Message-Passing and MPI Programming

Introduction

N.M. Maclaren

nmm1@cam.ac.uk

July 2010

1.1 Introduction

MPI stands for the *Message Passing Interface* standard. It is not the only message-passing interface specification, and "message-passing interface" is a generic term, so be careful not to get confused.

CPUs got faster at ≈40% per annum until 2003, but since then have only got larger; they are running at about the same speed they were in 2003. Vendors are now increasing the number of CPU cores per chip at ≈40% per annum. So the way to get more performance is to use more CPUs in parallel, and MPI is a tool for that. That was not new, of course, and MPI was derived from a several previous mechanisms.

This course was originally written as a "transferrable skills" course – i.e. a course given to staff and graduate students, not as part of a degree. The design was based on the analysis of (initially) twelve real scientific research applications. So it is practically oriented, and intended for research scientists.

However, the course also teaches some of the computer science that underlies MPI, including why things are as they are, and why certain methodologies are good programming practice. It is important to understand the reasons behind its recommendations, and to take note of warnings. You cannot just follow the recipes in these notes.

Applications I have looked at include Casino, CASTEP, CETEP, CFX 11, CPMD, Fluent, FFTW, my own mpi_timer, ONETEP, PARPACK, SPOOLES and ScaLAPACK – and, later, CRYSTAL, DLPOLY_3 and TOMCAT. They are not all independent, but cover a wide range. The only facility that they used which this course omits entirely is parallel I/O, which was used only only in Fluent and DLPOLY_3; it is very specialist, and few people will be interested in it.

1.2 Objectives and Prerequisites

At the end of this course, you will be able to understand the MPI usage of at least 90% of the scientific research applications that use MPI, and be able to write large, efficient, portable and reliable MPI programs of your own. Naturally, expertise develops with practice, but you will be familiar with all of the basic techniques and have an understanding of how MPI implementations work.

You will also be able to use the MPI standard itself as a reference, for looking up the precise definition of the functions and facilities you use and, once you are familiar with what this course covers, learning new ones.

It assumes that you are already a reasonably competent programmer in one of C, C++ or Fortran 90, and can use such techniques as adding diagnostic output statements for debugging. It also assumes that you can use Unix for program development. There are some Computing Service transferrable skills courses on those, as well as on other related aspects, which are referred to in passing. However, they are **not** part of this MPhil, and you will not get credit for them. For those, see:

The lectures will cover most of the following topics, though not necessarily in the following order:

- The basic principles of distributed memory parallelism, message passing and MPI.
- How MPI is used from the different languages.
- MPI datatype descriptors and collective operations (i.e. ones that involve all processes working together).
- Synchronous and asynchronous point-to-point transfers; this is what most people think of as message-passing.
- MPI's error handling.
- Working with subsets of the processes (MPI communicators etc.)
- Problem decomposition, performance, debugging and I/O.

1.3 Further information

There is a fair amount of material that goes with this course, including practical examples with specimen answers, to teach the use of specific MPI facilities. You are strongly advised to work through most of those, as there is no substitute for actually using a facility to check that you have understood it. There are also some specimen programs, and the interface proformas of the MPI calls used for each of the above language. This material is available on:

`MPI/`

You do not need to learn the MPI standard to use MPI effectively, but should be familiar enough with it to use it as a reference. It is very clear and well-written though, like almost all programming standards, its order is confusing and its indexing is extremely poor. The MPI Forum's home page is `http://www.mpi-forum.org/`.

Most books, Web pages and courses spend far too little time on the basic concepts, and too much time on the more advanced features; they also rarely warn you about potential problems. One of the better books is *Parallel Programming with MPI* by Peter Pacheco, which is described in `http://www.cs.usfca.edu/mpi/`. However, this course does **not** follow that book!

Be careful when searching the Web, because it is called The Web of a Million Lies for a reason – in fact, that is an underestimate! Information on MPI is relatively reliable, but often misleading. The following pages are all reliable, but are not an exhaustive list:

`http://www-users.york.ac.uk/~mijp1/teaching/4th_year_HPC/notes.shtml`

http://www.epcc.ed.ac.uk/library/documentation/training/

http://www-unix.mcs.anl.gov/mpi/

1.4 Distributed Memory and Message-Passing

Distributed memory is one of the basic models of parallelism. A program consists of a number of separate, independent processes, each of which behaves exactly as if it is a separate serial program. The term distributed memory means that there is no data (i.e. variables) shared between the processes. The only communication is by message passing.

Note that this is entirely distinct from whether the processes run on the same system or on different ones; both ways of working are common, and a MPI program does not even need to know which it is running under.

Message passing is one of the basic communication designs. The simplest form is when one process sends a message (i.e. packet of data) to another process, and the second one receives it. It is really a form of process-to-process I/O, and is very like Email; in fact, it is usually implemented in very similar ways. The difference is it is between two processes of a single program, and not between a process and a file or between two mail agents.

There are some more complications with message passing, but they are all based around the same ideas. The only exception in MPI is one-sided communication.

1.5 MPI: the Message Passing Interface

MPI, the Message Passing Interface standard, is the specification of a library callable from Fortran, C (and so also C++); bindings are also available for Python, Java etc. It is the current *de-facto* standard for HPC (High Performance Computing) programming on multi-CPU systems.

It uses the distributed memory, message passing model, and assumes a number of processes running in parallel, usually with CPUs dedicated for the MPI processes (i.e. gang scheduling). Essentially all HPC work on clusters uses MPI, and it works nearly as well on multi-core shared-memory systems running a mainstream operating system. It behaves poorly for background work on systems that are primarily used for other purposes (e.g. cycle stealing).

It can also be regarded as a specialist communications library, very like POSIX I/O, TCP/IP etc., but with a different purpose. Using its interface is almost never a problem; if you can use any moderately complicated library, you can use MPI. It is also almost completely independent of the system it is running on, so is very portable. The most important step is to understand its computational model, which means mainly the assumptions underlying its design. The same is true for C++, POSIX, Fortran, TCP/IP and .NET.

The MPI Standard was a genuinely open standardisation process mainly during the second half of the 1990s, with some extension work later, and work is still going on. See:

http://www.mpi-forum.org/docs/docs.html

MPI 1 is the basic facilities – it covers everything that most people use, and most people use only a small fraction of it! MPI 2 is some extensions (i.e. other facilities), plus some improved interfaces, and includes the MPI 1.3 update; it is upwards compatible with MPI 1. MPI 3 adds yet yet more, but those are not mentioned in this course. This course covers much of MPI 1 with the relevant improvements of MPI 2, and a few MPI 2 facilities.

The document is a *standard*, not a *user's guide*, which means that it is designed to be unambiguous, and is not primarily written to be easy to follow. As a standard, it is as good as the Fortran one, and much better than the C or POSIX ones. Unfortunately, its order and indexing are not good, and I am still finding new features after a decade and a half of using it. That is a very common fault of language standards.

Use something else to find what to look up, and then use the standard to look up the precise specifications of what you want to do.

There are two open source versions – MPICH and OpenMPI – which you can install as packages or build from source. Most vendors have their own, including Intel and Microsoft. Under the covers, most of them use shared-memory on multi-core machines for communication, TCP/IP over Ethernet and other networks, and often InfiniBand or the vendor's proprietary transport on suitable HPC clusters.

However, you need to make **no** code changes when moving your code between implementations, often not even to tune them. MPI programs are very portable, and can usually be written to run reasonably efficiently on almost all implementations.

1.6 The MPI Model

The programmer starts up one or more independent processes, and all of them start MPI and use it to communicate. There is no "master" (initial or main) process, though there are a very few circumstances where process zero is treated a little like one.

Communications between process may be "point-to-point" (pairwise), where only two communicating processes are involved, or they may be "collective", where all of the processes are involved. In the latter case, they must all make the same call, and must synchronise with each other.

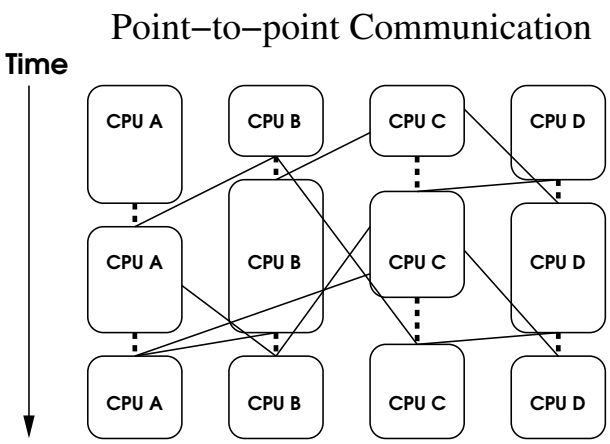

Figure 1.1

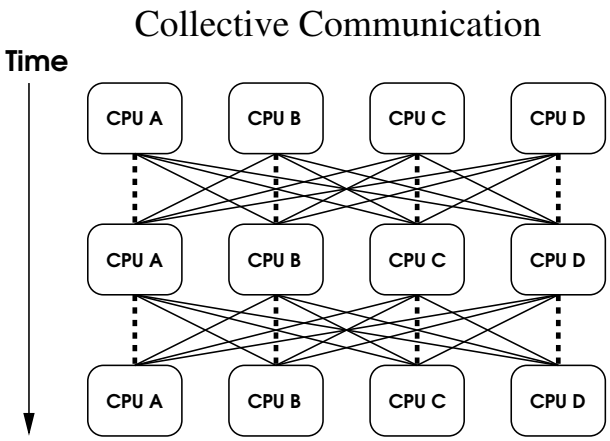

Figure 1.2

Communication may not always synchronise, and that applies nearly as much collectives as to point-to-point. Figure 1.2 is misleading in that respect. In general, processes need wait only when they need data (e.g. a send may return as soon as the message is transmitted and before the message is received). In theory, this allows for faster execution. If you want synchronisation, you must ask for it, and there are plenty of facilities for doing so, depending on precisely what you want to achieve.

Some MPI operations are non-local, which means that they may involve behind-the-scenes communication. If you make an error, or there is a system problem, they can hang (i.e. never return from the function call). Others are purely local, will never hang, and will return "immediately". Generally, this matters mainly to MPI implementors, and you only need to know that both forms exist, but a few important examples are mentioned later.

Almost everyone uses MPI in SPMD mode (that is *Single Program, Multiple Data*). In that case, each process runs using copies of a single executable. The processes can execute different instructions in that executable – i.e. they do not have to run in lockstep or SIMD mode (that is *Single Instruction, Multiple Data*), but start off by designing programs to run in SIMD mode.

SPMD mode is not required by MPI, which surprises people – in theory, each process could run a different program on a different, incompatible system, such as a collection of miscellaneous workstations. You are recommended **not** to do that, and not because of MPI. Some reasons are given in the course *Parallel Programming: Options and Design*. This course will assume SPMD mode, and many implementations support only that.

Also, in most cases, all CPUs are dedicated to your MPI program (i.e. the gang scheduling menioned earlier); that avoids certain problems (mainly performance-related).

The last critical MPI concept needed at this stage is *communicators*. These specify a group of processes and relevant context (see later), and all communications occur within communicators. Actions in separate communicators are independent. You start with

the communicator of all processes (MPI_COMM_WORLD), and you can subset any existing communicator to get smaller ones, creating a hierarchy. We shall return to these later, under advanced use but, for now, use only MPI_COMM_WORLD.

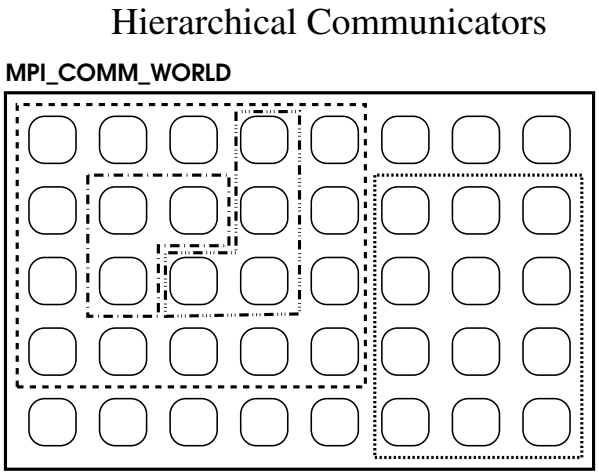

Figure 1.3

1.7 Levels of Parallelism

When considering the number of CPUs, you usually count "one, two, many"; for each of those, you often need to use different algorithms and code. There are a few algorithms that work on four CPUs but not three, or vice versa, but not many. To put it another way:

- One CPU is necessarily serial programming.
- Two CPUs are *this CPU* and *the other CPU*.
- Most parallelism issues arise only with many CPUs.

Serial codes may not work on many CPUs; parallel (many CPU) codes may not work on one CPU, and two CPU codes may not work on either. Some algorithms will work on any number of CPUs, of course, but you should always think first and code second.

MPI communicators can have any number of CPUs, from zero CPUs upwards – yes, MPI allows communicators with no CPUs. As a general rule, use 4 or more CPUs when debugging generic MPI codes – most applications assume at least that many. It is perfectly reasonable to assume a minimum number, if there is a check in your code that MPI_COMM_WORLD is large enough. Otherwise, you need different code for:

0:	typically do nothing (and is not executed!)
1:	use serial code for this
2–3:	a few generic algorithms fail
4+:	'proper' parallel working

1.8 Diversion – a Worked Example

This worked example of the use of MPI, which calculates the area of the Mandelbrot set, is to give you a feel for what MPI is about. Do not worry if you do not understand the details; every facility used will be explained later. The whole source is in the extra files, and there are Fortran 90, C and C++ versions.

The Mandelbrot set is defined in the complex plane, by the set of all parameter values of a recurrence where iterated use of the recurrence does not diverge (which implies $|x_n| \leq 2$). The recurrence is $x_{n+1} \Leftarrow x_n^2 + c$, with the starting condition $x_0 = 0$. Therefore the Mandelbrot set is the set of all c, such that:

$$|x_n| \leq 2 \quad \forall n \geq 0$$

This is, er, complicated – let's see a picture:

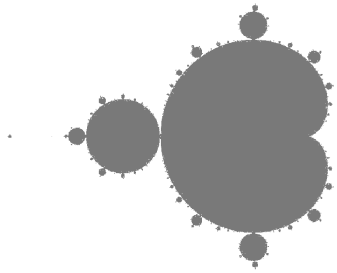

Figure 1.4

All points within the Mandelbrot set have $|c| \leq 2$, and it is also symmetric about the X-axis, so we consider just points c, such that:

$$-2 < re(c) \leq +2$$
$$0 < im(c) \leq +2$$

We turn that into a practical algorithm by choosing a suitable iteration limit and step size. We then see if each point stays small for that long, assume that it is in the set if it does, and we accumulate the scaled count of those that do. Incidentally, there are **three** sources of error in that process – it is left as an exercise to work out what they are! Here is a picture of what we do:

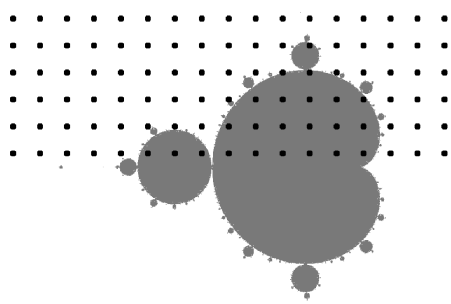

Figure 1.5

Note that this is the crudest form of numerical integration; it is not strictly Monte-Carlo integration, but it is related. Sometimes a sledgehammer is the best tool and, in this case, it works well enough.

1.9 The Example Code

We shall go through the Fortran 90 version here; the C and C++ ones are very similar. Most of it is just the ordinary, serial logic, and we go through the core first.

```
PURE FUNCTION Kernel (value)
    IMPLICIT NONE
    LOGICAL :: Kernel
    COMPLEX(KIND=DP), INTENT(IN) :: value
    COMPLEX(KIND=DP) :: work
    INTEGER :: n

    work = value
    DO n = 1, maxiters
        work = work**2 + value
        IF (ABS(REAL(work)) > 2.0 .OR. ABS(IMAG(work)) > 2.0) EXIT
    END DO
    Kernel = (ABS(WORK) <= 2.0)
END FUNCTION Kernel
```

The above is a function that runs the recurrence `maxiters` times for a single value of the parameter `value` and returns `.True.` if the value has not diverged (i.e. the point appears to be within the set). It is just an ordinary, serial function, but can be called in parallel on each process.

```
PURE FUNCTION Shell (lower, upper)
    IMPLICIT NONE
    REAL(KIND=DP) :: Shell
    COMPLEX(KIND=DP), INTENT(IN) :: lower, upper
    COMPLEX(KIND=DP) :: work

    Shell = 0.0_DP
    work = CMPLX(REAL(lower),IMAG(lower)+step/2.0_DP,KIND=DP)
    DO WHILE (IMAG(work) < IMAG(upper))
        DO WHILE (REAL(work) < REAL(upper))
            IF (Kernel(work)) Shell = Shell+step**2
            work = work+step
        END DO
        work = CMPLX(REAL(lower),IMAG(work)+step,KIND=DP)
    END DO
END FUNCTION Shell
```

This is a function that calls the kernel function for all points within a rectangle defined by the two complex numbers `lower` and `upper`, in steps of size `step`, and returns the area of the set in that rectangle (i.e. the scaled number of points). It is another ordinary, serial function and, again, can be called in parallel on each process.

```
LOGICAL, PARAMETER :: UseMPI = .True.
INTEGER, PARAMETER :: root = 0
INTEGER :: maxiters, error, nprocs, myrank
REAL(KIND=DP) :: buffer_1(2), step, x

IF (UseMPI) THEN
    CALL MPI_Init(error)
    CALL MPI_Comm_size(MPI_COMM_WORLD,nprocs,error)
    CALL MPI_Comm_rank(MPI_COMM_WORLD,myrank,error)
ELSE
    nprocs = 1
    myrank = root
END IF
```

This is the beginning of the main program. All processes initialise MPI and find out the number of processes and their current process number (the rank). If the program is compiled not to use MPI, it sets the number of processes to 1, so the program can be run either serially or in parallel. This is obviously MPI code.

```
COMPLEX(KIND=DP), ALLOCATABLE :: buffer_2(:,:)

IF (myrank == root) THEN
    ALLOCATE(buffer_2(2,nprocs))
    buffer_2(1,1) = CMPLX(-2.0_DP,0.0_DP,KIND=DP)
    DO i = 1,nprocs-1
        x = i*2.0_DP/nprocs
        buffer_2(2,i) = CMPLX(2.0_DP,x,KIND=DP)
        buffer_2(1,i+1) = CMPLX(-2.0_DP,x,KIND=DP)
    END DO
    buffer_2(2,nprocs) = CMPLX(2.0_DP,2.0_DP,KIND=DP)
ELSE
    ALLOCATE(buffer_2(2,1))    ! This is not actually used
END IF
```

The root process divides the bounding box of the Mandelbrot set into domains, and allocates some buffers for later use. While this is not MPI code, as such, it is needed for later use by MPI code.

```
INTEGER :: maxiters
REAL(KIND=DP) :: step

IF (myrank == root) THEN
    READ *, maxiters, step
    IF (maxiters < 10) THEN
        PRINT *, 'Invalid value of MAXITERS'
        STOP
    END IF
    IF (step < 10.0_DP*EPSILON(step) .OR. step > 0.1_DP) THEN
        PRINT *, 'Invalid value of STEP'
        STOP
    END IF
END IF
```

The root process reads in the maximum number of iterations and the step. This is more ordinary, serial code, but is executed only on the root.

```
REAL(KIND=DP) :: buffer_1(2)
COMPLEX(KIND=DP), ALLOCATABLE :: buffer_2(:,:)
COMPLEX(KIND=DP) :: buffer_3(2)

IF (myrank == root) THEN
    buffer_1(1) = maxiters
    buffer_1(2) = step
END IF

IF (UseMPI) THEN
    CALL MPI_Bcast(    &
        buffer_1,2,MPI_DOUBLE_PRECISION,    &
        root,MPI_COMM_WORLD,error)
    maxiters = buffer_1(1)
    step = buffer_1(2)
    CALL MPI_Scatter(    &
        buffer_2,2,MPI_DOUBLE_COMPLEX,    &
        buffer_3,2,MPI_DOUBLE_COMPLEX,    &
        root,MPI_COMM_WORLD,error)
ELSE
    buffer_3 = buffer_2(:,1)
END IF
```

This distributes the data from the root (which has calculated which process needs to do what, and read in the parameters) to all of the others. This is obviously MPI code, and is mostly executed on all processes (`MPI_Bcast` and `MPI_Scatter` are collectives, so all processes are involved).

```
buffer_1(1) = Shell(buffer_3(1),buffer_3(2))
IF (UseMPI) THEN
    CALL MPI_Reduce(    &
        buffer_1(1),buffer_1(2),1,MPI_DOUBLE_PRECISION,    &
        MPI_SUM,root,MPI_COMM_WORLD,error)
ELSE
    buffer_1(2) = buffer_1(1)
END IF

IF (myrank == root) THEN
    PRINT '(A,F6.3)', 'Area of Mandelbrot set is about',    &
        2.0_DP*buffer_1(2)
END IF
IF (UseMPI) THEN
    CALL MPI_Finalize(error)
END IF
```

This calculates the area in each domain, in parallel (the call to function `Shell`), ac-

cumulates the total area from all of the processes, the root process prints the result, and then all processes terminate. This is mainly MPI code.

1.10 Running the Example

So what happens when we run this? On one cluster, running with parameters '10000 0.001', we get about 1.508 – the true result is about 1.506. So let us see how long it takes:

1	67 seconds
4	46 seconds
16	23 seconds

That is not very scalable, which is is quite common; using MPI is much easier than tuning it. The reason that this is inefficient is because some of the domains have a lot of long-running iterations and others only a few. There is an alternative Fortran 90 version, which generates all of the points and randomises them. This gives each process a roughly matching workload, though it is a store hog, and takes some time to start. But its scalability is better:

1	70 seconds
4	19 seconds
16	8 seconds

It would scale better with more points – real MPI programs usually run for hours or days, not seconds - but this is good enough for an example.

There is a better way than even that, which is covered in the *Problem Decomposition* lecture; the first practical of that gets you to code it. The technique described there is suitable for most embarrassingly parallel problems, including parameter searching and Monte-Carlo work. Calculating the area of the Mandelbrot set was merely a convenient example. But that lecture is a lot later.