useful for when an MPI **process** crashes
Do that just as in the serial case
- You can usually force a **dump**, too
Just as you can in a serial program
- And you can often get one of each **process**
And compare them to see where they have got to

Debugging From Dumps (2)

- Biggest problem is getting the **dump**
System-dependent, and may need administrator

- All **dumps** may be written to file '**core**'
Bad news if all in the same **directory**

Can often avoid that by calling **chdir**
Or can configure to dump to '**core.<pid>**'

- One **dump** per **process** may be too big
There are bypasses, but contact your administrator

Debugging From Dumps (3)

- Main problem is **not** getting any **dump**
Or, occasionally, getting dump of wrong **process**
And, **far too often**, getting diagnostic **no stack**

May be a shell or system feature (e.g. **ulimit**)

May be a **compiler** or MPI implementation one

May be a **PATH**-related configuration issue

- Generally soluble, but no good rules
Have to investigate problem, and deal with it

Built-in MPI Facility (1)

MPI provides a built-in facility for tuning
It's useful for debugging, and some tools use it

All functions called **MPI_...** are wrappers
They call identical ones called **PMPI_...**

Exceptions are **MPI_Wtime** and **MPI_Wtick**
Plus a few **MPI 2** ones we haven't covered

Built-in MPI Facility (2)

All you do is to write your own **MPI_...** ones
Calling the **PMPI_...** ones to do the work
You can put in any tracing and checking you like

There is an example in **Wrappers/Wrappers.c**
It supports **only** original **MPI 1**

It worked very well in simple **tracing** mode

Its **scaling** wasn't entirely successful
It conflicted with the MPI **progress engine**

Built-in MPI Facility (3)

- You don't have to wrap all of the MPI functions
Wrapping the ones that you use is enough
- Keep the wrapper functions in a separate file
Then you can include them or not as you wish

It really is very easy to use

Function `MPI_Pcontrol` controls `profiling`

However, it is almost completely unspecified
It's really just a hook for a specification

MPI Memory Optimisation (1)

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 `gather` and `scatter`
- Reuse large buffers or free them after use
Be careful about overlapping use, of course

MPI Memory Optimisation (2)

If the above doesn't solve your problem:

- Scatter large structures across processes

This is the dreaded **data distribution** problem

- Read and write them in smaller sections

For very large amounts of data, it's no slower

- Watch out for **memory fragmentation**

That has nothing to do with MPI as such

MPI Memory Optimisation (3)

Used to be normal practice up to the 1970s
64 KB was often a lot of memory ...

It's a pain in the neck to program
Please ask for help if you need to do it

Generally, avoid optimising for memory
Don't waste excessive amounts, of course
But concentrate on writing clean code

- MPI itself is rarely an issue

MPI Performance

- Ultimately only **elapsed time** matters
The **real time** of program, start to finish
- All other measurements are just **tuning tools**

This actually simplifies things considerably
See later under **multi-core systems** etc.

- You may want to analyse this by **CPU count**
Will tell you the **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

Design For Performance (4)

- Going further, you have only one module to tune
And its code is **clean** and **simple**!
- It will also help an expert help you
Won't have to start by reverse engineering code

The **higher level** the **module interface** is
the more scope that you have for tuning

E.g. attempting to use **non-blocking** transfers
may be impossible with a **low level interface**

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 can be critical to efficiency

Some principles of that are mentioned later

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 processes 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**

Advanced Tuning

This includes even use of **non-blocking** transfers
Reasons for that are the **progress** issues

They are worth learning to avoid **deadlock**
Can help with **performance** on **some** systems

- This course is not going to cover tuning them
Or any other such advanced tuning

Tuning I/O is more **system-specific** than MPI

Elapsed Time (1)

Isn't `MPI_Wtime` the answer? – er, no

Times don't always mean what you think
Will describe this shortly, but it's complicated

Need to **design** program for **reliable** timing
Design methodology can also help with debugging

But some programs don't match it very well
It is very hard to measure the time in those

Elapsed Time (2)

Any outstanding **transfers** make times unreliable
These are ones that have not been **received**
and **completed** for **non-blocking**

Note that a **blocking send** remains outstanding
even after the **send** call returns

You can call **MPI_Wtime** even at such times
But interpreting its value can be extremely hard

Elapsed Time (3)

Simplest use that gives understandable times:

- Receive and complete all transfers across the whole communicator, of course [Collectives will do this automatically]
- Call `MPI_Barrier` on the communicator
- Call `MPI_Wtime` in any or all processes

All calls show roughly the same elapsed time

Elapsed Time (4)

Beyond that, things can get a bit complicated

Remember **collectives** are not **synchronised**

And that **point-to-point** can overlap them

This lecture now describes this in more detail

Progress (1)

MPI has an arcane concept called “**progress**”

Good news: needn't understand it in detail

- No valid MPI program can get stuck (hang)
I.e. MPI doesn't allow any “**deadly embraces**”

An **implementation** must always make **progress**

A **programmer** must not make that impossible

There are a few restrictions to ensure that is so

- Write sanely, and you will never notice them
Mistakes will happen, but fix the bug in **your** code

Progress (2)

MPI does not specify how it is implemented
Progress can be achieved in many ways

Bad news: do need to understand these issues

- All valid MPI programs will work in all cases
But it changes the most efficient coding style

Will describe a few of the most common methods
And indicate the main consequences of them
But will start by saying how to proceed

Processes vs CPUs

- More MPI processes than cores is Bad News
Some systems seem to crawl into a hole and die!
- Shared systems will have other threads running
- And remember MPI may have hidden threads

When setting MPI tuning parameters:

- Be careful with spin loops for waiting
Use only if each MPI process has its own core
Never use spin loops on a shared system

Multi-Core Systems

Use of **SMP** systems was described earlier

If using **SMP libraries** , **OpenMP** or **threading**

- Use only **one** MPI **process** per **system**
- Otherwise, write purely **serial** executables
And use multiple MPI **processes** per **system**

Either works – the combination doesn't

Serial MPI Processes on SMP

Use **total** core count for calculations

I.e. **cores/socket** times **sockets/system**

- Consider using only some **CPUs** for MPI
Often **increases** the total performance
- Only way to find out is to time two runs

First reason is that it stresses the **memory** less

More codes are **memory-bound** than **CPU-bound**

Second is that it may help **asynchronous progress**

As mentioned, can include **physical transfer**

Collectives (1)

They may start transferring as soon as they can
And may leave as soon as they have finished

- You can stop that by using `MPI_Barrier`
That can **sometimes** improve efficiency

It always makes initial tuning a lot easier
Calls to `MPI_Wtime` become reliable

Collectives (2)

```
error = MPI_Barrier ( MPI_COMM_WORLD ) ;  
start = MPI_Wtime ( ) ;  
error = MPI_Alltoall ( . . . ) ;  
error = MPI_Barrier ( MPI_COMM_WORLD ) ;  
total = MPI_Wtime ( ) - start ;
```

- After initial tuning, start removing the **barriers**
See if it runs faster with or without them
Remember that the **barriers** take time, too
- Tuning like this is generally quite easy

Behind The Scenes (1)

MPI does not specify **synchronous** behaviour
All **transfers** can occur **asynchronously**
And, in theory, so can almost all other actions

Transfers can overlap **computation**, right?
Unfortunately, it isn't as simple as that

Many **I/O mechanisms** are often **CPU bound**
TCP/IP over **Ethernet** is often like that

Will come back to this in a moment

Behind The Scenes (2)

MPI transfers also include data management
E.g. scatter/gather in MPI derived datatypes

InfiniBand has such functionality in hardware
Does your implementation use it, or software?

Does your implementation use asynchronous I/O?
POSIX's spec. (and .NET's?) is catastrophic

May implement transfers entirely synchronously
Or may use a separate thread for transfers

Eager Execution

This is one of the mainly **synchronous** methods
Easiest to understand, not usually most efficient

All MPI calls complete the operation they perform
Or as much of it as they can, at the time of call

- **MPI_Wtime** gives the obvious results
Slow calls look slow, and fast ones look fast
- Often little point in **non-blocking transfers**
But see later for more on this one

Lazy Execution

This is one of the mainly **synchronous** methods
Just not in the way most people expect

Most MPI calls put the operation onto a **queue**
All calls complete **queued** ops that are “**ready**”

- **MPI_Wtime** gives fairly strange results
One MPI call often does all of the work for another
The **total time** is fairly reliable, though

Possibly the most common **implementation** type

Asynchronous Execution

MPI calls put the operation onto a **queue**
Another **process** or **thread** does the work

- **MPI_Wtime** gives very strange results
Need to check the time used by the **other thread**
- Start by not using all **CPUs** for MPI
Further tuning is tricky – ask for help

Fairly rare – I have seen it only on **AIX**
May become more common on **multi-core** systems

Asynchronous Transfers

Actual data transfer is often asynchronous
E.g. TCP/IP+Ethernet uses a kernel thread

- One critical question is if it needs a CPU
If so, using only some CPUs may well help (a lot)
- Sometimes, non-blocking transfers work better
Even on implementations with eager execution
- And sometimes, blocking transfers do
Even with asynchronous execution

Reminder

- Localise all major communication actions
In a high level module, or whatever is appropriate
- Do nothing if it performs well enough
- Consider using only some of the CPUs
- Do simple, high-level, tuning (as above)
Often just by adding or removing barriers
- Only then, worry about fine-tuning your code
E.g. comparing blocking and non-blocking