Message-Passing and MPI Programming

Miscellaneous Guidelines

N.M. Maclaren

nmm1@cam.ac.uk

July 2010

9.1 Introduction

This is a miscellaneous set of practical points, that over-simplifies some of the topics covered in the extra lectures. Most of these are not actually about message-passing or MPI, but about problems with its interaction with programming language and operating system specifications and implementations. A summary is given at this stage, because the course has become too long, and there was too much non-MPI material.

- Remember that everything here is a half truth hiding a serious, practical problem; i.e. it is good as a guideline, but no more than that.
- You should remember that there is more information in the extra lectures (and the actual standards), if you hit a problem or use a facility in a non-trivial way.

9.2 Composite Types

So far, we have considered mainly contiguous arrays of basic types. Multi-dimensional arrays are stored in array element order, in both Fortran 77 and C, though which subscript varies fastest differs. The advanced collectives allow one level of separation, which can sometimes be useful. However:

- Fortran 90 assumed shape arrays are not always contiguous, and an N-dimensional array may have N levels of separation. You do not need to know the details.
- Fortran 90, C and C++ have structures and pointers (though Fortran pointers are less commonly needed or used), and structured "objects" are often built using them.
- Fortran 90 and C++ have "classes", which call procedures for built-in operations.

In a simple case, you can put the code inline and transfer each contiguous unit of data separately, or you can pack multiple transfers into one function. You should do whichever is simplest and cleanest. I.e.:

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

9.3 C and C++

C++ *PODs* and similar C structs are easy; you just use `sizeof` to calculate their use, and transfer as an array of bytes using the MPI datatype `MPI_BYTE`. But you **must** follow these rules:

- Do it only when using the **same executable**
- Do it only between **identical** types, not just similar ones
- Do **not** do it if they contain pointers
- Do **not** do it if have any environment data

Environment data includes anything that may be process-specific, which is a little more general than you might expect. And be careful to get the size right when using C99 variable sized structures. The above rules may seem excessive, especially that of using identical types (which essentially means compatible ones in C terms), but are necessary to avoid certain arcane C and C++ problems.

There are some C, C++ and POSIX features that are thoroughly toxic, and often cause chaos to almost all other interfaces; MPI is no exception. They can be used safely, but only by *real* experts, and very few people have that level of skill. These include `<signal.h>`, `<setjmp.h>`, `<fenv.h>` (and the C++ equivalents), as well as much of POSIX: threading, signal handling, scheduling, timer control, `alarm`, `sleep` and so on. These is some more information in the extra lectures, but generally try to avoid using those features. It is also extremely easy to break MPI's rules with C++ exception handling, such as by releasing an in-use non-blocking buffer; that will cause chaos.

9.4 Fortran 90

Good Fortran 90 uses assumed shape arrays, and MPI 3 supports those; MPI 3 supports them properly, but is not covered here. MPI 1 and 2 use assumed size arrays, because they use a Fortran 77 interface. This generally requires a copy on call and return, but you should ignore that if it does not cause a performance problem. The course *Introduction to Modern Fortran* provides more details. The only real problem is with non-blocking transfers, and the following rule is almost always safe:

- Convert them to a Fortran 77 form (i.e. explicit shape or assumed-size) in a common parent of both the send (or receive) and the wait. I.e. do not convert them from assumed shape to any of the Fortran 77 forms at any time between starting the transfer and waiting for its completion.

If your compiler objects to your uses of the buffer argument type because you are calling the same routine twice with incompatible arguments; this is also fixed in MPI 3. There are a couple of simple hacks to avoid the diagnostic; either will usually work, so use whichever is easier.

- Keep all calls in one module (or file) the same (e.g. use only `INTEGER` buffers or only `REAL(KIND=KIND(0.0D0))` ones). Fortran compilers rarely check over the whole program, but often do within a single module or file.
- Write trivial wrappers in external procedures, put each in a separate file, compile them separately, and link them in (e.g. `My_Send_Integer` and `My_Send_Double`).

Fortran 90 selectable precisions have already been described.

Fortran 2003 supports `BIND(C)` for C interoperability, and `BIND(C)` derived types can be handled just like C++ *POD*s, using the facilities in the `ISO_C_BINDING` intrinsic module.

In general, do not treat Fortran derived types like C++ *POD*s, and never do if they contain allocatable arrays. You have no option but to transfer them as components, which is tedious and messy, but not difficult. In particular, do **not** assume that `SEQUENCE` implies compatibility or that sequence derived types can be used in the same way; `SEQUENCE` has its uses for MPI, but it is rather too complicated to describe in this course.

9.5 Debugging and Tuning

In practice, debugging and tuning overlap to a large extent, and tuning MPI is more like tuning I/O than tuning executable code. Also many performance problems are logic errors (as is common in all forms of parallelism) – e.g. everything is waiting for one process. Many logic errors show up as poor performance, so do not consider debugging and tuning as completely separate.

A partial solution is to design primarily for debuggability, always remembering to *KISS* (*Keep It Simple and Stupid*). This course has covered most of the MPI specific points, and there is more in the course *Software Design*; you may also like to look at the Computing Service course *How to Help Programs Debug Themselves*. If you do that, you will rarely need a debugger, and diagnostic output is usually good enough. Only when you have got the program working, start to worry about performance.

The specimen answers to the practical examples waste most of the memory they use, but they are just trivial examples. Here are some guidelines for memory optimisation in real programs that need a lot of memory:

- Do not worry about small arrays and scalars; if they total less than 10% of the memory you use, who cares?
- For big arrays, allocate only what you need; for example, with gather and scatter you need a buffer for each process only on the root.
- Reuse large buffers or free them after use. Be careful about overlapping use if you do reuse them, of course; this is particularly likely to cause trouble with non-blocking transfers.

Ultimately only elapsed time matters to the performance of an MPI program – the real time taken by the program, from start to finish. If you have CPUs that are dedicated to the processes, how much CPU time they use is irrelevant. All other measurements are just tuning tools, which actually simplifies things considerably – you can concentrate on minimising just the elapsed time. You may want to analyse the elapsed time by CPU count, because it will tell you the scalability of the code, and whether it needs redesigning to make effective use of more processes.

9.6 Designing For Performance

Here is the simplest way to do this:

- Localise all major communication actions in a module, or whatever is appropriate for your language and programming style, and keep its code very clean and simple.
- Do not assume any particular compiler, MPI implementation or or operating system. This applies primarily to the module interface, which should be kept generic, clean and simple.
- Keep the module interfaces fairly high level, such as a distributed matrix transpose. Low level interfaces provide relatively little scope for tuning.
- Use the highest level appropriate MPI facility. Use its collectives where possible (preferably the `MPI_All...` forms), because collectives are easier to tune than point-to-point, surprisingly.

Most MPI libraries have had extensive tuning, and it is a rare programmer who will do as well. The `mpi_timer` code implements `MPI_Alltoall` in many ways, and was used for benchmarking HPC systems. Usually, one or two of them were faster than the vendor's built-in `MPI_Alltoall`, but not often the same ones, and almost always by under 5% (often by under 2%).

You should put enough timing calls into your module, so that you can summarise time spent in MPI and in computation. You should also check for other processes or threads (either ones started by MPI or other ones running on the system), but only for ones that are **active** during MPI transfers – sleeping threads do not matter.

Now you should look at the timing to see if you have a problem. If it is none, which is most likely, you need do nothing. If there is, try using only some of the cores on a multi-core CPU for MPI (i.e. reducing the number of MPI processes per node); it is an easy change, but may not help. If that fails, you need to start more detailed tuning.

9.7 High-Level Approach to Tuning

Try to minimise inter-process communication, and there are three main aspects to this:

- The amount of data transferred between processes. Inter-process bandwidth is a limited resource, whether the problem is in the memory subsystem, the network card or the switch.
- The number of transactions involved in transferring your data. Both the overheads involved in a transfer and the message-passing latency are significant.
- When one process needs data from another and cannot proceed until it gets it – this is mainly relevant to point-to-point. That may require the process to wait unnecessarily, wasting time.

Another aspect is that of data management. Partitioning is critical to efficiency, and has already been mentioned. You can also bundle multiple messages together, because sending one message has a lower overhead than sending several. You can minimise the amount of data you transfer, but that is only worthwhile if your messages are large. And you can arrange that all processes communicate at once – that can help a lot (surprisingly enough) because of progress issues, and is one reason that collectives are efficient.

On a typical cluster or multi-core system, packets of less than 1 KB are inefficient, because the overhead/latency dominates the time, and packets of more than 10 KB do not need bundling, because the bandwidth does. You should avoid transferring a lot of small packets, which is why packing up multiple small transfers helps, but **only** if significant amount of time is spent in them. There is no point in writing complicated code to tune something that accounts for only a few percent of the total time. And remember that integers can be stored in doubles.

9.8 Timer Synchronisation

Timer synchronisation means synchronisation across processes – i.e. are all results from `MPI_Wtime` consistent? It will almost always the case on SMP systems, and will often be the case even on clusters. Generally, you should try to avoid assuming or needing it, because you rarely need to compare timestamps across processes. If you use only local intervals, you will not have a problem, because time passes at the same rate on all processes, and so time intervals are comparable between processes.

Beyond that is a job for real experts only. The simplest way of thinking of it is that parallel time is like relativistic time, where the ordering of events depends on the observer. There is a partial solution to the problem of inconsistent clocks in directory `Posixtime`, which provides functions to return globally consistent timestamps. I wrote this for a system with inconsistent clocks, where I needed a global time to investigate some problems.

9.9 MPI and Normal I/O

This means language, POSIX and Microsoft I/O - i.e. the normal file and network I/O that every program uses. There are serious problems, and **not** because of MPI; they are caused by the system environment it runs under. This will describe the most common configuration only; if it does not apply, you should look at the extra lecture on I/O, or ask your system administrator to help you.

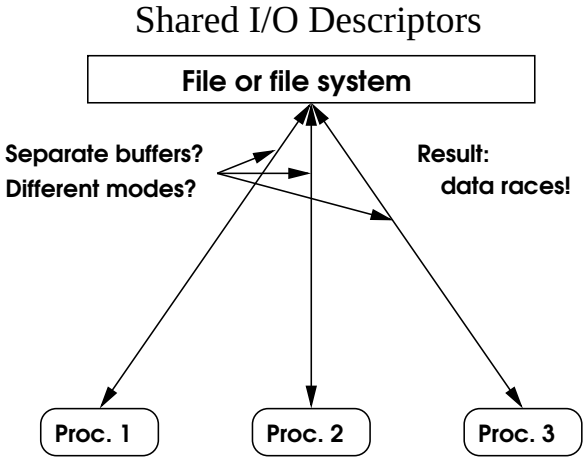

Figure 9.1

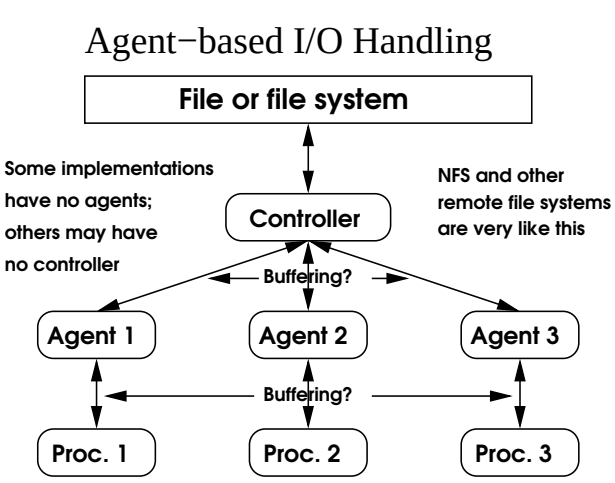

Figure 9.2

There are two, very different, classes of file: normal named and scratch files, and the standard units, `stdin`, `stdout` and `stderr`. The reason is that the former are local to a process, and the latter are global to the whole program. Almost all problems are caused by the system environments and implementations (e.g. clusters of distributed memory systems versus shared file descriptors on SMP systems).

- These issues are **not** specific to MPI, and all other parallel interfaces (including POSIX threads) have the same problems, though they may show up differently.

We shall assume all processes share a filing system. This may be shared directly, using POSIX on a SMP system, or indirectly, using NFS or something similar. The equivalents on Microsoft and other operating systems are very similar. We shall also assume that all processes share a working directory; with luck, that is either controllable (by you) or your home directory. The details are very system-dependent, as usual.

Here are some rules on how to use files safely:

- Always use write-once or read-many (i.e. a file is either written to by a single process or read by an arbitrary number of them, but not both). That applies to the whole duration of the run, and not just at any instant.
- **All** updates and accesses must be considered, including any that are done outside MPI, such as you accessing the file interactively.

To rephrase it, 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 at any time during the run. This may seem extreme, but is safe. If you need to do more than this, and it is fairly often necessary, there are some instructions on how to minimise the risks in the extra lecture on I/O.

Regard a directory as a single file (which is often the way it is implemented). If you change it in **any way** in any process, do not access it from any other process during the run. Creating a file in it counts as a change, of course. If you do, a directory listing run in parallel may fail in any one of a number of obscure ways. Listing a read-only directory

(i.e. one which is not changed during the run) is safe.

There is one exception to this , where you can update a single directory from multiple processes. You can create, delete and rename separate files in the same directory from different processes fairly safely, though not under Microsoft DFS. But you should do **all** operations on any single file in a single process – e.g. you should not create it in one and delete it in another.

You should not assume where scratch files go. While that statement applies even on serial systems, it gets it is even more complicated on parallel ones. It is common to have shared working directories, but separate, distributed scratch directories, or scratch directories shared between some processes and not others. This is just a warning, because clean code rarely has trouble.

9.10 Standard Units

The issues with these arise from implementation details. They almost always show up with standard output, which is probably just because almost all programs use it! It is an almost unbelievable can of worms, and you should not even try to program round the problems; the only good solution is to bypass the issue entirely. And remember that these issues are **not** specific to MPI; all other parallel interfaces have the same problems.

The "right" solution is also the simplest. Only the root process does I/O to or from the standard units. The root process does all the reading from `stdin`, and broadcasts or scatters it to the others. It gathers all of the output from the others, and then it writes it to `stdout`. It can also done for ordinary file I/O, and some MPI programs do that.

You have learnt all of the techniques you need, or you can look at the extra I/O lecture for details. If the root process both handles I/O and does computation, I do not recommend doing them asynchronously (i.e. so that they overlap each other). You should code the I/O transfers as a collective, which is relatively easy to debug and tune. Once you are reasonably confident with the basics of MPI, I recommend looking at the practicals associated with the I/O lecture.

You can just write error messages and other diagnostics to `stderr` or its equivalent; Fortran users may need to use `FLUSH` (either a system-dependent subroutine or the Fortran 2003 I/O statement). It may well get mangled for reasons given above, and it may get lost on a crash or following `MPI_Abort`, but it is simple to code, and errors should be rare! The same applies to `stdout`, with some programs (i.e. where it does not matter if it gets a bit mangled). Beyond that, you need to use a dedicated I/O process, just as we described above for `stdout`.

9.11 Practicals

There are some practicals on how to handle I/O, mainly how to spool it through the root process. You have already learnt all of the techniques needed to do them, so reading the later I/O lecture is not necessary. You are recommended to do at least some of these, as you are likely to need to use the techniques. There is also a trivial one on transferring structures.

9.12 Appendix: Progress

MPI has an arcane concept called *progress*. The good news is that you do not need to understand it in detail; the bad news is that you do need to understand the issues before starting any advanced tuning, and this appendix summarises them. Readers who use only the simple tuning rules described above may ignore this appendix; it is included here to explain why those rules are so simplistic.

No valid MPI program can get stuck (often called 'hang' or 'deadlock'); the MPI specification does not allow any way of creating 'deadly embraces'. If the program is valid, an implementation must always make progress and eventually complete. Obviously, a programmer must not make that impossible, so there are a few restrictions to ensure that it is possible. If you write sanely and follow the instructions in these lectures, you will almost certainly never notice the restrictions.

- Mistakes will happen, but deadlock almost always means that there is a bug in **your** code, so fix that.

MPI does not specify how it is implemented, and progress can be achieved in many ways. All valid MPI programs will work in all cases, but the implementation differences do change the most efficient coding style. The following will describe a few of the most common methods, and indicate the main consequences of them.

MPI does not specify synchronous behaviour, even for collectives and blocking transfers; all that it specifies is the ordering semantics. The actual transfers can occur asynchronously and, in theory, so can almost all other actions. This allows an implementation to overlap transfers and computation; unfortunately, it is not as simple as that, because many I/O mechanisms are often actually CPU bound.

On most shared memory systems, a CPU core is needed to copy data, but that is also the case for TCP/IP, especially over Ethernet; sent data has to be packed into TCP, IP and Ethernet messages, and unpacked from them on receipt. Also, some MPI transfers include data management, such as scatter/gather in MPI derived datatypes. Again, in theory, InfiniBand has such functionality in hardware, but not all InfiniBand interfaces use it. There are more hardware and operating system variations, too.

Eager Execution is one of the mainly synchronous methods, and if the easiest to understand, but not usually most efficient – for that reason, it is rarely used. In this case, 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. With this, there is often little point in non-blocking transfers.

Lazy Execution is also one of the mainly synchronous methods, just not in the way that most people expect. In this case, most MPI calls put the operation onto a queue, and all calls check the queue for operations that are 'ready' in some sense. The call that detects the readiness completes the queued operation. `MPI_Wtime` gives fairly strange results, because one MPI call often does all of the work for another, but the total time is fairly reliable. This is possibly the most common implementation type.

Asynchronous Execution is where MPI calls put the operation onto a queue, but another process or thread does the actual work (or, rarely, special hardware). `MPI_Wtime` gives

very strange results indeed, and you need to check the time used by the **other** process or thread. This is fairly rare – I have seen it only on *IBM AIX*, but have heard of other implementations – however, it is almost universal when using one-sided communication on clusters. With this, the only simple tuning is not to use all CPUs for MPI, and beyond that tuning gets very tricky.