An Introduction to MPI

MJ Rutter
mjr19@cam

Michaelmas 2022

These notes were to be found, in early 2023, at https://www.mjr19.org.uk/courses/MPI/

Typeset by FoilTEX

Contents

Parallel Computing: The Background **6**

Parallel Computing . . . 6

SMP vs MPP . . . 12

Top500 . . . 23

The Beginning **36**

Background . . . 36

Hello, World (`MPI_Init, MPI_Finalise, MPI_Comm_size, MPI_Comm_rank, MPI_Abort`) . . . 40

Simple Quadrature (`MPI_Reduce`, MPI datatypes, `MPI_Allreduce`) . . . 57

MPI Versions (`MPI_Get_version`) . . . 65

C Memory Management . . . 68

More Quadrature **76**

Monte Carlo . . . 83

Broadcasts (`MPI_Bcast`) . . . 89

Gather / Scatter (`MPI_Gather, MPI_Scatter`) . . . 93

Basic error handling (`MPI_Comm_set_errhandler`) . . . 102

The Mandelbrot Set **104**

The Mandelbrot Set (`MPI_Barrier`) . . . 105

Point to point (`MPI_Send, MPI_Recv`) . . . 125

Master Slave . . . 129

Laplace's Equation **140**

Serial . . . 143

Parallel . . . 150

Buffered Sends (`MPI_Bsend, MPI_Buffer_Attach`) . . . 160

Immediate Sends (`MPI_Isend, MPI_Waitall, MPI_Wait, MPI_Irecv, MPI_Test`) . . . 164

Deadlocks, and Summary of Sends (`MPI_Sendrecv, MPI_Ssend`) . . . 171

Miscellaneous **178**

Unknown Message Sizes (MPI_Probe) . . . 179
Mistakes . . . 182
Alltoall and Transposes (MPI_Alltoall) . . . 189
The Wrong Buffer (MPI_Get_count) . . . 192
Performance (MPI_Wtime) . . . 195
Progress . . . 202
More Advanced Features **210**
Communicators . . . 211
MPI I/O . . . 218
User-defined Datatypes . . . 243
Hardware **250**
Parallel computers . . . 251
MPP . . . 255
SMP & NUMA . . . 262
Hybrid MPI/OpenMP . . . 282
Zero Copy . . . 284
Further Work **288**
Hello, World . . . 288
Quadrature . . . 289
Mandelbrot Set . . . 291
Laplace . . . 292
Master Slave revisited . . . 295
Miscellaneous . . . 295
Advanced Features . . . 297
Hardware: NUMA . . . 299
Reference **304**
MPI Datatypes . . . 304
Fortran mpi vs mpi_f08 . . . 304
Summary of MPI-I/O commands in this booklet . . . 305

Summary of MPI commands in this booklet 307
Index **309**

Parallel Computing: The Background

Why Parallel?

Speed? Memory? A desire to use all the cores in one's computer?

A desire to be modern?

Speed

Speed is sometimes over-emphasised. Patience can be a good substitute for speed.

However, there are occasions where speed is critical.

If one cannot forecast tomorrow's weather in less than 24 hours, the forecast will be useless.

If you cannot complete the calculations you need for your PhD in under three years, there will be problems too.

Memory

This can be a bigger problem. If the memory required for a 'big' (detailed?) simulation is more than fits in a single desktop, then the simulation simply won't run. With speed one can often compromise by adding patience, but with memory one cannot.

Trying to use hard disks as a substitute for DRAM is rarely profitable unless one is very careful. DRAM access times are under 100ns as measured from an application, whereas hard disk access times tend to be around 10ms. A factor of 10^5 slower. Bandwidths too are very different: a disk would be expected to give around 0.15GB/s, whereas DRAM at least 15GB/s.

It has recently become possible to purchase relatively cheap computers in which a single CPU core can access 1TB or more (cheap being around £10,000). But one can still reach much larger numbers by using multiple computers joined together to act as a single parallel machine.

Being Modern

Given that parallel computing started to emerge in the 1960s, perhaps not very modern.

But for various reasons adoption was quite slow. Even by the mid 1980s the fastest computers in the world were still single CPU (and single core). PCs with multiple CPUs existed by the mid 1990s, but the prices in a 1995 advertisement listing a dual 133MHz Pentium with 16MB RAM and a 1GB hard drive for $4,600, with the option to double the memory and disk capacity for an extra $1,000 (15″ monitor included in both prices) suggest that these would not have been common.

It was not until 2005 and the near simultaneous launch of dual-core desktop CPUs by Intel (Pentium D) and AMD (Athlon64 X2) that multiple cores or CPUs in a computer started to become common. The change was quite swift, in that Intel's Core2 processor launched in 2006 did not have a single core desktop version, only dual and quad core versions.

Of course today (2022) one would struggle to find a single-core computer. Desktop, laptop, tablet, smartphone are almost never seen in single-core versions. Even dual core is rare. For £35 one can buy a Raspberry Pi 4, a basic hobbyist's computer with 2GB of memory and a quad core processor (but no PSU, so it is tricky to turn on).

Most programming languages have their roots in a world in which parallel computing was not common: Fortran (1950s), C (early 1970s), C++ (mid 1980s), Python (late 1980s, python 2 in 2000).

Is it easy?

Sometimes very easy, sometimes very hard, and generally somewhere in between.

Brute-force factorisation is very easy.

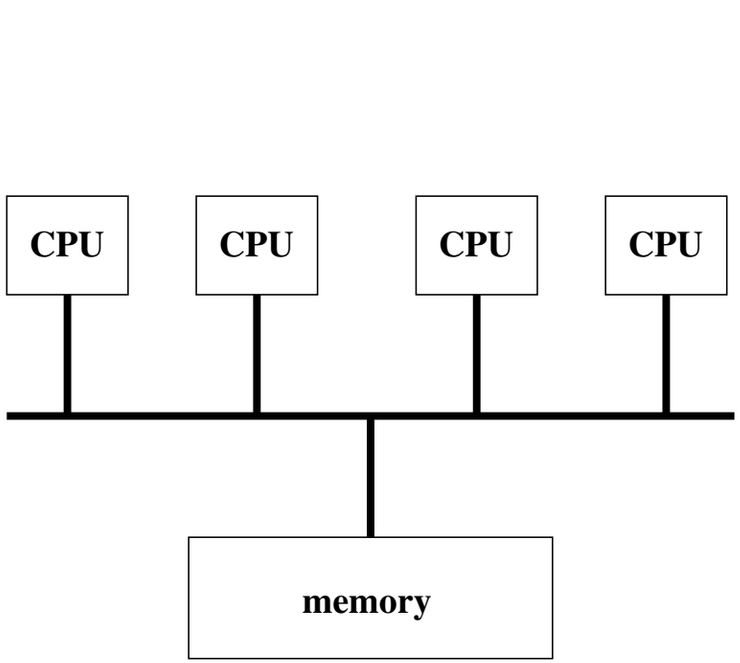
Evolving a simply differential equation over a large number of timesteps is very hard (or impossible).

But much lies between these extremes.

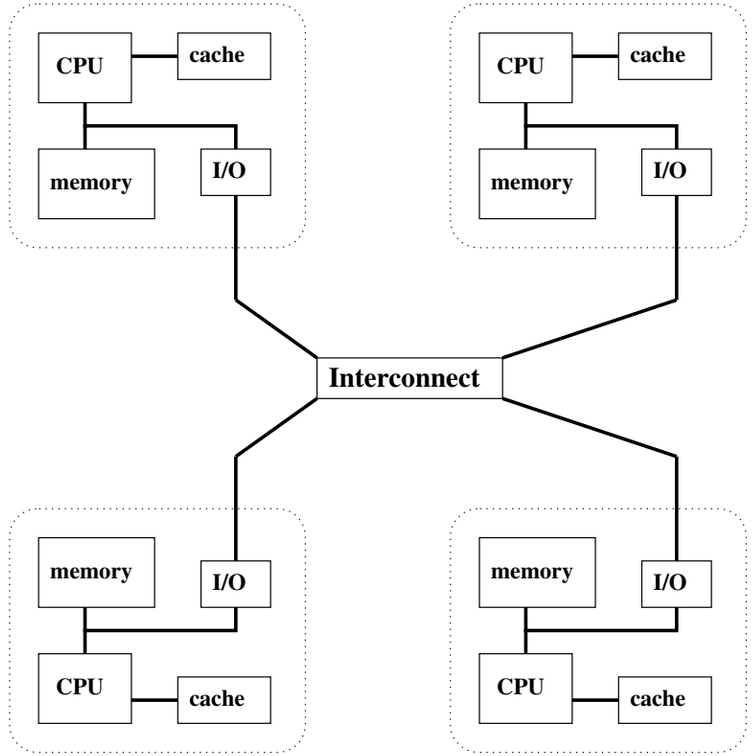
Two Roads

There are two main routes for parallelisation. In one, at the hardware level, all the memory is (potentially) addressible by all processors. The memory is said to be shared by all the processors, and the result is referred to as an 'SMP' machine (Shared Memory Processor (or Parallelism)).

In the other, memory is distributed around the processors, and each processor can access some amount directly, and for the rest it needs to send explicit communication messages over some form of network. This is normally refered to as a distributed memory machine, but generally abbreviated MPP (Massively Parallel Processor).



VS



Pros and Cons

Shared Memory is much more flexible. The hardware might say that all memory can be addressed by all processors, but one can always choose not to do so, and thus pretend that one is running on a distributed memory machine.

There is one big problem. As any CPU can talk to any piece of memory, the internal connections quickly become a complex, expensive, compromised mess which slows access between CPUs and memory. This slowdown occurs even if the software is not actually sharing memory – it is a consequence of the hardware being able to support genuine sharing.

SMP works well for around 10 cores, considerably less well and more expensively for up to 100, and 1,000 is certainly very ambitious. Economics are also bad, as large core-count machines are rare, so have low production volumes, and high costs.

Cheap and simple

A classroom full of desktop PCs, connected by a network, could, with the correct software, be regarded as a single parallel computer. It is likely to have a couple of hundred cores and a TB or so of RAM.

It will also probably be cheapish – the individual computers can be standard mass-produced ones. They will not need a non-standard, complex, and slow, memory controller. A CPU in a large SMP machine will probably run code more slowly than the same CPU running at the same speed in a simple uniprocessor desktop mostly because of the inferior performance of the complex memory controller it needs.

Thoughtful

Distributed memory machines do need a little more thought when programming. How should data be distributed amongst the discrete computers? How can communication be minimised? And it is less flexible.

If I have access to a 1TB 24 core shared memory machine, and cannot work out how to parallelise my code at all, I can run a serial code which can access all 1TB. If I have access to twenty 64GB eight-core computers, I might be happier – in theory I have 1.25TB and 160 cores. But, if I cannot work out how to parallelise my code, I am stuck with a 64GB limit for a serial process.

This course focuses more on the mechanics of how to use MPI rather than techniques for parallelising algorithms.

Good News

Although SMP and MPP programming requires two different approaches, there is one approach for each which is dominant.

For SMP, OpenMP dominates, and for distributed memory parallelism MPI dominates.

Both are standardised by international bodies, and both support both C and Fortran in the official standards. OpenMP requires compiler support, and MPI does not even require that: it is simply a library.

Processes vs Threads

OpenMP programming is based on threads. A team of threads make up a single process, and all have the same address space, so any thread can access memory used by any other thread. All threads have the same Process ID (PID), and it is the combination of PID and virtual address which maps to a physical address in a computer's memory. So the same virtual address in two different threads in the same process maps to the same physical memory location and hence the same data.

If one wishes to break up a loop so that different threads execute different parts of it, one needs to ensure that the loop counter is not shared by all threads (OpenMP does this automatically), and that any temporary variables local to each iteration are not shared (OpenMP does not do this automatically, leading to bugs whose impact depends on the compiler optimisation level, for if such temporaries reside entirely in registers there is no issue).

MPI programming is based on processes. As ever, different processes run in their own address spaces, and are generally completely isolated from each other, as isolated as if they were running on different computers. If they wish to communicate, or to exchange data, they must do so explicitly, and that is what the MPI library is for.

Blatant Bias

Given that I am here to lecture MPI, I would of course say that MPI is more important than OpenMP. Here are some arguments to support this theory.

Firstly, MPI code will run on an SMP machine (and also on an MPP machine). OpenMP code will run only on SMP machines.

Secondly, on an SMP machine there is often the option of linking against parallelised maths libraries for FFTs, matrix operations, etc., and keeping one's own code serial. For many problems, this achieves a similar speed-up to full parallelisation of one's code, and is much simpler than either OpenMP or MPI.

Finally, at the hardware level both MPI and OpenMP require data to move from one core to another. This process, requiring lots of cache synchronisation, can be slow. In MPI it is very explicit: only calls to the MPI library do it. In OpenMP it is not explicit in the code, so the programmer has little control over precisely when it happens, and is sometimes oblivious to the fact that it is happening at all. For something which may have performance implications, this is not ideal.

MPI

Fortran and C are supported officially, and C++ on the basis that one can readily call any C library from C++.

Python seems to suffer from two competing interfaces to MPI: mpi4py and Boost.MPI (Boost also has a C++ interface to MPI). Java seems to be settling on mpiJava.

MPI 1.0 was released in 1994, and it continues to evolve, partly to introduce new features, and partly to make use of newer language features (e.g. MPI 3's support of Fortran 2008).

Cores, Processors, Nodes and Computers

Cores and Processors: from the programmer's perspective, they are pretty much identical. A single quad-core processor, or two dual-core processors, it is all much the same. There is not even a guarantee that all cores in a processor will share the same last level cache.

From a hardware perspective, there is an obvious difference. The processor is the thing which plugs into a processor socket, and which may contain internally multiple cores, and perhaps a GPU too.

Node: that part of a computer in which all processors can access all the memory. So an SMP computer is a single node.

Computer: something which accepts a single parallel job. If the job is an MPI one, this might be multiple nodes, each of which could be a computer in its own right from the perspective of serial jobs.

So much scope for confusion.

The Importance of Not Communicating

For distributed memory computers, sending data between nodes is slow. One would be very lucky to do so in 1μs. If a node has a peak performance of 100 GFLOPS (actually quite modest), then the communication is as costly as 100,000 floating-point operations.

Communication needs to be minimised, by good design of data distribution and algorithm.

Actually, in many cases the design does not need to be particularly good, just not absolutely dreadful. In some cases one does need to think quite hard.

And one does not need to avoid communicating as much as one needs to avoid synchronous hard disk access, which is likely to take 10ms, so is at least a thousand times worse.

What uses MPI?

A very incomplete list:

CASTEP/Onetep/Siesta: quantum Density Function Theory codes (and others not associated with TCM, such as Quantum Espresso and Abinit)

Met Office 'Unified Model': for climate and weather, also ocean modelling codes such as NEMO.

Fluent: commercial Computational Fluid Dynamics software

LS-DYNA: finite element software, much used in car crash modelling

SPRINT: parallelised R, a genomics code

Nuclear weapons simulations: allegedly

MPI-AMRVAC: astrophysical magnetofluid dynamics

HACC and others: cosmology. Also most Lattice Quantum Chromodynamics software.

All of the first two lines in this list are known to be written in Fortran. Some names above may be trademarks.

Top500

The 'Top500' is a list of the 500 fastest supercomputers in the world: https://www.top500.org/

The benchmark it uses is the solution of a set of real, double-precision, simultaneous linear equations with the problem size adjustable to suit the machine in question. The method must be similar to Gaussian elimination (i.e. order n^3), and, in particular, Stressen's algorithm is not permitted.

The speed is calculated by timing the run, and assuming that $\frac{2}{3}n^3 + 2n^2$ operations were required.

Of course there are many good reasons for arguing that this is a bad benchmark. The best of these reasons is the argument that it does not reflect the type of code that *you* want to run. Also it is rather easy to scale to a very good fraction of peak theoretical performance, something which is not true for most real-world codes.

(It should also be noted that some computers do not appear in the Top500 list. Organisations such as GCHQ, AWE, the NSA, etc., are rather reticent about admitting what hardware they use, and tend not to advertise it here or anywhere else.)

Top500 History

The first list was published in June 1993. The top thirty-three entries in the list all contained multiple cores/CPUs, but the top ten contained two entries with just four-cores, and two sixteen-core machines. The top entry did have the highest core count at 1,024 (and achieved 59.7 GFLOPS). The highest single-core machine was ranked 34th (an NEC SX-3 achieving 5.8 GFLOPS).

A forerunner to the Top500 list was a paper by Dongerra in 1987 *Performance of Various Computers using standard Linear Equations Software in a Fortran Environment*. Table 6 of that paper is the closest to the current Top500 list. It contains 36 double-precision entries, six of which are noted as using more than one processor, and only one of those six is in the top ten.

The June 2016 list was headed by a computer with over 10 million cores, and this Chinese machine retained the top position until dropping to second in June 2018. None of the computers in the top ten in June 2016 had fewer than 100,000 cores. Every machine in the top 250 has more than 10,000 cores.

None of the computers in the top twenty of the June 2019 (or 2022) list has fewer than 200,000 cores.

So whereas in 1987 parallel programming was rare, and unnecessary to access the full performance of a top ten in the world machine, now it is necessary to access the full performance of almost every laptop, and very large scale parallelism is needed for a world-leading machine.

Top500 Parallel Efficiency

The top entry in the June 2016 list, the Chinese Sunway TaihuLight, achieves 93 PFLOPS and has a theoretical peak performance of 125.4 PFLOPS. So it achieves 74% of theoretical peak performance. The number of simultaneous equations solved was 12,288,000 leading to a memory use ($8N^2$) of 1.2×10^{15} bytes. The installed memory in the machine was 1.4×10^{15} bytes, so the problem size was as big as would fit.

Entry number five, a Japanese computer made by Fujitsu with a mere 705,024 cores manages a benchmark score of 10.51 PFLOPS from a theoretical peak of 11.28 PFLOPS, so a parallel efficiency of over 93%.

For less synthetic problems, achieving a 93% of theoretical peak performance for a single serial thread before one worries about parallelisation is hard and commendable.

Of course there are other measures of efficiency. The Sunway TaihuLight consumed 15MW to achieve 93 PFLOPS, so 6 GFLOPS/W. The producers of the Top500 list also produce a Green500 list, ordered by GFLOPS/W. Cambridge's Wilkes-3 cluster is currently (June 2022) nineth, at 30 GFLOPS/W, but only number 304 in the performance list, with 2.29 PFLOPS. The majority of entries in the Top500 list decline to supply their power consumption, and some that do are under 1 GFLOPS/W.

Commodity Rules

Whereas in the 1980s and 1990s most supercomputers used processors specifically designed for them (Cray, NEC, Fujitsu, Hitachi), today most use commodity processors (or close relatives of commodity processors and graphics processors).

Designing a CPU is expensive. Then writing and maintaining software (operating system and optimising compilers) for it is also expensive. If there is insufficient investment in the software side, the CPU's potential will not be realised.

So today Cray uses CPUs from Intel or AMD, as do many other companies which previously designed their own. There are some exceptions. In the top 10 of the June 2021 list one can find two IBM Power9s, a Sunway SW26010 (a 260-core 64 bit RISC design from China), and an ARM A64FX. But the other six are all x86_64, three Intel and three AMD. In many cases nvidia GPU-based accelerators are also present. The following year was similiar, except that it was five AMD and just one Intel.

The x86_64 processors and nvidia GPUs, or their close relatives, can be found in items costing well under £1k, as well as these computers which surely cost well over £10m.

Interconnects

Even if using commodity nodes, specialised interconnects are often used. Ethernet is the cheap commodity answer. Infiniband a less cheap mostly commodity answer (certainly commodity in the sense that multiple vendors manufacture it). Then there are various custom interconnects from companies such as Cray, IBM, and others.

For something which scales as well as the Linpack benchmark for the Top500 test they are barely necessary. In the June 2019 list there is a 84,000 core entry using nodes with two 12 core 2.5GHz E5-2680v3 CPUs each (so 3,500 nodes) connected by 10GBit/s ethernet. It manages 1.1 PFLOPS from a theoretical peak of 3.36 PFLOPS, so almost 33% of peak on this benchmark. It is unlikely to scale that well running a more communication-intensive application such as Castep though!

A similar entry based again on dual processor nodes containing the same CPUs, but just 3,168 of them, but with a QDR Infiniband interconnect, manages 2.25 PFLOPS from its theoretical peak of 3.04 PFLOPS, so about 74% of peak.

It is quite likely that for the use that the Chinese machine is intended the interconnect speed matters even less than for Linpack, so the machine is a sensible configuration. If one's intention was simply to run the Linpack benchmark(?!), then a better interconnect in this case more than doubles the score even after a slight reduction in the number of CPUs.

Lenovo ThinkServer RD650 of Internet Company A, China, vs Dell PowerEdge R630 of Makman-2, Saudi Arabia. Both machines entered the list in 2015, and both were still in the June 2019 list, though that as the last time that ethernet-connected machine was included.

In the 1993 list, the USA has the first four entries, and eight of the top ten (Japan at five and Canada at six being the others). The UK's highest entry was number 12 (European Centre for Medium-range Weather Forecasting). France had an entry at number 25, the Netherlands at 41, Switzerland at 44, Germany at 55, 59 and 64, but the rest of the top 70 positions were the USA or Japan.

The 2016 list starts with two entries from China, two from the USA, one from Japan, two more from the USA, then Switzerland, Germany and Saudi Arabia. France is at 11, and the ECMWF is at places 17 and 18.

The June 2021 top twenty has USA, 7; Japan, 3; Germany, 3; China, Italy, and Saudi Arabia, 2 each; Switzerland, 1. The highest UK entry is at 58, which is unusually low.

June 2022 has USA, 8; Japan, 3; China and France, 2; Finland, Germany, Italy, Saudi Arabia and South Korea, 1. The highest UK entry is 25.

The shift to commodity parallel supercomputers is underlined by entries such as number 39 (June 2016), the Makman-2 (Saudi Arabia) which is described as 'Dell PowerEdge R630' running Redhat Linux. The R630 is a dual CPU server, an inch and three quarters high, with a starting price of £1,100 (exc VAT). There might well be examples in several Research Groups in Cambridge. With 76,032 cores, I assume that the Makman-2 contains about 3,168 of them, as it uses 12 core Xeon E5-2680 CPUs (a rather higher specification than the a single six-core CPU and 8GB, which is what you get for £1,100). Similarly number 12 on the 2022 list is made of Dell PowerEdge C4140 servers. The C4140 has a starting price of around £1,500.

The Need for Parallel Software

If your software cannot cope with any parallelisation at all, then it will not start to exploit machines in the current Top500 list.

If your software has SMP (OpenMP) parallelism, so cannot work with distributed memory, then it can scale to just 24 cores on a machine like Makman-2, rather than potentially 76,032 cores.

When I started my PhD (1990s), I had access to a Cray YM-P/8. If I did not parallelise my code (I did not), I could access all of its memory. My code would run slower in real time than if I parallelised over its eight CPUs, but as I was charged in CPU-hours, unless I could parallelise with zero overheads, I'd be charged more for my runs. As queueing time was much greater than runtime, and the other CPUs would run other people's code if I ran serial code on it, so CPU time was not wasted with serial code, the incentive to parallelise was somewhere between slight and slightly negative.

There Is No Alternative

In the past the serial performance of CPUs increased rapidly. Today that is not so. In terms of clock speed things are very much stalled: the 3.8GHz Pentium 4 was released in 2005, a 4.0GHz Core i7 (Haswell) in 2014.

Of course there is an excellent lecture in another course in this CDT which points out that clock speed is a very bad measure of a CPU's performance. However, given that even the Pentium 4 could sustain one instruction per clock-cycle (actually more), there remain three ways of increasing throughput:

1/ Multiple instructions executing simultaneously on a single core (instruction level parallelism, ILP).

2/ Instructions operating on multiple data elements at once: vectorisation or SIMD.

3/ Multiple cores / CPUs.

TINA 2

ILP: certainly supported by all modern processors, and one might hope to sustain two to three instructions per cycle under near-ideal conditions. Useful, but not Earth-shattering. Mostly dealt with automatically by the compiler.

SIMD: typical vector lengths for SIMD supported in hardware are four, with eight being introduced in high-end Intel CPUs in 2016. Useful, but not Earth-shattering. Mostly dealt with automatically by the compiler.

Multiple cores / CPUs: most desktop and laptops have four cores, high-end desktops have about 16, a Research Group could generally afford something with a hundred, and a large Department or a University should easily be able to manage something with thousands of cores.

It is clear that the big potential gains lie with multiple cores, but that paragraph lacks the coda ‘mostly dealt with automatically by the compiler.’ Hence MPI (and OpenMP).

Cambridge HPCS

It would be remiss of me not to mention the Cambridge's HPCS, or CSD3 (Cambridge Service for Data Driven Discovery) as it is now called.

The main Peta4 machine contains 544 nodes, each with two 38-core Intel Xeon 8368Q CPUs. Half the nodes have 256GB of memory, and half 512GB.

My own smallish Research Group can equal it in one respect: we have a machine with 512GB of memory. So, if your code cannot cope with distributed memory, one small Research Group can potentially equal the CSD3's multi-million pound machine.

Just like the Saudi system mentioned earlier, CSD3 is built from standard Dell servers. In so far as there is special magic, it is contained in the interconnect between the nodes and associated software, and the fileserver.

The interconnect is Mellanox HDR200 InfiniBand.

Infiniband

Commodity CPUs may have replaced specialised processors in supercomputers. However, so far commodity interconnects (i.e. ethernet) have not replaced specialised interconnects, the most widespread of which is Infiniband.

Infiniband's lead is diminishing. Ethernet is universally 1GBit/s, very readily 10GBit/s, and sometimes 40GBit/s, 100GBit/s, 200GBit/s or even 400GBit/s. The first Infiniband standard was just 2.5GBit/s, but anything in use today is likely to be 25GBit/s or 50GBit/s (HDR), with 100GBit/s (NDR) appearing in late 2022. However Infiniband generally uses multiple lanes per link, so HDR100 is two 50GBit/s lanes, and HDR200 is four 50GBit/s lanes. (A PCIe 3 x16 link is only 126GBit/s.)

Latencies are better with Infiniband, not least because most communication on ethernet is layered on top of TCP/IP which is designed for lossy, routeable, variable-speed, networks of millions of devices. It is not optimised for mostly lossless, switched, constant-speed links of thousands of nodes. So although hardware latencies may be similar between Infiniband and ethernet (although in Infiniband's favour), latencies measured using something like MPI tend to be much more in Infiniband's favour.

Typical MPI latency figures (2021) might be around 1μs for Infiniband and a little over 5μs for 10G Ethernet, although much will depend on the details of the interconnect's topology, and one's definition of latency. The hardware latencies for a single Infiniband switch can be as little as 100ns, but it is unusual to observe a latency of under 1μs at the MPI level.

How Difficult?

How difficult is MPI? I hope that this course convinces you that the answer is 'not very.'

In many cases MPI calls can be confined to a very small part of the code. It is quite likely that most functions need not be changed from their serial equivalents. And the number of different MPI calls one needs to make can be quite modest.

The last slide shows a complete MPI program for performing some rather naive 1D quadrature. It is under 20 lines, and contains five MPI calls.

As a method for integrating a particular Gaussian from 0 to 10, it scores very few marks. But as a demonstration that MPI code can be simple it works better.

(To fit it in 20 lines, a few short-cuts have been taken. But it will work sanely on any number of processors which is a factor of 27720...)

Not That Difficult

```
#include <stdio.h>
#include <math.h>
#include <mpi.h>
#define STEPS 27720

int main(int argc, char *argv[]){
  int rank,nproc,start,end,i;
  double total=0, result;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  start=rank*(STEPS/(double)nproc);
  end=(rank+1)*(STEPS/(double)nproc)-1;

  for(i=start;i<=end;i++)
     total+=exp(-pow(10.0*((i+0.5)/(STEPS+1)),2));
  printf("Total from process %d is %f\n",rank,total);

  MPI_Reduce(&total,&result,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
  if (rank==0) printf("Grand total %f, integral %f\n",
                              result, 10*result/STEPS);
  MPI_Finalize(); return 0;
}
```

The Beginning

What is MPI?

MPI, the Message Passing Interface, is a library for running jobs of multiple processes on multicore computers, and on multiple computers.

MPI was originally developed for the Fortran and C languages, and version 1.0 was released in 1994. It is developed by an international, vendor-neutral, consortium, www.mpi-forum.org, and is supported by a wide range of different vendors and platforms.

It has become the pre-eminent method for writing portable parallel programs on distributed memory systems. A well-written MPI code will run on a laptop and on a Cray, and perhaps even under MS Windows.

F90 and F2008 are now supported. C++ is not explicitly supported by the official MPI standard, but C++ programmers should have no difficulty using MPI in a C-like fashion.

I do not recommend the C++ interface to MPI – it was deprecated in version 2.2 (2009), and removed in version 3.0 (2012). If you see things starting '`MPI::`' rather than '`MPI_`' you are using the deprecated C++ binding.

'The C++ bindings add minimal functionality over the C bindings while incurring a significant amount of maintenance to the MPI specification. Since the C++ bindings are effectively a one-to-one mapping of the C bindings, it should be relatively easy to convert existing C++ MPI applications to use the MPI C bindings.' (Quote from MPI 2.2 standard justifying dropping C++.)

What isn't MPI?

MPI is not OpenMP. OpenMP assumes that there exists a large block of memory to which all tasks have direct access. Effectively this means a shared memory machine. MPI does not assume this – all data transfers between tasks need to be explicitly requested.

This means that MPI can run on shared memory computers, and on a collection of discrete computers joined by ethernet, or by something like ethernet only faster (e.g. Infiniband).

In practice this means that OpenMP works well scaling up to a dozen processes or so, not least because this is about as many cores as one can put economically in a single shared memory computer. MPI can scale to hundreds or thousands of cores. For best scalability one can even combine the two (OpenMP using all cores in a single node, then MPI between the multiple nodes of a large cluster).

In MPI the number of processes used is generally set when the program starts. In OpenMP the number of threads can vary.

This Course

This course is not intended to teach the whole of MPI 3.0. MPI 3.0 contains close to 300 functions. This course covers about 10% of them.

However, that 10% is quite enough to write perfectly good MPI code in a variety of different forms. It is also enough to work on most MPI codes. For instance, Castep is a big MPI code, with over 450,000 lines of source. It uses fewer than two dozen MPI functions.

Although this course mentions MPI 3.0, there is almost nothing in this course which is not in MPI 2.2.

MPI 3.1 (2015) is a minor update to 3.0 (2012). MPI 4.0 was standardised in June 2021.

Definitions

Some words have meanings which change with time and place. I shall try to keep these notes consistent with:

CPU meaning CPU core (unless otherwise specified) – the difference between a single quad core CPU, two dual core CPUs, and four single core CPUs for most purposes is slight, and dependent on more details than those statements provide.

Node meaning (part of) computer in which all CPUs can directly access all of the memory in the node.

Parallel computer – potentially multiple nodes, configured so that a single job can be run across all.

Using MPI

To use MPI one needs to make a few changes to one's source code, then link with MPI libraries at link time, and then run with a special program launcher traditionally called `mpirun`, and now often called `mpiexec`. If one is also using a queueing system, then there needs to be close interaction between the queueing system and the `mpiexec` command.

(The queueing system will reserve nodes for a job. It then needs to inform `mpiexec` which those nodes are, so that `mpiexec` tries to spawn its processes in the right places.)

Hello, World (C)

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[])
{
  MPI_Init(&argc,&argv);

  printf("Hello C parallel world.\n");

  MPI_Finalize();
  return(0);
}
```

Hello, World (C++)

```
#include <iostream>
#include <mpi.h>

int main(int argc, char *argv[])
{
  MPI_Init(&argc,&argv);

  std::cout << "Hello C++ parallel world.\n";

  MPI_Finalize();
  return 0;
}
```

Hello, World (F77)

```
program hello
include 'mpif.h'
integer ierr
call mpi_init(ierr)
write(*,*)' Hello F77 parallel world'
call mpi_finalize(ierr)
end
```

(Don't do this for new code)

Hello, World (F90)

```
program hello

use mpi
integer :: ierr

call mpi_init(ierr)

write(*,*)' Hello F90 parallel world'

call mpi_finalize(ierr)
end
```

(Don't do this for new code)

Hello, World (F2008)

```
program hello

use mpi_f08

call mpi_init()

write(*,*)' Hello F90 parallel world'

call mpi_finalize()
end
```

Better Fortran

MPI defines three methods of Fortran support:

1. `USE mpi_f08`: This method is described in Section 17.1.2. It requires compile-time argument checking with unique MPI handle types and provides techniques to fully solve the optimization problems with nonblocking calls. This is the only Fortran support method that is consistent with the Fortran standard (Fortran 2008 + TS 29113 and later). This method is highly recommended for all MPI applications.

2. `USE mpi`: This method is described in Section 17.1.3 and requires compile-time argument checking. Handles are defined as INTEGER. This Fortran support method is inconsistent with the Fortran standard, and its use is therefore not recommended. It exists only for backwards compatibility.

3. `INCLUDE 'mpif.h'`: This method is described in Section 17.1.4. The use of the include file mpif.h is strongly discouraged starting with MPI-3.0, because this method neither guarantees compile-time argument checking nor provides sufficient techniques to solve the optimization problems with nonblocking calls, and is therefore inconsistent with the Fortran standard. It exists only for backwards compatibility with legacy MPI applications.

The above quote comes from version 3 of the MPI Standard. Now is not the time to discuss why the old `use mpi` version broke the Fortran standard (perhaps a lecture next Term?), but You Have Been Warned. (That said, in practice it worked, and still works.)

MPI version 3 (2012) introduced `mpi_f08`, but you will still see plenty of things using the older version, including parts of this course...Note that the standard also states 'in a single application, it must be possible to link together routines which `USE mpi_f08`, `USE mpi`, and `INCLUDE 'mpif.h'`.'

Starting and stopping

All processes must call `MPI_Init` precisely once. It need not be the first executable statement, though it must precede (almost) all other MPI calls. It does not mark the point at which the program moves from being serial to parallel: `mpiexec` launches the specified number of processes from the beginning, and this number is fixed for the duration of the MPI program's execution.

C/C++ programmers may call `MPI_Init` as simply `MPI_Init(NULL,NULL)`.

All processes must call `MPI_Finalize` precisely once before exiting, and as the last MPI call they make. If a process exits without calling `MPI_Finalize`, the other processes will (probably) notice and abort.

A process may continue to run, performing no further MPI calls, after calling `MPI_Finalize`.

To avoid misleading, it should be added that if your code makes any use of threading (threaded maths libraries, using OpenMP, using GTK, etc.), then `MPI_Init` should be replaced by `MPI_Init_thread` (see later).

Compiling and Running

```
$ mpifort hello.f90
$ mpiexec -n 2 ./a.out
 Hello F90 parallel world
 Hello F90 parallel world
```

The name `mpifort` will need adjusting for your system and language: `mpif90`, `mpf90`, `mpicc`, `mpic++` etc.

The days when it was sensible to use the standard compiler and to add extra arguments such as `-lmpi` are gone.

The argument to `mpiexec` specifies the number of processes. This is generally set to the number of CPU cores in the machine used for production runs. For testing purposes one can set it to more or fewer.

Older systems may use `mpirun` instead of `mpiexec`:

```
$ mpirun -np 2 ./a.out
```

then is what is needed to start a job.

Success?

It is hard to tell, for with many MPI systems `mpiexec` will happily run multiple copies of non-MPI programs.

```
$ mpiexec -n 2 /bin/echo Hello serial world
Hello serial world
Hello serial world
```

So we need a slightly more complicated (and more useful) example.

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[])
{
  int rank,nproc;

  MPI_Init(&argc,&argv);

  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  printf("Hello C parallel world, I am process %d out of %d\n",
                                rank,nproc);
  MPI_Finalize(); return 0;
}
```

```
#include <iostream>
#include <mpi.h>

int main(int argc, char *argv[])
{
  int rank,nproc;

  MPI_Init(&argc,&argv);

  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  std::cout << "Hello C++ parallel world, I am process " << rank <<
    " out of " << nproc << "\n";

  MPI_Finalize(); return 0;
}
```

```
program hello

  use mpi_f08
  integer :: npe,mype

  call mpi_init()

  call mpi_comm_size(mpi_comm_world, npe)
  call mpi_comm_rank(mpi_comm_world, mype)

  write(*,101) mype,npe
101   format(' Hello F90 parallel world, I am process ', &
       I3,' out of ',I3)

call mpi_finalize()
end
```

C vs Fortran

In C all MPI calls are functions returning an error code. Their names are case-sensitive, and start `MPI_`. This is what a C programmer would expect.

In Fortran all MPI calls are subroutines. They have one additional final argument compared to their C counterparts, which is used for the return code. Their names are case insensitive. If using `mpi_f08` the return code argument is optional, and was omitted in the above example.

One call differs: `mpi_init` has arguments of `&argc` and `&argv` in C only, so that all processes see the same command-line arguments and can parse them.

Almost all MPI calls have an argument of a 'communicator'. For these simple examples this will always be the constant `MPI_COMM_WORLD` which means all processes are involved.

For the moment we shall ignore the error codes entirely. Later (page 102) we will try justifying this approach.

One is strongly recommended to avoid using any names starting `MPI_` or `PMPI_` for one's own variables / functions / subroutines.

Useful

```
$ mpifort hello.mpi.f90
$ mpiexec -n 3 ./a.out
 Hello F90 parallel world, I am process   2 out of   3
 Hello F90 parallel world, I am process   0 out of   3
 Hello F90 parallel world, I am process   1 out of   3
```

The call `mpi_comm_size` states how many processes are in this job. This will be set when the job is launched, probably via the `-n` argument to `mpiexec`.

The call `mpi_comm_rank` returns the rank of that particular process within the job. All processes will get a different result, and they are guaranteed to run from zero to one less than the size of the communicator in the above call. This further implies that process zero in `mpi_comm_world` is guaranteed to exist.

More Notes

Note that here all processes can do I/O, even to shared units / file handles such as stdout (although this is not guaranteed by the MPI standard, in practice it is rare to find a counter-example).

Note that the order of output is not defined. Nor is the order of execution precisely defined. MPI code is *not* 'lock step' – it is not the case that all tasks execute the same line of the source at the same time. Synchronisation, if required, must be sought explicitly.

An unexpected stop, the equivalent to a serial code's
`if (bad_thing) exit(error_code)`
is
`if (bad_thing) MPI_Abort(MPI_COMM_WORLD, error_code)`
and may be called on any process to bring the whole job to a slightly undignified end. If one's code needs at least a certain number of processes, then one solution is

```
if ((rank==0)&&(nproc<2)){
  fprintf(stderr,"Too few processes\n");
  MPI_Abort(MPI_COMM_WORLD,1);
}
```

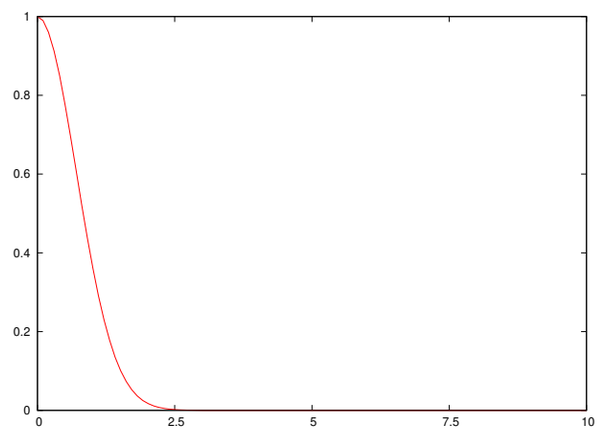
Something More Useful

The next example evaluates

$$\int_0^{10} \exp(-x^2)\,\mathrm{d}x$$

by approximating this as the sum of the function at equispaced intervals excluding the end points, suitably normalised.

$$\int_0^{10} \exp(-x^2)\,\mathrm{d}x \approx \int_0^{\infty} \exp(-x^2)\,\mathrm{d}x = 0.5\sqrt{\pi}$$



```c
#include <stdio.h>
#include <math.h>
#include <mpi.h>
#define STEPS 27720

int main(int argc, char *argv[]){
  int rank,nproc,start,end,i;
  double total=0, result;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  start=rank*(STEPS/(double)nproc);
  end=(rank+1)*(STEPS/(double)nproc)-1;

  for(i=start;i<=end;i++)
     total+=exp(-pow(10.0*((i+0.5)/(STEPS+1)),2));
  printf("Total from process %d is %f\n",rank,total);

  MPI_Reduce(&total,&result,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
  if (rank==0) printf("Grand total %f, integral %f\n",
                              result, 10*result/STEPS);
  MPI_Finalize(); return 0;
}
```

Fortran

```
program hello
  use mpi_f08
  implicit none
  integer, parameter :: steps=27720
  integer :: nproc,rank,ierr,i,start,end
  real(kind(1d0)) :: total, result

  call mpi_init()
  call mpi_comm_size(mpi_comm_world, nproc)
  call mpi_comm_rank(mpi_comm_world, rank)

  total=0
  start=rank*(steps/real(nproc,kind(1d0)))
  end=(rank+1)*(steps/real(nproc,kind(1d0)))
  do i=start,end-1
    total=total+exp(-(10d0*(i+0.5d0)/(steps+1))**2)
  enddo
  write(*,101)rank,total
101 format('Total from process ',i3,' is ',g16.8)

  call mpi_reduce(total,result,1,mpi_double_precision,mpi_sum,0, &
       mpi_comm_world)

  if (rank.eq.0) write(*,102) result,10*result/steps
102   format('Grand total is ',g16.8,' and integral is ',f13.9)

  call mpi_finalize()
  end
```

Success!

At some cost to style and sanity, we have an MPI code which does something useful in under two dozen lines. There are a few points to note.

Defining the number of steps as a compile-time constant is ugly, and the magic number of 27720, chosen to avoid (unnecessary?) worries about remainders when dividing by integers ≤ 12, is madness.

Each process has genuinely done different, useful, work towards this integral. MPI provides a collection of useful *reduction* operations for collecting results spread in common fashions.

```
$ mpicc int1.c -lm
$ mpiexec -n 4 ./a.out
Total from process 1 is 1.000246
Total from process 2 is 0.000000
Total from process 3 is 0.000000
Total from process 0 is 2455.709415
Grand total is 2456.709660 and integral is 0.886259
```

MPI_Reduce

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
     MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

The two buffers (send and receive) must be distinct. The count is the number of data items on each node – here just one. The root task is the one which receives the result on the reduction operation, and on which the receive buffer will be filled with useful data.

The reduction operation can be one of MPI_SUM, MPI_PROD, MPI_MAX, MPI_MIN, MPI_MAXLOC, MPI_MINLOC and some generally less interesting bitwise and logical operations.

(Note that MPI_MAXLOC and MPI_MINLOC return both location and value, so read some documentation regarding the datatype you need to provide as the receive buffer!)

In Fortran, the datatype and operation will be probably be integer (parameters).

MPI Datatypes

Specifying the correct datatype is always important (so that send or receive buffers have their lengths calculated correctly). With reduction operations it is particularly important, and mixing ints and floats will produce nonsense. The main datatypes available are:

C: `MPI_CBOOL MPI_CHAR MPI_SHORT MPI_INT MPI_LONG MPI_LONG_LONG`
`MPI_UNSIGNED_CHAR MPI_UNSIGNED_SHORT MPI_UNSIGNED ...`
`MPI_INT8_T MPI_INT16_T MPI_INT32_T MPI_INT64_T` also `MPI_UINT8_T` etc.
`MPI_FLOAT MPI_DOUBLE`
`MPI_C_COMPLEX MPI_C_DOUBLE_COMPLEX`

C++: As C, and add `MPI_CXX_BOOL`
`MPI_CXX_FLOAT_COMPLEX MPI_CXX_DOUBLE_COMPLEX`

Fortran: `MPI_LOGICAL MPI_CHARACTER MPI_INTEGER`
`MPI_REAL MPI_DOUBLE_PRECISION`
`MPI_COMPLEX MPI_DOUBLE_COMPLEX`

The standard requires that one uses the correct datatype, including language, even though often one will often get away with using the wrong one, such as using `MPI_DOUBLE` in Fortran.

Complex support in C/C++ is a mess. In C, it requires MPI 2.2.

`MPI_C_FLOAT_COMPLEX` is a synonym for `MPI_C_COMPLEX`.

Do not think of passing pointers. As the memory layout of different processes is very likely to be different, even though the same executable is being run, the pointer would have no meaning on a different MPI rank.

A Master

In this example there is a weak sense of a master process. Only process zero receives the final result, and only process zero prints it.

Convention gives such tasks to process zero, but there is often no reason not to choose any other rank, save that choosing a rank which does not exist will lead to errors. . . (Some systems permit I/O from the rank zero process only. It is very rare, though permitted by the standard, for a system not to permit I/O from rank zero.)

If one wishes to be more egalitarian, there is an MPI function called `MPI_Allreduce` It has the same arguments as `MPI_Reduce`, save that the root argument is absent, for now the final answer will appear on all processes. It is essentially equivalent to `MPI_Reduce` followed by broadcasting the result back to all processes.

Note that OpenMP is much less egalitarian. In OpenMP thread zero executes all the code, and the other threads execute in the parallel regions only. In MPI all processes execute all the code, and if inequality is wanted it has to be introduced via statements similar to

```
if (rank==0)...
```

MPI vs OpenMP (again)

If one compares an n process MPI job and an n process OpenMP job, then, in general, some of the differences in execution are as follows.

The OpenMP process will spawn $n - 1$ extra threads at some point after startup. It will behave as though these extra threads are spawned on entry to every parallel region, and destroyed at the end of each region, although it is likely that they are retained, idle, outside of the parallel regions for efficiency.

The MPI job will be started by `mpiexec`. This will launch n processes, and then continue to run as a 'shepherd' process until the job exits. If some of the processes are on remote nodes, then `mpiexec` will have to use some form of remote launch, perhaps simply ssh over ethernet, perhaps something more efficient. It will need to ensure that the processes can communicate with it, and with each other. How this is achieved can vary greatly, depending partly on what mechanisms might be available. The programmer should not have to worry, and should assume that the MPI system has set up a near-optimal communications network.

The `mpiexec` process may handle output to `stdout` and `stderr` from the processes it launches.

What happens if an MPI executable is launched without using `mpiexec` (or `mpirun`) is not specified. It might behave as though it were a single-process MPI job, and this is the recommendation (but not requirement) of the current MPI standard.

Which Version?

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[])
{
  int major,minor;

  printf("Compile-time MPI version is %d.%d\n",MPI_VERSION,
         MPI_SUBVERSION);

  MPI_Get_version(&major,&minor);
  printf("Run-time MPI version is %d.%d\n",major,minor);
  return 0;
}
```

The absence of a call to `MPI_Init` is correct – `MPI_Get_version` is one of the very few MPI functions which can be called before or after `MPI_Init`.

MPI 3.0 (Sept 2012) introduces the Fortran 2008 bindings. Version 2.2 (Sept 2009) lacks them.

In Fortran

```
program version
  use mpi_f08
  integer major,minor,ierr
  write(*,*)'Compile time MPI version is ',MPI_VERSION,MPI_SUBVERSION
  call mpi_get_version(major,minor)
  write(*,*)'Run time MPI version is ',major,minor
  end
```

(With few marks for the neatness of the output.)

Older Fortran code will use `use mpi`. There is then a compulsory final argument of `ierr` for almost all routines, and the MPI-specific user-defined types in arguments are replaced by integers. There is an incomplete table of the differences on slide 304 for reference.

Whose MPI?

There are two major open source versions of MPI: OpenMPI (not to be confused with OpenMP), and MPICH. There are also several commercial versions of MPI.

All standard-conforming versions of MPI will conform to the MPI standard. But they need not behave identically. Unfortunately the MPI standard specifies several constants, some of which we have seen already, such as `MPI_COMM_WORLD`, `MPI_DOUBLE` and `MPI_SUM`. It does not specify what numeric values they should take (or even that they are numeric types in many cases), and different implementations make different choices. As for how `mpiexec` interacts with an MPI executable to cause multiple processes to be launched and tied together, the standard says very little.

So it is very important that the same MPI is used at compile time, link time and run time. Otherwise chaos results.

OpenMPI: `www.open-mpi.org` (founders: Los Alamos, Unis of Tennessee, Indiana & Stuttgart)
MPICH: `www.mpich.org` (founder: Argonne)

Many commercial MPIs are based on MPICH, although OpenMPI is the more widely used free implementation. LAM/MPI used to exist, but is now officially dead with parts merged into OpenMPI.

Memory Management in C

As MPI supports C rather than C++, a couple of slide on how C does memory management may be useful. There is nothing wrong with using C++'s `new` and `delete` with MPI, but you will see examples using C-style memory management.

C's memory management routines are simple, basic, and therefore easy to understand. There are three operations:

Give me a block of memory
I no longer want a block of memory
I want this block of memory resized

All sizes are in bytes. There is no record or concern of what type of data lives in which block. There is no automatic deallocation at the end of a function (or scoping unit).

An Example

```
#include<stdlib.h>

int *a;
a=malloc(1000*sizeof(int));
```

Now `a[0]` to `a[999]` can be freely used.

```
free(a);
```

And now no reference to `a` is valid (save for an assignment to it). In particular it is not permitted to free `a` again. One cannot even test whether `a` is a valid pointer.

(One should always use `sizeof`, and not assume that an `int` is four bytes. If one has structures, then `sizeof(struct mystruct)` works fine.)

A Better Example

```
int *a;
a=malloc(1000*sizeof(int));
if (!a){
  fprintf(stderr,"Disaster! Malloc failed!\n");
  exit(1);
}
```

If malloc cannot fulfill a request it will return a null pointer. One should *always* test for this. Note `if(!a)` is equivalent to `if(a!=NULL)`

```
free(a);
a=NULL;
```

After freeing `a`, unless `a` is immediately going out of scope, it is best to set it to the null pointer. Trying to reference `a` after it has been freed is an error. But it will almost certainly work. What happens is not defined, but immediate use is likely to find whatever it used to point to still in place, but later some other structure will probably occupy the same memory...

Setting `a` to `NULL` ensures that future erroneous access attempts produce an immediate crash. Far safer.

Growth

```
a=realloc(a,2000*sizeof(int));
if (!a){
  fprintf(stderr,"Disaster! Malloc failed!\n");
  exit(1);
}
```

This resizes (larger or smaller) a memory block. It may also move the block, so be very careful if there are other pointers pointing to elements within the block. If `a` is initially `NULL`, then `realloc()` acts like `malloc()`. If the size is zero, then `realloc()` acts like `free()`. In this case it may return a null pointer, or it may return a pointer to a zero-length block of memory. In the second case that pointer can be passed to `realloc()` or `free()`. If `a` has been freed, or was not returned by `malloc()` or `realloc()`, it is an error.

```
for(i=0;i<10;i++) a=malloc(1000);
```

is an example of unwanted growth. Ten blocks of 1000 bytes are allocated, but the pointers to nine of them are discarded without any attempt to free them. These nine blocks of memory are now inaccessible and unfreeable, but they will take up space until the program exits. This is called a memory leak. Don't do this!

Functions must explicitly free any memory they have allocated when they exit, unless they have done something to preserve a pointer to the memory so that it can be used, or freed, in the future (e.g. they return the pointer to the caller).

Magic

The programmer cannot tell whether a pointer is valid, nor how large the block it points to is. If you wish to pass a pointer to an array to a function, it is likely that you will also need to pass its length separately for it to be of any use. (Not true in Fortran when used sensibly, was also true in Fortran 77 and earlier.)

Annoyingly the C library will know whether a pointer is valid, and will know how large a memory block it is associated with.

There is no concept of reference counting or garbage collection. No-one knows when there are no longer any pointers pointing to a memory block, and such an orphaned block will never be freed, and can never be accessed.

Code will crash if it attempts to access memory not allocated to it by the OS. But an invalid use of a pointer which results in accessing memory which has been allocated to the program by the OS will be undetected, but will almost certainly lead to wrong results and later chaos. (Equally true in C++ and Fortran!)

The OS allocated memory to processes in multiples of pages, generally 4KB. So near misses to the end of an array are unlikely to be detected.

The return type of `malloc()` is a void pointer, so may be assigned to any other pointer type. Pedants may do this explicitly.

```
a=(int*)malloc(1000*sizeof(int));
```

Completeness

C provides a couple of other utility functions for dealing with blocks of memory.

`memmove(void *dest, const void *src, size_t n);`

copy `n` bytes from `src` to `dest`.

`memset(void *dest, int c, size_t n);`

set the first `n` bytes of `dest` to the value `c`, generally used with `c=0` to zero memory. Note `0<=c<=255`.

`calloc(size_t nmemb, size_t size);`

allocate and clear, equivalent to

```
a=malloc(nmemb*size);
if (!a) memset(a,0,nmemb*size);
return(a);
```

The Common Errors

```
int *a;
a=malloc(4*sizeof(int));
a[4]=6;
```

just as invalid as

```
int a[4];
a[4]=6;
```

but even less likely to be spotted by a compiler. (`a[0]` to `a[3]` are the four valid elements of `a`.)

```
int *a;
a=malloc(4*sizeof(double));
```

This is not actually an error. But it is almost certainly a bug. The compiler is very unlikely to check that the type of the pointer (`int`) matches the type of the argument to `sizeof` (`double`), yet it almost always should.

```
int *a;
a=malloc(16); // sizeof(int) is always 4, isn't it?
```

is also valid but dangerous. Don't do this! Omit `sizeof` only when the datatype is bytes, e.g. image data or compressed data.

More Quadrature

More Quadrature

The previous example of quadrature used a very simple sum. One might wish to use a more sophisticated quadrature algorithm, perhaps even one sufficiently sophisticated that you don't wish to write it yourself. This is fine. Indeed, the quadrature routine you use need not know anything about MPI itself.

Quadrature with GSL

The Gnu Scientific Library, which is effectively C-only, has a quadrature routine which is used in serial code as follows.

Note that to link with GSL (assuming it is installed) one will need a command similar to

```
$ gcc myprog.c -lgsl -lgslcblas -lm
```

```
int gsl_integration_qag (const gsl_function *f, double a, double b,
   double epsabs, double epsrel, size_t limit, int key,
   gsl_integration_workspace *workspace, double *result,
   double *abserr)
```

evaluates the integral of `f` from `a` to `b` with an absolute error of `epsabs` and a relative error of `epsrel` making a maximum of `limit` bisections using method `key` with a user-supplied workspace. The result and its estimated error are returned in the final two parameters.

In theory a C function like this is callable from Fortran using the `iso_c_binding` interface. In practice it is messy, so left to the exercises.

GSL

```
#include<stdio.h>
#include<math.h>
#include<gsl/gsl_errno.h>
#include<gsl/gsl_integration.h>

double g(double x, void *v){return exp(-x*x);}

int main(){
  double result,err,xmin,xmax,tol;
  gsl_integration_workspace *work;
  gsl_function F;

  work=gsl_integration_workspace_alloc(100000);
  F.function=g;  F.params=NULL;
  xmin=0;  xmax=10;  tol=1e-4;

  gsl_integration_qag(&F,xmin,xmax,0,tol,100000,GSL_INTEG_GAUSS15,
                      work,&result,&err);

  printf("Integral is %f\n",result);
  gsl_integration_workspace_free(work);
  return(0);
}
```

Parallelising

To parallelise this using MPI, all the modifications are made to `main()` (save for the `#include<mpi.h>` statement).

No modifications are needed to the MPI-unaware GSL library, or to the function `g`.

This will be a recurring theme. Writing an MPI code does not mean that every function, or even most functions, need to differ from their serial forms.

There is one caveat: some maths libraries are threaded, and often automatically use as many threads as there are processor cores in the node. This interacts badly with launching as many MPI processes as there are processor cores in the node. A 32-core node might end up with 1024 threads trying to run! This will be very inefficient, and, anyway, to use threading at all in an MPI program one needs to replace the call to `MPI_Init` by `MPI_Init_thread`. Many MPI installations attempt to reduce this problem by setting the environment variable `OMP_NUM_THREADS` to 1 (if not already set to a different value), as many threaded libraries will use that variable to determine how many threads to use.

Serial libraries are always fine.

```
int main(int argc,char **argv){
  int rank,nproc;
  double result,err,xmin,xmax,tol,total;
  gsl_integration_workspace *work;
  gsl_function F;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  work=gsl_integration_workspace_alloc(100000);
  F.function=g;  F.params=NULL;
  xmin=rank*(10.0/nproc);  xmax=(rank+1)*(10.0/nproc);  tol=1e-4;

  gsl_integration_qag(&F,xmin,xmax,0,tol,100000,GSL_INTEG_GAUSS15,
                      work,&total,&err);

  MPI_Reduce(&total,&result,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);

  if (rank==0) printf("Integral is %f\n",result);

  gsl_integration_workspace_free(work);
  MPI_Finalize();
  return(0);
}
```

Error Accumulation

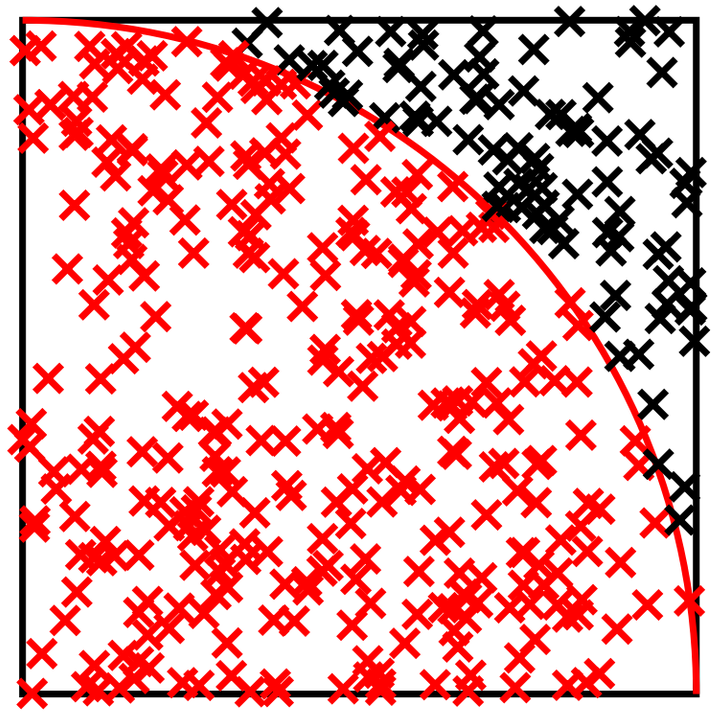
A small amount of thought needs to be given to error accumulation. If one wishes the integral to be calculated with an absolute error of no more than ϵ, then breaking it into `nproc` regions, calculating in each region to an absolute error of ϵ, and then summing the results, is the wrong answer.

Depending on your beliefs with respect to the errors, the absolute error in each region ought to be $\epsilon/\sqrt{\texttt{nproc}}$, or maybe $\epsilon/\texttt{nproc}$.

As one is using the same integration technique in all regions, and the function may be of similar form in all regions, assuming that the errors in each region are independent may be unwise.

More pi?

Another calculation we can perform which has an answer related to π is to pick points at random in the unit square $0 \leq x \leq 1$ (and similarly for y) and see what fraction lie within unit distance of the origin. The fraction should be $\pi/4$.



$\pi \approx 3.09$

Serial

```
#include<stdio.h>
#include<stdlib.h>

#define COUNT 100000

int main(){
  int i,hit=0;
  double x,y;

  for(i=0;i<COUNT;i++){
    x=rand()/(double)RAND_MAX;
    y=rand()/(double)RAND_MAX;
    if (x*x+y*y<1.0) hit++;
  }

  printf("Ratio: %f\n",hit/(double)COUNT);

  return 0;
}
```

Not Parallel

```
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
#define COUNT 100000

int main(int argc,char **argv){
  int i,nproc,rank,hit=0;
  double x,y,t,result;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  for(i=0;i<COUNT;i++){
    x=rand()/(double)RAND_MAX; y=rand()/(double)RAND_MAX;
    if (x*x+y*y<1.0) hit++;
  }
  t=hit/(double)COUNT;
  MPI_Reduce(&t,&result,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
  if (rank==0) printf("Ratio: %.9f\n",result/nproc);

  MPI_Finalize();  return 0;
}
```

An Error

It appears that the above is coded to perform `COUNT * nproc` samples. It isn't.

C's `rand` function will use the same seed on all processes, so each process will use precisely the same random points for its sampling, and get precisely the same answer. This is not what is wanted at all for Monte Carlo sampling.

One should explicitly set the random number seed to different values on the different processes. In C, simply calling `srand(rank)` would suffice.

(As `srand()` on some versions of Linux returns the same sequence after calls to `srand(0)` and `srand(1)`, it might be preferable to call `srand(rank+1)`.)

An Aside

C's `rand()` function is pretty dreadful – it is likely that the sequence returned repeats after 2^{32} values (or maybe 2^{31}), and, on some systems, there are merely 2^{16} different values returned. It is also not thread-safe, though this is not a problem for MPI. However, on some systems, including Linux, `rand()` is much better than the minimum specification allowed.

If using Fortran, the initial state of `random_number` is not defined, so running a program multiple times may always produce the same random sequence, or may produce different sequences. If one cares, one needs to set the seed explicitly.

Generating 'good' random numbers can be almost as hard as defining what one means by 'good'.

Reproducibility

For ease of debugging, it is best if an MPI program produces the same results regardless of the number of processors it is run on. For Monte-Carlo simulations, this is unlikely to be possible. All results may be correct, without being identical.

If one is desperate, for some algorithms techniques such as writing a lot of random numbers to a file, and then having each rank read in its random numbers with a step of `nproc`, or, maybe, from an offset in the file of `rank` $\times$ (number of randoms numbers one rank will read) may work to produce results independent of the number of processes. However, for other Monte-Carlo techniques such tricks do not work at all.

Perfect reproducibility is unlikely to be achieved due to the non-associative nature of finite precision arithmetic. If one is calculating the sum of a large array, the sum is likely to depend on the order of summation. An extreme example is

$$\sum_{n=1}^{\infty} \frac{1}{n}$$

which is perfectly convergent (but to very different values) in single and double precision arithmetic, and diverges in reality. If one spreads an array across a varying number of processes, and then sums it, the order of summation is almost certain to depend on the number of processes used, and thus too the result.

Do What?

In most of the example code in this lecture, all the parameters for specifying the problem (range of integration, tolerance, iterations, etc.) are hard coded into the source, and to change any one needs to recompile. This is not how 21st century code is meant to be written.

Saner code would read such parameters from the command line, from stdin, or from an input file, and thus be capable of different calculations without being recompiled.

In MPI there are two possibilities: read the input file on all processes, or read it on just one, and broadcast the required data to the other processes. In general the latter is preferred, especially as scaling can be an issue with the other approach. (How impressed will your fileserver be if a thousand MPI processes simultaneously attempt to open and read the same input file? A good fileserver won't mind much, but a cheaper one might.)

Broadcast

A broadcast is the opposite of a reduction operation. Just one process starts with the data, and all end up with the data. One might regard it as synchronising a variable (or array) to the values on a given master process.

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm)
```

Note that on the root process the buffer is a send buffer, and is read, and on other ranks it is a receive buffer, and is written.

```c
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>

int main(int argc,char **argv){
  int i,nproc,rank,hit=0,count;
  double x,y,t,result;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  if (rank==0) count=atoi(argv[1])/nproc;
  MPI_Bcast(&count,1,MPI_INT,0,MPI_COMM_WORLD);

  srand(rank+1);
  for(i=0;i<count;i++){
    x=rand()/(double)RAND_MAX;
    y=rand()/(double)RAND_MAX;
    if (x*x+y*y<1.0) hit++;
  }
  t=hit/(double)count;
  MPI_Reduce(&t,&result,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);

  if (rank==0) printf("Ratio: %.9f\n",result/nproc);
  MPI_Finalize(); return 0;
}
```

An Example

And therefore generally somewhat contrived.

Clearly the value for `count` could have been calculated on each process individually, but we chose to set it on the process with rank zero only, then broadcast it.

MPI programs can accept command-line arguments just like any other program:

```
$ mpiexec -n 2 ./a.out 10
Ratio: 0.800000000
$ mpiexec -n 2 ./a.out 100000
Ratio: 0.786520000
```

$(\pi/4 = 0.785398)$

More Collectivism

So far we have seen reduction (one data item on all processes reduced, probably via summation, to one data item on root process) and broadcast (one data item on root process sent to all processes).

Sometimes it is useful for the root process to gather in results from all other processes, and thus receive `nproc` times as much data as a single process sends, and even to do the reverse, to take a vector of data on the root process, and to give each rank a different single element each.

MPI provides for this, though one should be slightly wary of the memory requirements on the root process. It is probably fine for dealing with scalars, but one can send large arrays from all ranks to root...

Gathering and Scattering

```
MPI_Gather (void *sendbuff, int sendcount, MPI_Datatype sendtype,
            void *recvbuff, int recvcount, MPI_Datatype recvtype,
            int root, MPI_Comm comm)
MPI_Scatter(void *sendbuff, int sendcount, MPI_Datatype sendtype,
            void *recvbuff, int recvcount, MPI_Datatype recvtype,
            int root, MPI_Comm comm)
```

It is generally an error for `sendtype` and `recvtype` to differ, or for `sendcount` and `recvcount` to differ.

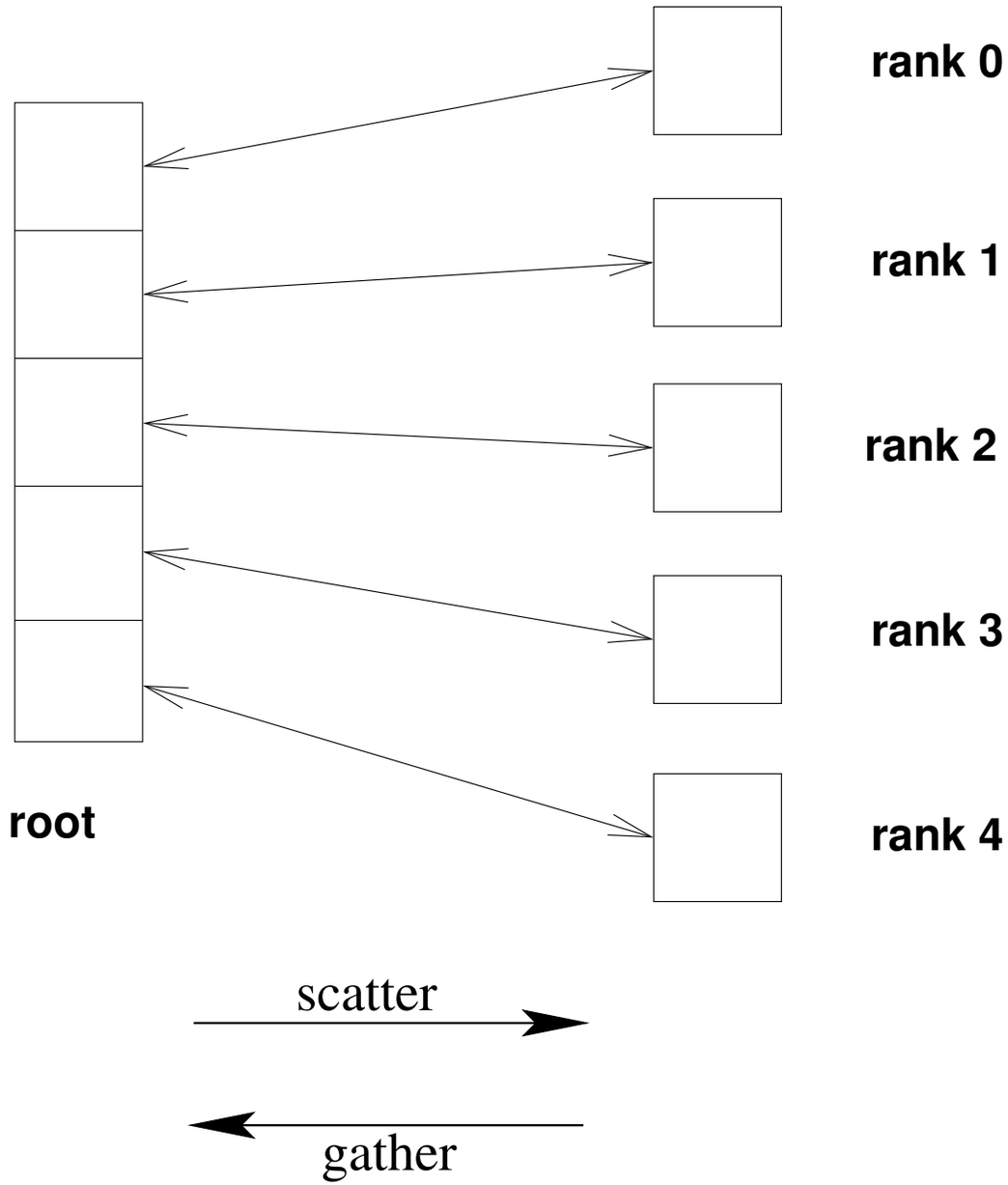
On the root process `sendbuff` and `recvbuff` must differ.

Gather: on root, the first `recvcount` elements of `recvbuff` are set to the first `sendcount` elements of `sendbuff` on process rank zero, the next `recvcount` elements of `recvbuff` to the contents of `sendbuff` on process rank one, etc.

Scatter: the first `sendcount` elements of the `sendbuff` on root are sent to process rank zero, the next `sendcount` elements to process rank one, etc.

Note that root does not have to be rank zero, and will send to itself (presumably optimised to a simple copy).

Gathering and Scattering Pictures



Gathering Example

```
double *all_results;

[...]

if (rank==0) all_results=malloc(nproc*sizeof(double));

[...]

MPI_Gather(&total,1,MPI_DOUBLE,all_results,1,MPI_DOUBLE,
           0,MPI_COMM_WORLD);
  if (rank==0)
    for(i=0;i<nproc;i++)
      printf("Total from process %d is %f\n",i,all_results[i]);
```

(Based on example on page 58. Also add `#include <stdlib.h>` to include the prototype for `malloc`.

Of course doing this followed by an `MPI_Reduce` on `total` is madness – the `MPI_Reduce` should now be replaced by a sum on the root process.)

More Detail

For `MPI_Gather`, the three recv variables are relevant only on the root process. Similarly for `MPI_Scatter`, the three send variables are relevant only on the root process.

Just as there was an `MPI_Allreduce` corresponding to `MPI_Reduce`, save that it lacked the root process argument for the result of the reduction appears on all processes, so there is an `MPI_Allgather` which lacks the root argument and fills the receive buffer on all ranks.

For the case in which the count of data elements to scatter or gather is not the same for all ranks, `MPI_Scatterv` and `MPI_Gatherv` exist.

`Gatherv` and `Scatterv`

```
MPI_Gatherv(void *sendbuff, int sendcount, MPI_Datatype sendtype,
            void *recvbuff, int recvcounts[nproc], int offsets[nproc],
            MPI_Datatype recvtype, int root, MPI_Comm comm)
MPI_Scatterv(void *sendbuff, int sendcounts[nproc], int offsets[nproc],
             MPI_Datatype sendtype, void *recvbuff, int recvcount,
             MPI_Datatype recvtype, int root, MPI_Comm comm)
```

For `MPI_Gatherv` note that `sendcount` on rank `i` should match `recvcount[i]` on rank `root`. It is not required that `recvbuff` be filled in a contiguous manner, or that the ranks fill it sequentially. It is required that elements in `recvbuff` are written no more than once by this call (i.e. the counts and offsets must not create overlaps), but this may mean that rank `nproc-1` writes the first elements, then there is a gap, then ranks zero to `nproc-2` fill in the rest. (Though it is unclear why one would wish to do this.)

For `MPI_Scatterv` note the perhaps surprising restriction that elements in `sendbuff` may be read no more than once.

There also exists `MPI_Allgatherv`.

Fortran users have to think in terms of zero-indexed arrays, so an offset of zero (not one) means the first element, and `sendcounts(1)` refers to rank zero, etc. They may choose to define these arrays so that their indices do start at zero.

Null and void

C programmers will happily pass the null pointer for those pointer arguments which are not used. Fortran programmers have no null pointer, and it is best if they pass an array of the correct type, but of any length, quite possibly an array called `dummy` of length one.

C programmers need not gloat too much. Because the prototypes for all data buffers in MPI calls have them as void pointers (because C has no concept of the type of one argument depending on the value of another) C's argument checking will never catch

```
int *total;
[...]
MPI_Gather(total,1,MPI_DOUBLE, [...]
```

Indeed, nothing will catch the above, and garbage will be generated at run-time.

Fortran programmers are no better off.

In Place

We have stressed that the send and receive buffer arguments must always differ, a restriction that arises from Fortran's (sensible) restriction that arguments must be distinct.

It is still possible to perform many collective operations in place, by passing the special value `MPI_IN_PLACE` as one of the buffers (always the smaller if they differ in size!), and the datatype and count of that buffer are ignored.

For `MPI_Gather` and `MPI_Reduce` this value is used for the send buffer.

For `MPI_Scatter` this value is used for the receive buffer.

In Place Reduction Example

WRONG:

```
double result;
[...]
MPI_Reduce(&result,&result,1,MPI_DOUBLE,MPI_SUM,root,MPI_COMM_WORLD);
```

CORRECT:

```
if (rank==root)
  MPI_Reduce(MPI_IN_PLACE,&result,1,MPI_DOUBLE,MPI_SUM,root,MPI_COMM_WORLD);
else
  MPI_Reduce(&result,NULL,1,MPI_DOUBLE,MPI_SUM,root,MPI_COMM_WORLD);
```

The wrong form will not produce an error at compile time or run time, and is even likely to produce correct results. However, it might also produce incorrect results sometimes, so don't do it!

Note that the correct version has the send buffer as `&result` on all processes except the root of the reduction, which has `MPI_IN_PLACE`, and the receive buffer on all processes save the root process is unused as usual. The temptation to write simply

```
MPI_Reduce(&result,MPI_IN_PLACE,1,MPI_DOUBLE,MPI_SUM,root,MPI_COMM_WORLD);
```

on all ranks must be avoided.

Error Handling

In the examples in this course the return code from MPI calls is never checked. There is an excuse for this laziness. MPI defaults to regarding errors as fatal, which means there is no need to check the return code. If it is not `MPI_SUCCESS`, the call will have caused the program to abort.

If one wishes to check return codes, then it is necessary to call

```
MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN)
```

Then one is under some obligation to check every MPI return code to stop undetected errors propagating chaos. So the antidote to the above is to restore the default position.

```
MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_ARE_FATAL)
```

If checking error codes, it may be bad style to assume that `MPI_SUCCESS` has the value zero, but it is a rare example of an MPI constant whose value is defined by the standard, and it is zero.

The Mandelbrot Set

The Mandelbrot Set

The ubiquitous example for parallel programming, and much else besides. Named after Benoit Mandelbrot for his work on the set at IBM c. 1980, although an ASCII-art representation had been published two years earlier by Brooks and Matelski.

In the unlikely event you have not met it before, consider the recurrance relationship

$$z_{n+1} = z_n^2 + z_0$$

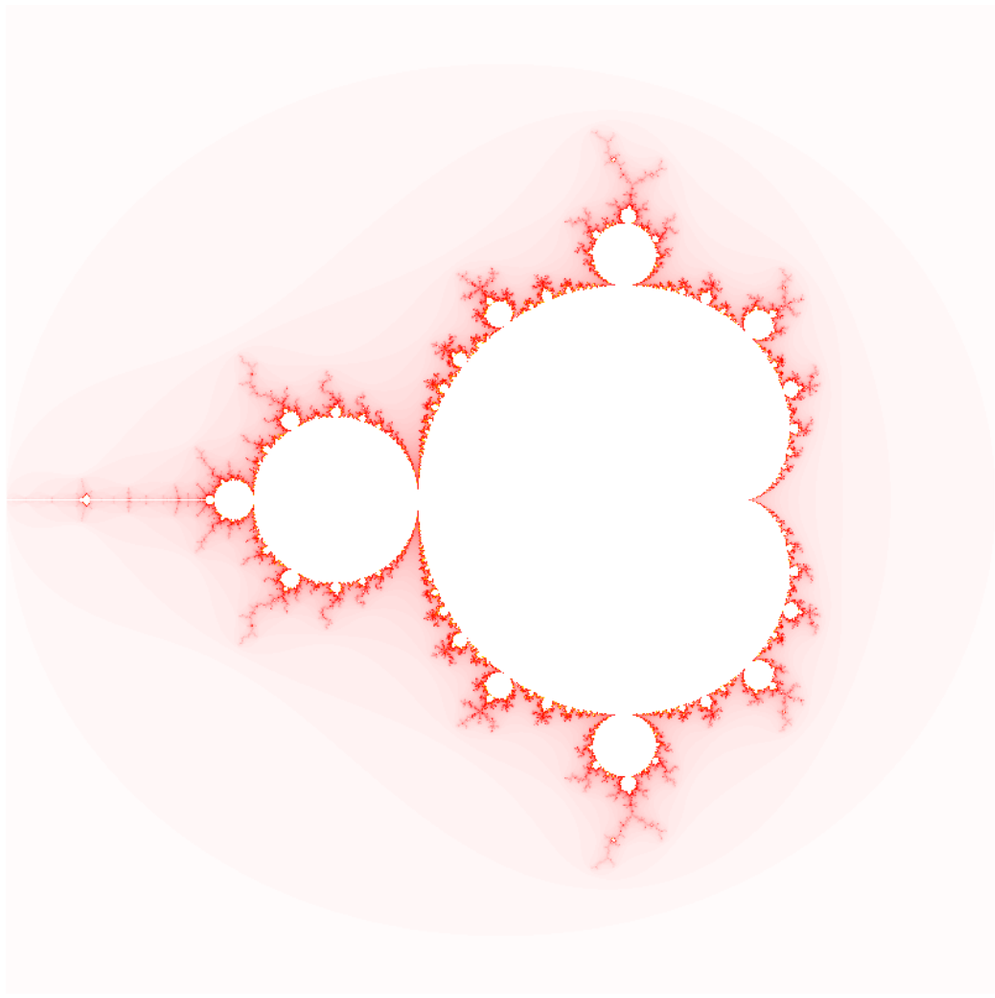
The Mandelbrot set is simply the values of z_0 in the complex plane for which z_∞ is finite.

Fortunately one can show that if $|z_n| > 2$ for any n then z_∞ is infinite. So in practice one merely iterates the above formula a few hundred times to see if $|z_n|$ exceeds two.

In Serial

We shall first consider this problem using simple serial code. We shall use C99 which, unlike C89 has support for complex numbers.

We shall also output a pretty picture, choosing the colour of each point based on the value of n for which $|z_n|$ first exceeds two.



Pretty Pictures

People often imagine producing pictures involves calling complicated libraries. Producing sane, compressed, images, such as png files, does. However, for good reasons we are not going to do that, but rather we shall write pam files directly. A pam file is an uncompressed bitmap (huge – don't store, best not written to a remote filesystem save for high performance fileservers). It is quite simple to write. Its format is:

```
P6
[width] [height] 255
[binary data, red green blue values, 3 x width x height bytes]
```

where [width] and [height] are integers expressed in ASCII, and 255 is the maximum value for a colour, implying one byte per colour value. It is followed by a single byte of whitespace (conventionally a newline) before the start of the binary data.

mandel.c, Part I

```
#include <stdio.h>
#include<complex.h>

#define SIZE 800

int mandel(complex z0){
  int i;
  complex z;

  z=z0;
  for(i=1;i<320;i++){
    z=z*z+z0;
    if ((creal(z)*creal(z))+(cimag(z)*cimag(z))>4.0) break;
  }

  return i;
}
```

mandel.c, Part II

```
int main(){
  double xmin,xmax,ymin,ymax;
  int i,j,rows,columns;
  complex z;
  int row[SIZE];
  unsigned char line[3*SIZE];
  FILE *img;

  img=fopen("mandel.pam","w");
  fprintf(img,"P6\n%d %d 255\n",SIZE,SIZE);
```

mandel.c, Part II

```
xmin=-2; xmax=1;
ymin=-1.5; ymax=1.5;

for(i=0;i<SIZE;i++){
  for(j=0;j<SIZE;j++){
    z=xmin+j*((xmax-xmin)/SIZE)+(ymax-i*((ymax-ymin)/SIZE))*I;
    row[j]=mandel(z);
  }

  (choose colour and place in line[])
    fwrite(line,1,3*SIZE,img);
  }

  return 0;
}
```

Choose colour…

```
for(j=0;j<SIZE;j++){
  if (row[j]<=63){
    line[3*j]=255;
    line[3*j+1]=line[3*j+2]=255-4*row[j];
  }
  else{
    line[3*j]=255;
    line[3*j+1]=row[j]-63;
    line[3*j+2]=0;
  }
  if (row[j]==320) line[3*j]=line[3*j+1]=line[3*j+2]=255;
}
```

Commentary

No comments, and useful parameters such as size and `xmin` / `xmax` etc. set at compile time – no marks for style.

The code assumes $z = x + iy$.

The pam file format scans from top to bottom, hence starting at `ymax` and progressing to `ymin`.

The code calculates a whole row, then converts that row into the relevant colours, then writes the whole row. This is fairly sane, for there is a significant overhead on I/O operations, so doing one per row is much better than one per pixel.

We relied on an implicit `fclose(img);` as the program exits. Again, no marks for style.

An MPI conversion

There are many ways of converting this to MPI. Our first attempt will let all MPI processes do I/O to the output file. We can do this because we know that row `i` will be written at an offset of `3*SIZE*i` plus the length of the header from the start of the file.

As for the length of the header, you do recall that `fprintf()` is a function which returns the number of bytes written?

As MPI processes are entirely separate processes, there is no point trying to pass file pointers between them (or even file descriptors). Each process will have to open the output file separately,

So the first part of the MPI conversion is to add

```
#include <mpi.h>
```

to the headers.

MPI Initialisation

```
int main(){
  double xmin,xmax,ymin,ymax;
  int i,j,rows,columns,rank,nproc;
  complex z;
  int row[SIZE],hdr;
  unsigned char line[3*SIZE];
  FILE *img;

  MPI_Init(NULL,NULL);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

File Initialisation

```
img=fopen("mandel.pam","w");
MPI_Barrier(MPI_COMM_WORLD);

if (rank==0) hdr=fprintf(img,"P6\n%d %d 255\n",SIZE,SIZE);

MPI_Bcast(&hdr,1,MPI_INT,0,MPI_COMM_WORLD);
```

Paranoid and Conservative

`MPI_Barrier` synchronises the processes – no process passed the barrier until all have reached it. This guarantees that all nodes finish opening the file (and truncating it) before any node writes to it.

`MPI_Bcast`, a broadcast, takes data on a single 'root' node and copies it to all nodes. This example is much better than our previous example for a broadcast, for here the data broadcast have been calculated on the root process, and could not readily be calculated by the other processes.

Breaking up the Loop

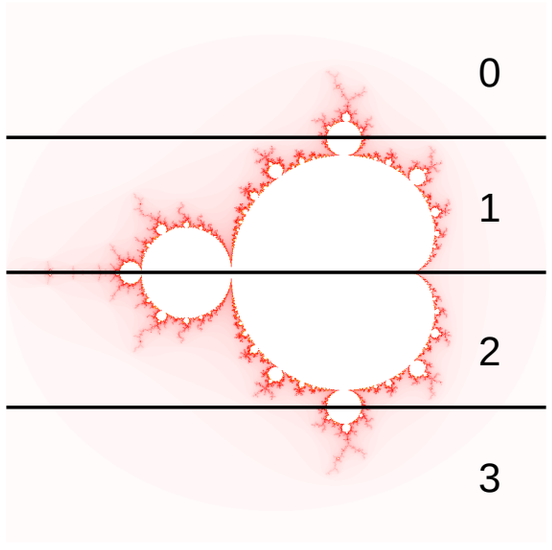
```
for(i=(rank*SIZE)/nproc;i<((rank+1)*SIZE)/nproc;i++){

[...]

  fseek(img,hdr+3*SIZE*i,SEEK_SET);
  fwrite(line,1,3*SIZE,img);
}

MPI_Finalize();
return 0;
```

That's it!

Our 60 line code has had about a dozen changes, and is now a fully parallel code. Parts of the code are completely unchanged by the parallelisation – the function `mandel`, and the two inner loops over `j`, for instance. However, this parallelisation is not yet optimal.

A necessary (but not sufficient) criterion for optimal parallelisation is that work is divided equally amongst the parallel tasks. For the Mandelbrot set, the calculation is much slower in the interior of the set, where the loop in `mandel()` needs to execute 320 times, than in the exterior, where it might exit after a couple of iterations.

With the numbers given, of an y range of -1.5 to 1.5, the reflection symmetry about $y = 0$ gives perfect *load balancing* for two processes, but very poor for four as the region of $0 < y < 0.75$ is much harder to calculate than $0.75 < y < 1.5$. It will be even worse with higher numbers of processes.



Try again

```
for (i=rank;i<SIZE;i+=nproc){
```

Much better for load balancing, as adjacent rows have similar computational difficulty.

However, note that there is a limit to how far this program will scale – it will cannot use more processes than it has rows to calculate.

This is common – there is often a maximum number of processes which can be sensibly used for a given algorithm or program. This often depends on the size of problem – double the requested resolution, and one can (perhaps) scale to twice as many processors.

Alternatives

There are many, many ways of parallelising Mandelbrot set generation. The ones we have seen so far all assume that all tasks can do I/O to the same file. This might not be true. The simplest MPI cluster is two PCs and an ethernet cable, not two PCs, an ethernet cable, and an NFS server set up on one.

We can readily rewrite the above to perform all I/O on the rank zero process by having the other ranks send their output back to rank zero to be written.

```
rank 0: calc row 0, gather, write rows 0-3, calc row 4, gather, write rows 4-7, calc row 8, ...
rank 1: calc row 1, gather,                 calc row 5, gather,                 calc row 9, ...
rank 2: calc row 2, gather,                 calc row 6, gather,                 calc row 10, ...
rank 3: calc row 3, gather,                 calc row 7, gather,                 calc row 12, ...
```

The Conversion

Firstly declare `buff` to be a pointer to an unsigned char, then add immediately after the three MPI initialisation calls

```
if (rank==0){
  buff=malloc(nproc*3*SIZE);
  if(!buff){fprintf(stderr,"Malloc failed\n");exit(1);}
}
```

Now we have a code where not only do different ranks do different things, but they have different memory requirements too.

Now we need to replace the two lines

```
fseek(img,hdr+3*SIZE*i,SEEK_SET);
fwrite(line,1,3*SIZE,img);
```

(One also needs to add `#include <stdlib.h>` due to the `malloc()` and `exit()` calls.)

Conversion, Part II

```
MPI_Gather(line,3*SIZE,MPI_CHAR,buff,3*SIZE,MPI_CHAR,0,
           MPI_COMM_WORLD);
if (rank==0) fwrite(buff,1,3*nproc*SIZE,img);
```

MPI_Gather is a very useful routine for gathering together results. Its arguments are

```
MPI_Gather(void *sendbuff, int sendcount, MPI_Datatype sendtype,
           void *recbuff,  int reccount,  MPI_Datatype rectype,
           int root, MPI_Comm communicator)
```

The result is that, on the root process, `recbuff` holds the concatenation of the `sendbuffs` on all processes.

Note that the `reccount` is per transfer from a process; the total size of `recbuff` must be this multiplied by the number of processes in the communicator.

Generally `reccount=sendcount` and `rectype=sendtype`.

More Details

The three "rec" arguments are ignored expect on the root process, so passing an uninitialised, or null, pointer for the receive buffer on the other processes is fine.

The standard says that the receive buffer is distinct from the send buffer, so making `line` nproc times as big on the root process and writing

```
MPI_Gather(line,3*SIZE,MPI_CHAR,line,3*SIZE,MPI_CHAR,0,
           MPI_COMM_WORLD);
if (rank==0) fwrite(line,1,3*nproc*SIZE,img);
```

is **wrong**.

Though **wrong**, it is the class of **wrong** that the compiler will not spot, and which will work sometimes on some implementations. Problems like this are not restricted to MPI – they must be learnt and avoided.

Because MPI has some Fortran heritage, its rules for arguments are the same as Fortran: any argument which is written to must be distinct from all other arguments. Excluding aliasing in this fashion helps some optimisation. If one wishes to do the gathering in place, the correct answer is

```
MPI_Gather(MPI_IN_PLACE,0,MPI_CHAR,line,3*SIZE,MPI_CHAR,0,
           MPI_COMM_WORLD);
if (rank==0) fwrite(line,1,3*nproc*SIZE,img);
```

Potential Problems

Gathering data to the root process and then writing it out tends to lead to the root process needing more memory than the others. This is fine unless it leads to the root process needing very significantly more memory.

Parallelisation is sometimes used to make code run faster. But almost as often it is used to gain access to more memory. If the reason for the parallelisation was that the data would not all fit on a single node, then using `MPI_Gather` to collect all data onto a single node is unhelpful.

Collectivism

So far all the MPI calls we have seen have been *collective* commands. All processes have issued the same MPI call, all have been involved in doing something for that call, but perhaps not all have ended up with the same result.

In many scenarios, this is the simplest way of using MPI. However, there is a different approach, which involves using the lower-level 'point to point' MPI calls. For the examples we shall consider, for a given transaction just two processes issue the calls, and they issue different calls.

The first example will be to modify the last Mandelbrot so that all the I/O is done on the rank zero process, but it needs no extra memory.

Point to Point

```
if (rank==0){
  fwrite(line,1,3*SIZE,img);
  for(j=1;j<nproc;j++){
    MPI_Recv(line,3*SIZE,MPI_CHAR,j,i+j,MPI_COMM_WORLD,&st);
    fwrite(line,1,3*SIZE,img);
  }
}
else
  MPI_Send(line,3*SIZE,MPI_CHAR,0,i,MPI_COMM_WORLD);
```

If rank is not zero, send line calculated to rank zero.

If rank is zero, write out own line, and then loop over other ranks, receiving their lines and writing them out.

```
rank 0: calc row 0, write r0, rec r1, write r1, rec r2, write r2, rec r3, write r3, calc row 4, ...
rank 1: calc row 1, send to rank 0,                                                 calc row 5, ...
rank 2: calc row 2, send to rank 0,                                                 calc row 6, ...
rank 3: calc row 3, send to rank 0,                                                 calc row 7, ...
```

Only in practice rank 1 will start calculating row 5 as soon as it has finished its send: it won't wait for rank 0 to finish all of its writes and to start on its calculation. It will (probably) pause when it tries to send row 5 to a rank 0 which (probably) is not yet ready to receive it.

The Detail

```
MPI_Send(*buff,count,type,dest,tag,comm);
MPI_Recv(*buff,count,type,source,tag,comm,*status);
```

To send a message, one must specify both the destination and a tag. The tag is an positive integer which one is free to choose. The range of tags guaranteed to be valid is from zero to 32,767. There is no requirement to make tags unique – one could set them all to the same value.

To receive a message, one specifies sender and tag. In the Mandelbrot example, we have chosen to use the row number as the tag, as being suitably unique. One can receive with a tag value of `MPI_ANY_TAG`, or a source value of `MPI ANY SOURCE`.

The status argument is a pointer to an `MPI_Status` structure in C, or an integer array of length `MPI_STATUS_SIZE` in Fortran. It records information about the receive, which, for the moment, we shall ignore.

(For Fortran 2008 the status argument is of `type(mpi_status)`.)

More Detail

The calls above are *blocking* calls. That is, the receive does not return until it has received the requested data, and once the send has returned, one is free to over-write the buffer it was given as the message has been sent.

These calls tend to be the easiest to work with, and bugs usually manifest themselves as the program simply hanging.

Note that 'message sent' and 'message received' are not equivalent statements. The sending process may return from the `MPI_Send` call and continue executing before the receiving process has received the message. The MPI library will have taken a local copy of the message if this occurs.

The messages will be received in the order that the root process executes the `MPI_Recv` calls. The order in which the sends are initiated is irrelevant.

It is not an error to send to oneself, provided there is a matching receive. On the other hand, in this case it would be an error, as the receive cannot come first (else it will block and the send never be reached), and the send is not guaranteed to return before a matching receive is executed (though for small messages it often will). With other variants of send and receive which we will meet later it is possible.

Master Slave Parallelism

So far all processes have performed approximately equal amounts of work. Perhaps one was more equal than the others, in doing the I/O, but we have been close to the ideals of communism.

There is another, politically unpopular, approach to organising workers, and that is the master slave relationship. In this case one task, the master, does no work apart from telling the other tasks, the slaves, what to do.

The code looks very different, in that the code run by the master and slaves is very different, and the whole program becomes

```
if master
  [masterful stuff]
else
  [servile stuff]
endif
```

There are many ways one could perform master-slave parallelism for this task. The method choosen here is that the master will send messages to each slave indicating which row number to work on. When a slave finishes a row, it will send back the resulting data, and await further instructions. A row number of -1 will instruct the slave to exit. The slave will use the tag value to indicate which row it has calculated.

Masterful things

The master first executes a small loop over the slaves (of which there are `nproc-1`) in order to send initial work to them.

In the main loop, the master waits for any incoming message (any tag, any source). It uses the status structure returned with the `MPI_Receive` call to determine the tag and sender, and assumes that the tag indicates the row number. Having written out the data, using `fseek` because the rows might not be returned in order, it then sends more work to the slave until all work has been sent out.

There is a final small loop over all slaves to collect their final results, and this time to tell them to exit.

The syntax for dealing with MPI statuses differs between C and Fortran.

```
int tag,sender;
MPI_Status st;
...
MPI_Recv(...,&st);
tag=st.MPI_TAG;
sender=st.MPI_SOURCE;
```

```
integer :: tag,sender
integer :: st(MPI_STATUS_SIZE)
...
call MPI_Recv(...,st,ierr)
tag=st(MPI_TAG)
sender=st(MPI_SOURCE)
```

More differences

It differs again between the F90 and F2008 Fortran modules.

```
use mpi
integer :: tag,sender
integer :: st(MPI_STATUS_SIZE)
...
call MPI_Recv(...,st,ierr)
tag=st(MPI_TAG)
sender=st(MPI_SOURCE)
```

```
use mpi_f08
integer :: tag,sender
type(mpi_status) :: st
...
call MPI_Recv(...,st)
tag=st%MPI_TAG
sender=st%MPI_SOURCE
```

The F2008 way is nicer, but is it sufficiently nicer to cause code to be rewritten? I believe that most code is currently using the older F90 module, which is why this course uses it.

Note that `MPI_TAG` changes from being a valid array index, therefore presumably an integer, to being a component of a type, and therefore not an integer.

Servile things

The slaves sit in a single infinite loop. This starts by waiting for a message from the master node. If the single integer sent is -1, they exit, else it is assumed to be a row number, and the relevant row is calculated and sent back.

Beginning

```
int main(){
  double xmin,xmax,ymin,ymax;
  int i,j,rows,columns,rank,nproc,nslaves,msg;
  complex z;
  int row[SIZE],hdr,r,s;
  unsigned char line[3*SIZE], *buff;
  FILE *img;
  MPI_Status st;

  MPI_Init(NULL,NULL);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  nslaves=nproc-1;

  xmin=-2; xmax=-1;
  ymin=0; ymax=1;

  if (rank==0){ /* Master */
    img=fopen("mandel.pam","w");
    hdr=fprintf(img,"P6\n%d %d 255\n",SIZE,SIZE);
```

Master I

```
for(i=0;i<nslaves;i++)
  MPI_Send(&i,1,MPI_INT,i+1,0,MPI_COMM_WORLD);

for (i=0;i<SIZE;i++){
   MPI_Recv(line,3*SIZE,MPI_CHAR,MPI_ANY_SOURCE,MPI_ANY_TAG,
            MPI_COMM_WORLD,&st);

   r=st.MPI_TAG;
   s=st.MPI_SOURCE;
   fseek(img,hdr+3*SIZE*r,SEEK_SET);
   fwrite(line,1,3*SIZE,img);

   if ((i+nslaves)<SIZE) msg=i+nslaves;
   else msg=-1;

   MPI_Send(&msg,1,MPI_INT,s,0,MPI_COMM_WORLD);
  }
}
```

Slave I

```
else{  /* Slave */
  for(;;){
    MPI_Recv(&i,1,MPI_INT,0,0,MPI_COMM_WORLD,&st);
    if (i==-1) break;

    for(j=0;j<SIZE;j++){
      z=xmin+j*((xmax-xmin)/SIZE)+(ymax-i*((ymax-ymin)/SIZE))*I;
      row[j]=mandel(z);
    }
```

Slave II and End

```
for(j=0;j<SIZE;j++){
  if (row[j]<=63){
    line[3*j]=255;
    line[3*j+1]=line[3*j+2]=255-4*row[j];
  }
  else{
    line[3*j]=255;
    line[3*j+1]=row[j]-63;
    line[3*j+2]=0;
  }
      if (row[j]==320) line[3*j]=line[3*j+1]=line[3*j+2]=255;
    }
    MPI_Send(line,3*SIZE,MPI_CHAR,0,i,MPI_COMM_WORLD);
  }
 } /* End if master / slave */

 MPI_Finalize();
 return 0;
}
```

Master Slave Advantages

A master slave approach tends to be very good for load balancing. Each slave is given new work as it finishes its current work. The slaves are thus kept optimally busy whether or not the work parcels are the same size, or the slaves equally fast. The slaves do not wait for each other to finish work.

As programmed in this example, the slaves also do no I/O.

The result can be ideal for loose heterogeneous networks of different workstations. In general the master may need less memory than the slaves. (In this example the master makes no use of the array `row`, so we could have made this a pointer, and allocated it on the slaves only.)

In this example, the master does not perform a single floating-point operation either.

Master Slave Disadvantages

The master does no work. The code simply will not run with a single process, and on two processes only one process will be doing computation. The efficiency cannot be more than $(n-1)/n$. However, for moderately large n, this can be reasonably close to one.

But, the master can be a bottle-neck. If there are many slaves, and the parcels of work they are given is small, they can spend all their time waiting to report their results to the master, and waiting for new instructions from the master. Careful thought is needed to get this approach to scale to very large n.

In our case we have the master write out a slave's results before sending new work to the slave. This is silly – we should send new work immediately, and write out the data only after instructing the slave anew.

Tags and Collectives

Point–to–point communications are matched (paired) based on the user-supplied tags.

Collective operations have no tags. They are matched in the order that they are encountered, which therefore must be the same on all ranks, although this matching is independent of any point–to–point communication occuring.

That is to say if one rank executes a barrier, then a scatter, then a reduce, all ranks must execute the precisely matching collectives in that order, perhaps with different random interleaving of point–to–point and enquiry functions.

And the only collective operation which guarantees synchronisation (no process passes it until all have reached it) is an explicit barrier.

Laplace's Equation

A Constant

For the trivially parallelisable task of Mandelbrot set generation, there is no communication between workers (save for reporting back results), and the core of the algorithm, the function `mandel()` is the same in the serial version of the code, and all the parallel versions we have seen. Indeed, it would be quite challenging to parallelise that function in a useful fashion.

For some algorithms, the changes required for parallelisation do go rather deeper into the code.

As an example, we will consider a naïve solver for Laplace's equation.

Laplace's Equation

Laplace's equation is

$$\nabla^2 x = 0$$

with boundary conditions which exclude the trivial solution of $x = 0$. We will consider this equation on a square, with x fixed on two opposite edges of the square (the top and bottom). It turns out that an iterative solution for this is

$$x_{i\,j} = \frac{1}{4}\left(x_{i-1\,j} + x_{i+1\,j} + x_{i\,j-1} + x_{i\,j+1}\right)$$

a formula which one iterates until there is no further change.

(Note that if the right hand side is not zero, then this is called Poisson's equation. Laplace's equation is useful for solving for the electric potential (and hence field) in a vacuum, amongst other things.)

Serial Solution

Again we start with serial code for solving this equation. We define two grids, `g1` and `g2`, and fill `g2` with the above averages from `g1`. This done, we swap pointers so that `g2` becomes the new `g1`, and the old `g1` is then used as `g2`.

We need special case code for the top and bottom rows of the grid, for these are our unchanging boundary conditions. Also for the left and right columns, as these are missing one neighbour. (We could have chosen to use periodic boundary conditions, although this also needs careful coding.)

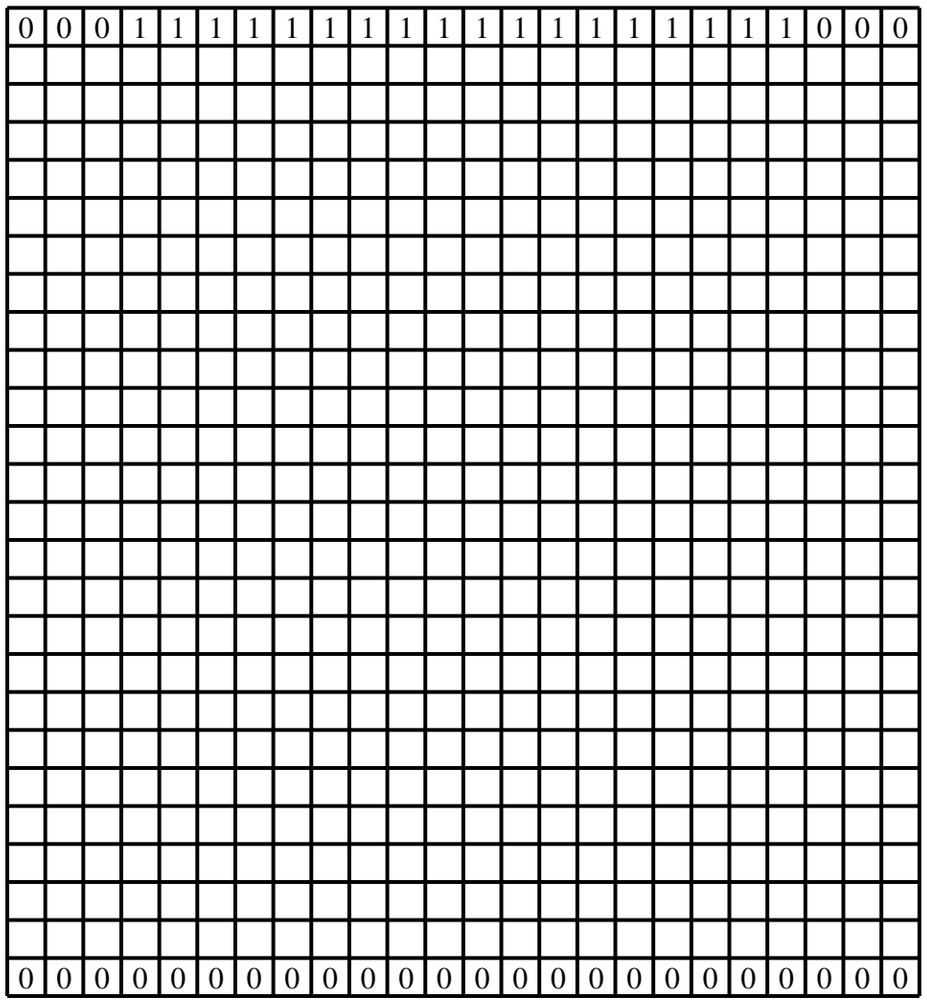
Rather untidily, the grid is fixed to be square by a single parameter. The height is then extended by two cells to add in the fixed boundaries.

For no very good reason, this code uses a fixed number of iterations.

As the top and bottom rows are fixed, there is no reason not to set them initially in both `g1` and `g2`, and then leave them alone. This simplifies the inner loop to being over just the left and right columns and the main body.

The boundary condition chosen here is bottom zero, top one from one eighth width to seven eighths width, else zero.

In Pictures




```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 260 280

10 10 250 { 10 moveto 0 260 rlineto stroke } for

10 10 270 { 10 exch moveto 240 0 rlineto stroke } for

/Times-Roman 8 selectfont

1 1 24 { 10 mul 3 add 12 moveto (0) show } for

1 1 3 { 10 mul 3 add 262 moveto (0) show } for

4 1 21 { 10 mul 3 add 262 moveto (1) show } for

22 1 24 { 10 mul 3 add 262 moveto (0) show } for
```

Here a 24x24 grid, with a top and bottom fixed boundary, making a 24x26 grid.
(Also EPS source for figure for amusement.)

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define SIZE 400

int main(){
  int i,j,n,ht,niter;
  double *g1,*g2,*p;
  FILE *img;

ht=SIZE+2;
niter=40000;

g1=malloc(SIZE*ht*sizeof(double));
g2=malloc(SIZE*ht*sizeof(double));

for(i=0;i<SIZE*ht;i++) g1[i]=0.0;          /* Zero whole grid */
for(i=SIZE/8;i<7*SIZE/8;i++) g1[i]=1.0;    /* Boundary condition */

/* Copy top row to second grid */
for(i=0;i<SIZE;i++) g2[i]=g1[i];
/* Copy bottom row to second grid */
for(i=0;i<SIZE;i++) g2[i+(ht-1)*SIZE]=g1[i+(ht-1)*SIZE];
```

The Main Loop

```
for(n=0;n<niter;n++){

  for(j=1;j<ht-1;j++){
  /* Left */
      g2[j*SIZE]=(g1[j*SIZE+1]+
                          g1[(j-1)*SIZE]+g1[(j+1)*SIZE])/3.0;
  /* Body */
    for(i=1;i<SIZE-1;i++){
      g2[j*SIZE+i]=0.25*(g1[j*SIZE+i-1]+g1[j*SIZE+i+1]+
                   g1[(j-1)*SIZE+i]+g1[(j+1)*SIZE+i]);
    }
  /* Right */
      g2[j*SIZE+SIZE-1]=(g1[j*SIZE+SIZE-2]+
                          g1[(j-1)*SIZE+SIZE-1]+g1[(j+1)*SIZE+SIZE-1])/3.0;
  }

  p=g1;g1=g2;g2=p;
}
```

The End

```
  img=fopen("poisson.pam","w");
  fprintf(img,"P2\n%d %d 255\n",SIZE,SIZE);
  for(i=SIZE;i<(SIZE+1)*SIZE;i++) fprintf(img,"%d\n",(int)(255*g1[i]));
  fclose(img);

  return 0;
}
```

The Above

The above works, and introduces us to a new form of image. This one is greyscale, and has the byte values written in ASCII, not binary. This makes it even more inefficient than the last. . .

It is often called a pgm ‘portable grey map’, whereas the previous coloured example is often called a ppm ‘portable pixmap’.

(There are six flavours of pnm – black and white, greyscale and full colour, and image data in binary or ASCII. You may find these formats documented by `man pbm`, `man pgm` and `man ppm`.

To convert such images to something compressed, and of more reasonable size to store, there are useful commands such as `pnmtopng`.)

Speed

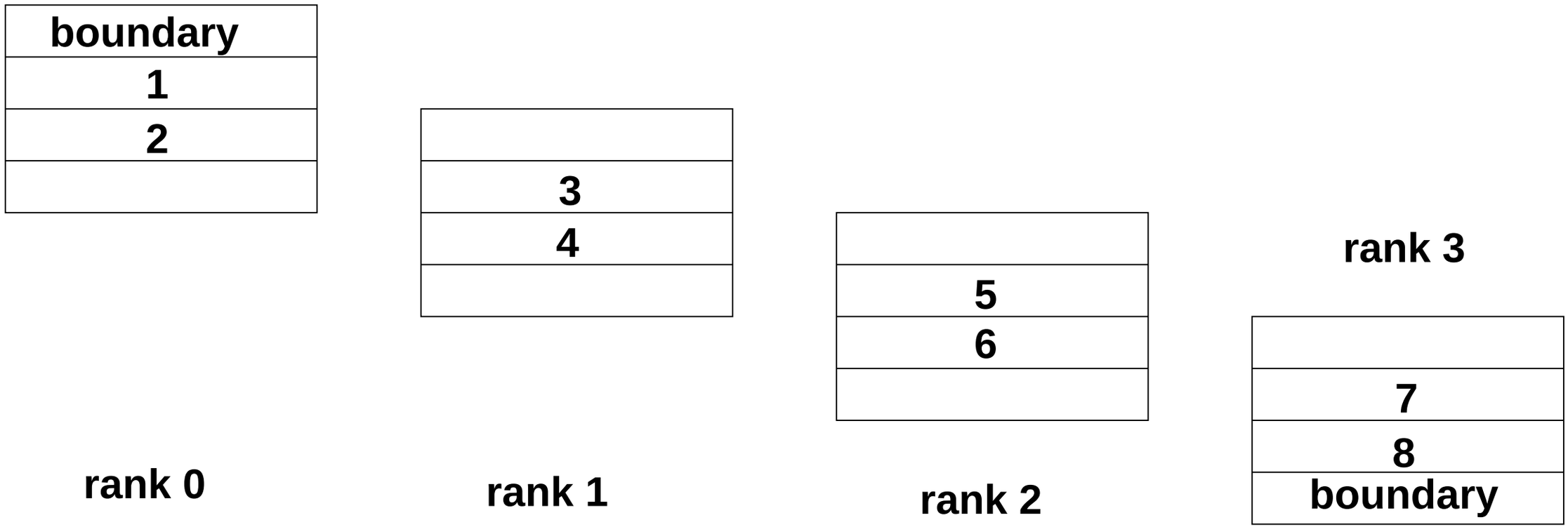
With the parameters given, the 'body' loop is executed 40000×400^2 times. The run time on a 3.3GHz computer was 36.2s, so around 13.5 clock cycles per iteration. Not brilliant for three adds and one multiply. However, there are a lot of integer operations calculating array indices as well.

The above timing was using gcc with no options. Recompiling with '`-O3`' drops the run-time to 9.8s. One must never ignore possible speedups from serial optimisations. We will do well if, through parallelising on a four core computer, we can beat that factor of 3.7 speedup! It is now down to under four clock cycles per iteration of the inner loop, which also sounds more reasonable.

A Parallelisation Scheme

Our first attempt at parallelisation will divide the grid into bands. As each iteration constructs `g2` from `g1`, each point in `g2` is calculated independently of any other point in `g2` – parallelisation heaven!

Just as in the serial code the grid had an extra row at the top and bottom for the fixed boundary conditions, for the parallel version we will repeat this idea for each process. Only now this extra row will be set by a neighbouring process and used to provide continuity within the grid. Considering an 8×8 grid on four processes, we have:



```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

#define SIZE 400

int main(){
  int i,j,n,rank,nproc,ht,niter;
  double *g1,*g2,*p;
  FILE *img;
  MPI_Status st;

MPI_Init(NULL,NULL);
MPI_Comm_size(MPI_COMM_WORLD,&nproc);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);

ht=SIZE/nproc+2;
niter=40000;
g1=malloc(SIZE*ht*sizeof(double));
g2=malloc(SIZE*ht*sizeof(double));

for(i=0;i<SIZE*ht;i++) g1[i]=0.0;      /* Zero our part of of grid */
if (rank==0) for(i=SIZE/8;i<7*SIZE/8;i++) g1[i]=1.0; /* Top bc */
```

```
for(n=0;n<niter;n++){

  /* Top */
  if (rank==0)
    for(i=0;i<SIZE;i++) g2[i]=g1[i];
  else
    MPI_Recv(g1,SIZE,MPI_DOUBLE,
             rank-1,MPI_ANY_TAG,MPI_COMM_WORLD,&st);

  if (rank!=nproc-1)  /* Send our bottom row, which is neighbour's top */
    MPI_Send(g1+(ht-2)*SIZE,SIZE,MPI_DOUBLE,
             rank+1,1,MPI_COMM_WORLD);
```

```
/* Bottom */
if (rank==nproc-1)
  for(i=0;i<SIZE;i++) g2[i+SIZE*(ht-1)]=g1[i+SIZE*(ht-1)];
else
  MPI_Recv(g1+SIZE*(ht-1),SIZE,MPI_DOUBLE,
           rank+1,MPI_ANY_TAG,MPI_COMM_WORLD,&st);

if (rank!=0) /* Send our top row, which is neighbour's bottom row */
    MPI_Send(g1+SIZE,SIZE,MPI_DOUBLE,
           rank-1,1,MPI_COMM_WORLD);
```

```
for(j=1;j<ht-1;j++){
/* Left */
  g2[j*SIZE]=(g1[j*SIZE+1]+
              g1[(j-1)*SIZE]+g1[(j+1)*SIZE])/3.0;
/* Body */
  for(i=1;i<SIZE-1;i++){
    g2[j*SIZE+i]=0.25*(g1[j*SIZE+i-1]+g1[j*SIZE+i+1]+
                 g1[(j-1)*SIZE+i]+g1[(j+1)*SIZE+i]);
  }
/* Right */
  g2[j*SIZE+SIZE-1]=(g1[j*SIZE+SIZE-2]+
                     g1[(j-1)*SIZE+SIZE-1]+g1[(j+1)*SIZE+SIZE-1])/3.0;
  }


  p=g1;g1=g2;g2=p;
}
```

```
if (rank==0){
  img=fopen("poisson.mpi.pam","w");
  fprintf(img,"P2\n%d %d 255\n",SIZE,SIZE);
  for(i=SIZE;i<(ht-1)*SIZE;i++) fprintf(img,"%d\n",(int)(255*g1[i]));
  for(j=1;j<nproc;j++){
    MPI_Recv(g1,ht*SIZE,MPI_DOUBLE,
             j,MPI_ANY_TAG,MPI_COMM_WORLD,&st);
    for(i=SIZE;i<(ht-1)*SIZE;i++) fprintf(img,"%d\n",(int)(255*g1[i]));
  }
  fclose(img);
}
else
  MPI_Send(g1,ht*SIZE,MPI_DOUBLE,0,1,MPI_COMM_WORLD);

MPI_Finalize();

return 0;
}
```

Faster!

Serial	9.5s
-np 1	8.9s
-np 2	5.6s
-np 3	4.2s
-np 4	3.6s

Not quite what I was expecting – the difference in time between the serial code and the MPI code run on a single process I cannot explain – they give identical answers.

That this MPI example does not scale brilliantly well is not surprising. Run-times of around 3s mean that overheads for startup and shutdown become significant, and there are a lot of MPI messages sent – two per process per iteration of the `1 .. niter` loop, with `niter` being 40,000. The final writing of the file must also take time.

Universal Results

The parallel version is faster, as one would hope.

Serial optimisation is important.

Total memory usage in the parallel code is greater than that in the serial code, as some data are replicated. This overhead is generally small, but is usually higher than in the OpenMP case. However, the memory use per node is generally smaller than the OpenMP case once multiple nodes are involved (a trick OpenMP cannot do).

The boundary region around the data computed by each process, which the parallelisation requires to be replicated on multiple nodes, is called either a 'halo region' or a 'ghost region'. Here we have a 2D problem, but the parallel decomposition is only in one dimension. If we had parallelised in both dimensions, then the halo region would surround the region being computed, hence the name.

(For some algorithms, such as when higher derivatives are required, the halo region might be two cells thick, not one.)

The overhead associated with parallelisation does increase as the problem size increases, but it increases less rapidly than the total problem size. For a fixed number of processes, ratio of the parallelisation overhead to the total work decreases as the problem size increases.

Don't Do This

This is a bad way of solving Laplace's Equation for two reasons.

Firstly, it is an iterative method which makes no attempt to check for convergence. This minor matter could be improved.

Secondly, it is a rather poor iterative method. Its use of two grids and nearest-neighbour averages means that, starting from a grid of zeros and a non-zero boundary condition at the top, after the first iteration only the top row contains non-zero elements. After the first two iterations, only the top two rows, etc. Information travels very slowly across the grid, and the expected number of iterations for convergence is of order $\texttt{SIZE}^2$.

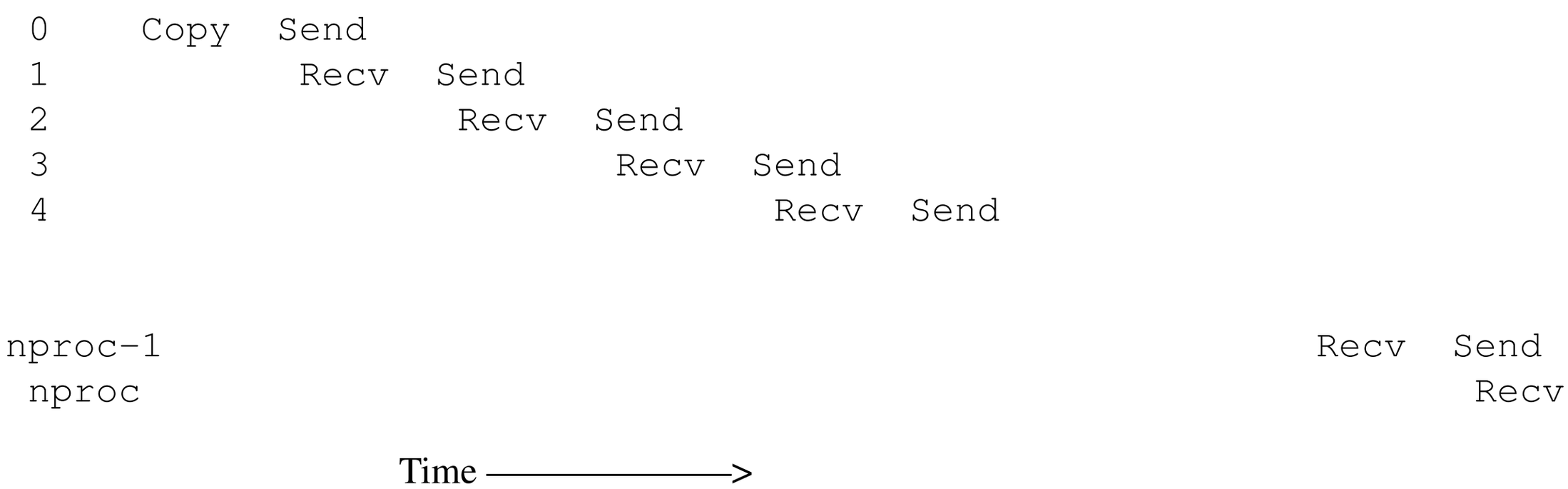
Detailed Worrying

```
if (rank==0)
  for(i=0;i<SIZE;i++) g2[i]=g1[i];
else
  MPI_Recv(g1,SIZE,MPI_DOUBLE,rank-1,MPI_ANY_TAG,MPI_COMM_WORLD,&st);

if (rank!=nproc-1)
  MPI_Send(g1+(ht-2)*SIZE,SIZE,MPI_DOUBLE,rank+1,1,MPI_COMM_WORLD);
```

This code is not very parallel. No process may pass the receive call until it has received the message, which cannot happen until the corresponding process has sent the message.

What Happens?

Rank 0 has no receive, only a send (to rank 1). This means that rank 1 can complete its receive, and move on to its send (to rank 2). Only then can rank 2 move past its receive, etc. To make matters worse, the next section is a similar set of sends and receives, only this time they are solvable in the opposite order – rank `nproc-1` must start, and rank 0 will be last. The communication is taking order `nproc` time, whereas, if perfectly parallelised, the computation will take order `1/nproc` time.



There must be a better way, and there is – at least two.

Other Sends

The simple `MPI_Send` we have used so far guarantees that, when it returns, a copy of the data to be sent has been taken. This may, or may not, mean that the data have been received (probably not for short messages, but yes for large ones). There are two other forms of `MPI_Send`. One, `MPI_Bsend` places the data in a user-supplied buffer, and returns once that is done.

The other, `MPI_Isend`, does not even do that. One is not permitted to alter the contents of the data (to be) sent until one has confirmation that the sending has occured. One does get back a handle one can use to enquire after progress.

Purists frown on `MPI_Bsend` – the extra buffer uses time and space.

First one must define some buffer space. It is an error to start more buffered sends than one has supplied buffer space for, and the default buffer space is zero. Each MPI process can have either zero or one such buffers defined at any time.

```
  void *mpi_buff;
[...]
  mpi_buff=malloc(4*SIZE*sizeof(double));
  MPI_Buffer_attach(mpi_buff,4*SIZE*sizeof(double));
```

```
MPI_Buffer_Attach(void *buff, int len)
```

The amount of buffer space provided must be the total size of all sends expected to be buffered simultaneously, *plus* `MPI_BSEND_OVERHEAD` times the number of sends expected to be buffered simultaneously.

Fortran users will be upset that, although the `buff` argument of `MPI_Buffer_attach` can be of any type, its length needs specifying in bytes, and care needs to be taken to ensure that the buffer passed is not automatically deallocated when the current subroutine exits.

`Bsend` **part 2**

Now we can move the two sends to the start of the `niter` loop.

```
for(n=0;n<niter;n++){

  if (rank!=nproc-1)
    MPI_Bsend(g1+(ht-2)*SIZE,SIZE,MPI_DOUBLE,
              rank+1,1,MPI_COMM_WORLD);
  if (rank!=0)
     MPI_Bsend(g1+SIZE,SIZE,MPI_DOUBLE,
              rank-1,1,MPI_COMM_WORLD);
```

With just four processes, the results are not electrifying – down to 3.5s, the actual improvement being around 0.15s. With larger numbers of processes, such changes become more important.

Note that the arguments for _`Send` and _`Bsend` are identical, and the matching _`Recv` call cares not which is used.

Voiding buffers

For completeness, to discard a buffer Fortran needs to call

```
call MPI_Buffer_detach(buff,len,ierr)
```

The first argument is not used (save with the F2008 interface), the second will return the length of the buffer just removed.

C is more surprising (example from MPI standard):

```
#define BUFFSIZE 10000
int size;
char *buff;
MPI_Buffer_attach( malloc(BUFFSIZE), BUFFSIZE);
/* a buffer of 10000 bytes can now be used by MPI_Bsend */
MPI_Buffer_detach( &buff, &size);
/* Buffer size reduced to zero */
MPI_Buffer_attach( buff, size);
/* Buffer of 10000 bytes available again */
```

BUT the prototype is

```
int MPI_Buffer_detach(void *buffer, int *len)
```

though the first argument is treated as a pointer to a pointer, and that pointer is set to the buffer detached. This is 'to avoid complex type casts' (according to the MPI standard).

Immediate Sends

The original `MPI_Send` code could also be converted to use `MPI_Isend`. Again, no change is needed on the receive side, but this time there is an extra parameter needed on the send calls.

We send from array `g1`, which means we are not permitted to modify those items until we have checked that the send has completed. We are permitted to continue to read those items, so, in the interest of delaying this check as long as possible, the check will be done at the top of the `niter` loop for all iterations save the first. At this point, the swapping of pointers means that what was called `g1` at the last send operation is now called `g2` and will be over-written on this iteration.

We will have made two sends in the previous loop, and will simply wait for both to complete with a single command. It is almost inconceivable that they have not completed, but that is no excuse for not checking.

(Note MPI up to version 2.1 prohibited all accesses to the send buffer until the send had completed. MPI 2.2 and beyond permit reads. We are assuming 2.2 (2009) here.)

```
MPI_Isend(*buff,count,type,dest,tag,comm,*req);
```

Changes

```
  MPI_Request reqs[2];
  MPI_Status stats[2];

[...]

  reqs[0]=reqs[1]=MPI_REQUEST_NULL;
```

The idea of an `MPI_Status` we have met before. This adds an `MPI_Request`. This is a unique tag assigned by the MPI library to an immediate send so that we can reference it and request information about its status. For reasons which will become clear later, we have neatly initialised the requests array to hold the null request.

In Fortran an MPI_Request is simply an integer, whereas in C it is a structure. We have already seen that in Fortran an MPI_Status is an integer array.

Changes (2)

```
for(n=0;n<niter;n++){

  if (n>0) MPI_Waitall(2,reqs,stats);

  if (rank!=nproc-1)
    MPI_Isend(g1+(ht-2)*SIZE,SIZE,MPI_DOUBLE,
              rank+1,1,MPI_COMM_WORLD,&reqs[0]);
  if (rank!=0)
     MPI_Isend(g1+SIZE,SIZE,MPI_DOUBLE,
               rank-1,1,MPI_COMM_WORLD,&reqs[1]);
```

That is it! One extra argument to the send commands, a pointer to a request object, and the `MPI_Waitall` call has the obvious syntax of

```
int MPI_Waitall(int n, MPI_Request r[n], MPI_Status s[n])
```

In Fortran `MPI_Waitall` expects as its request argument an integer array length `n`, and as its status argument an integer array dimensions `(MPI_STATUS_SIZE,n)`.

It Works!

. . . and for this simple example it is not possible to say much about speed. It is faster than `MPI_Send`, and about the same as `MPI_Bsend`.

There is one slight trick above. Processes rank zero and `nproc-1` perform only one send, so should check for only one send. By initialising the requests array to `MPI_REQUEST_NULL`, we ensure that the extra request id these ranks pass is `MPI_REQUEST_NULL`, and a wait operation on a request id of `MPI_REQUEST_NULL` is a null operation. If we had not initialised this array, then two processes would have been waiting on invalid request ids. This would not cause happiness, and would probably have caused a segfault (if one assumes that the MPI library uses the request id as a pointer, or an index into an array, and makes an illegal memory reference as a result of following an uninitialised one).

Synchronous vs Asynchronous

In computing, a synchronous operation happens, to completion, when it is requested. An asynchronous operation is not synchronised with the call which requested the operation – it happens at some undefined time in the future (including, possibly, immediately).

If one performs a synchronous operation, and then performs some other operation which will have the effect of checking whether the previous operation has completed, the answer will always be yes. With an asynchronous operation, the answer could be either yes or no.

Simple I/O generally uses a weaker form of asychronisity, in that the program continues before the data have actually been written to the physical disk, but attempts to read the data just written will always see it as though it had been written, even if it is simply cached.

More on Asynchronous Communication

It is possible to wait for a single send:

```
int MPI_Wait(MPI_Request *r, MPI_Status *s)
```

It is possible to test the completion of a request, without waiting (blocking) if it is incomplete.

```
int MPI_Test(MPI_Request *r, int *flag, MPI_Status *s)
```

On return, `flag` is set to true if the request has completed, and then `s` contains the relevant status, or, if the request has not completed, `flag` is set to false. Fortran users please note that, in Fortran, `flag` is a logical, not an integer.

Yet More on Asynchronous Communication

Non-blocking receives are also provided. It is not meaningful to look at the contents of the receive buffer until a call to `MPI_Test` or `MPI_Wait` has indicated that the request has completed.

The rest you can guess. The name of the call is `MPI_Irecv`, and there is one extra parameter immediately after the communicator, a pointer to an `MPI_Request`.

One should also note that MPI_Request ids are unique at any given time, but may be reused as the program runs. Once `MPI_Wait`, `MPI_Waitall`, or `MPI_Test` (with `flag` returned true) returns, the associated request ids are no longer valid, and those calls will have deallocated any associated internal structures, and reset the request passed to `MPI_REQUEST_NULL`. This prevents code leaking request ids (provided one checks for completion!).

To test for completion without nullifying a request id, one can use `MPI_Request_get_status`, which is otherwise identical to `MPI_Test`. For completeness one should mention

```
int MPI_Request_free(MPI_Request *req)
```

which frees a request without checking its completion status, but cannot be called on `MPI_REQUEST_NULL`.

Why Asynchronous?

It can avoid deadlocks.

```
MPI_Send(parcel,1,MPI_DOUBLE,(rank+1)%nproc,1,MPI_COMM_WORLD);
MPI_Recv(parcel,1,MPI_DOUBLE,(rank-1+nproc)%nproc,1,MPI_COMM_WORLD);
```

That might look like a way of rotating a parcel amongst workers, and it will probably work with small parcels. With a large parcel, everyone will be stuck in a blocking send before anyone posts the corresponding receive.

There are several solutions. Turn the sends into isends. Turn the recvs into irecvs and move them before the sends. And, in this case, reverse the order of the send and receive depending on whether the process's rank is odd or even (works only if `nproc` is even).

Performance? In theory asynchronous MPI may allow a process to continue real work, whilst some communication goes on in the background. This possibility tends to be overstated, but that does not mean it is completely false. There will be more details later.

Not Asynchronous (1)

MPI does provide two convenience functions for the above situation, and the situation of a pair of processes wishing to exchange data but with no ready order for who should go first.

```
        Process 1                          Process 2
   (with targ=process 2)              (with targ=process 1)

MPI_Send(data1,1,MPI_DOUBLE,       MPI_Send(data1,1,MPI_DOUBLE,
         targ,MPI_COMM_WORLD);              targ,MPI_COMM_WORLD);
MPI_Recv(data2,1,MPI_DOUBLE,       MPI_Recv(data2,1,MPI_DOUBLE,
         targ,MPI_COMM_WORLD);              targ,MPI_COMM_WORLD);
```

Is not a good way for two processes to exchange data.

In practice it will probably work with small message sizes (as above), then fail once the message size becomes large. (See later comments on ‘eager’ transfers.) It can introduce a subtle sort of bug for which the code works on small test cases, but fails on larger systems. . .

Not Asynchronous (1)

```
if (rank<targ){
  MPI_Send(data1,1,MPI_DOUBLE,
           targ,MPI_COMM_WORLD);
  MPI_Recv(data2,1,MPI_DOUBLE,
           targ,MPI_COMM_WORLD);
}
else{
  MPI_Recv(data2,1,MPI_DOUBLE,
           targ,MPI_COMM_WORLD);
  MPI_Send(data1,1,MPI_DOUBLE,
           targ,MPI_COMM_WORLD);
}
```

```
if (rank<targ){
  MPI_Send(data1,1,MPI_DOUBLE,
           targ,MPI_COMM_WORLD);
  MPI_Recv(data2,1,MPI_DOUBLE,
           targ,MPI_COMM_WORLD);
}
else{
  MPI_Recv(data2,1,MPI_DOUBLE,
           targ,MPI_COMM_WORLD);
  MPI_Send(data1,1,MPI_DOUBLE,
           targ,MPI_COMM_WORLD);
}
```

This works. The same code on both processors, but each will execute the send and receive in opposite orders. Only it doesn't quite work. What happens if `rank=targ`? Legal in MPI, and still deadlock here. A few more lines are needed to fix this.

SendReceive

```
int MPI_Sendrecv(void *sendbuff, int sendcount, MPI_Datatype sendtype,
                 int dest, int sendtag,
                 void *recvbuff, int recvcount, MPI_Datatype recvtype,
                 int source, int recvtag,
                 MPI_Comm comm, MPI_Status *st);
```

Not pretty with its twelve arguments, but this is guaranteed not to deadlock if one of the two possible send/receive orderings would succeed. One can consider it as running a send and receive simultaneously (perhaps in separate threads) and then continuing when both have returned. It is unlikely that the implimentation really uses threads...

There is a slightly prettier form

```
int MPI_Sendrecv_replace(void *buff, int count, MPI_Datatype datatype,
                         int dest, int sendtag, int source, int recvtag,
                         MPI_Comm comm, MPI_Status *st);
```

In this form the single buffer is used for both the send and receive. If performed between a pair of processes, this simply swaps the data in `buff`. It can also be used for a cyclic permutation, and it can even match standard sends and receives. It is not considered to be a collective operation.

Not Asynchronous (2)

`MPI_Ssend`. In every way like `MPI_Send`, except that it is guaranteed not to return until the matching receive has been reached, even for small messages. It thus provides a degree of synchronisation.

The extra ‘s’ stands for ‘synchronous’.

Decomposition in General

In this example we took a 2D grid and distributed it across ranks in a 1D fashion, in that only one dimension was split. Sometimes this is a reasonable approach.

In general one wishes to minimise the amount of halo region replicated on each rank to the amount of data being calculated on each rank. Haloes are expensive in time, as they need to be exchanged, and expensive in memory, as they are things being replicated.

In our example, if we had as many processes as rows, each rank would have one row to work on, but still two rows (top and bottom boundary conditions) in its halo. Thus the total memory requirement for the array summed over all ranks would be three times that for the serial code, and the time taken to exchange the halos would surely dominate over the computational time.

In general one obtains the best scaling by distributing across multiple dimensions, with each rank working on a near square (cube, n-cube) part of the full grid. This does lead to more, and more complex, data exchanges though – a square has four neighbours, not two. So there are certainly situations in which not distributing in one (or more) dimensions is preferable.

(NB some disciplines prefer to use the term “ghost” rather than “halo.”)

Summary

MPI_Isend: immediate return, cannot change contents of send buffer until request completes (check with `MPI_Wait` or `MPI_Test`).

MPI_Bsend: immediate return after copying data into user-supplied buffer, which must be big enough for this and other outstanding `MPI_Bsend`s.

MPI_Ssend: will not return until it is safe to change contents of send buffer and receiver has reached corresponding receive call.

MPI_Send: may act like `MPI_Ssend`, or like `MPI_Bsend` using an internal buffer. Behaviour may change with message size.

For all except `MPI_Isend`, it is safe to modify the send buffer as soon as the call returns.

`MPI_Isend` has a single extra argument, a pointer to the request id.

Any may be paired with either of the two receives we have met, `MPI_Recv` and `MPI_Irecv`.

A user-supplied buffer must be provided before any call to `MPI_Bsend`.

The combined `MPI_Sendrecv` can be useful in avoiding deadlocks, and can be paired with itself, or with the above sends and receives.

Miscellaneous

Unknown Message Sizes

MPI allows for the case in which the receiver does not know how big a message received in a point–to-point communication is. This could be the case in a master–slave situation if the orders are something like 'find all primes in this range'.

One solution MPI provides is to call `MPI_Probe` in a fashion like `MPI_Recv`, only it merely fills in the status, and does not actually receive any data. Calling `MPI_Get_count` on the status returned gives the number of items, a suitable buffer can then be allocated, and `MPI_Recv` called as usual.

Most of you will never think of a use for this, yet for some of you it might be very useful.

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *st)
```

(Another solution is to call `MPI_Recv` with a buffer guaranteed to be large enough. A 'short' send will match without error, and the actual message size can be determined by calling `MPI_Get_count` on the receive's status.)

MPI_Probe

```
int count;
MPI_Status st;

[...]

MPI_Probe(source,tag,MPI_COMM_WORLD,&st);
MPI_Get_count(&st,MPI_DOUBLE,&count);
buff=malloc(count*sizeof(double));
MPI_Recv(buff,count,MPI_DOUBLE,source,tag,
         MPI_COMM_WORLD,&st);
```

Here the MPI_Status variable is re-used. This is not necessary.

Note that the MPI datatype must be known by the receiver, and given on the `MPI_Get_count` call.

MPI_Iprobe

There is also a non-blocking variant of `MPI_Probe` called `MPI_Iprobe`. Unlike the relationship between `MPI_Recv` and `MPI_Irecv`, `MPI_Iprobe` has an extra argument over the blocking form.

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *fl, MPI_Status *st)
```

The flag returns false if there is no matching message waiting (in which case no useful data will be returned in `st`), and true if there is, with `st` filled appropriately. The Fortran version is:

```
logical :: flag
integer :: source, tag, comm, ierr, st(MPI_STATUS_SIZE)
call MPI_Iprobe(source, tag, comm, flag, st, ierr)
```

Mistakes

There are many ways to go wrong with MPI. The first few lectures have carefully avoided any major errors (I hope!), but now we shall deliberately stray into dangerous territory.

The first source of errors we will consider is that of unmatched sends and receives using point-to-point communication. This collection should convince you that using the collective calls is often rather safer!

Common Code

These examples all assume a common start of

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(){
  int rank,nproc;
  void *ptr;
  MPI_Status st;

  MPI_Init(NULL,NULL);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

Deadlock

```
#define SIZE 10240

  ptr=malloc(SIZE);

  if (rank==0){
    MPI_Send(ptr,SIZE,MPI_CHAR,1,1,MPI_COMM_WORLD);
    MPI_Recv(ptr,SIZE,MPI_CHAR,1,1,MPI_COMM_WORLD,&st);
  }
  if (rank==1){
    MPI_Send(ptr,SIZE,MPI_CHAR,0,1,MPI_COMM_WORLD);
    MPI_Recv(ptr,SIZE,MPI_CHAR,0,1,MPI_COMM_WORLD,&st);
  }

  printf("Rank %d finished\n",rank);
  MPI_Finalize();
  return 0;
}
```

Deadlock

This is the awkward case that will (probably) work for small values of `SIZE`, but not for large values. The unbuffered sends cannot complete until their data are copied elsewhere, for as soon as the calls complete the program is free to destroy the buffers passed. However, no receive has been posted for the sends.

There are many solutions to this problem: asynchronous sends, use of `MPI_Sendrecv`, and reversing the order of one send/receive pair.

No compiler will tell you that you have just made this mistake.

Utterly Unmatched

```
  ptr=malloc(1);
  if (rank==0)
    MPI_Send(ptr,1,MPI_CHAR,1,1,MPI_COMM_WORLD);
  if (rank==1)
    MPI_Recv(ptr,1,MPI_CHAR,0,0,MPI_COMM_WORLD,&st);

  printf("Rank %d finished\n",rank);
  MPI_Finalize();
  return 0;
}
```

Rank zero sends a message with tag one to rank one, whilst rank one waits to receive a message with tag zero from rank zero.

This is not going to work. What actually happens on the machine this was run on is that rank zero prints that it has finished, and, although it does not exit, it stops using any CPU time (and thus becomes invisible to those who think that '`top`' lists all their processes), whereas rank one cannot complete the receive, and sits burning CPU time.

Idle Wait vs Spin Wait

There are always two ways of waiting for I/O, including incoming MPI messages. One is to sleep, and let the kernel wake one up when there is I/O to process. The other is to sit in a tight loop continuously proactively checking. The former is called idle wait, the latter spin wait.

Spin wait offers shorter response times when the I/O arrives. Shorter by something of the order of μs. Idle wait allows other processes to make use of the CPU.

For most I/O, idle wait is the correct answer. If every program used spin wait to respond to events like keypresses or mouse activity, computers would be very slow (and very power-hungry). MPI often does use spin-wait in order to reduce latencies, and on the assumption that the core would otherwise be idle, so it is not taking time from anything useful.

This means that an MPI program making no progress because it is waiting for I/O which will never arrive can use 100% of the CPU and thus appear to be active and progressing.

DIY Collectivism

Some people are remarkably keen to use combinations of individual sends and receives in order to emulate the function of a collective. In almost all cases, this is silly, even though one could emulate all collective operations using just send and receive.

A sensible MPI library will do most collective operations in $\log N$ time by constructing trees: for a broadcast on the first timestep rank 0 sends to rank 1, on the second ranks 0 and 1 send to 2 and 3, on the third ranks 0 to 3 send to 4 to 7, etc. Similar tricks work in reverse for reduction.

On some machines with dedicated MPI networks barriers are special, and can be done in something closer to constant time than $\log N$. In general one should rely on the MPI library to know whether it is best to use a base 2 or base 4 tree for the system you are using.

If your computer is made up of shared-memory nodes joined by a slowish interconnect, one would expect the collective calls to minimise the use of the slow interconnect.

The Ultimate Collective?

One collective operation we have so far overlooked. It is the biggest (in some sense), and one which, if you need it, it is much better to use than to attempt to emulate. It deals with the not uncommon case of a matrix being distributed across the processes by columns, and then needing to be redistributed by row, or vice versa. It has no root, and, in code, looks like

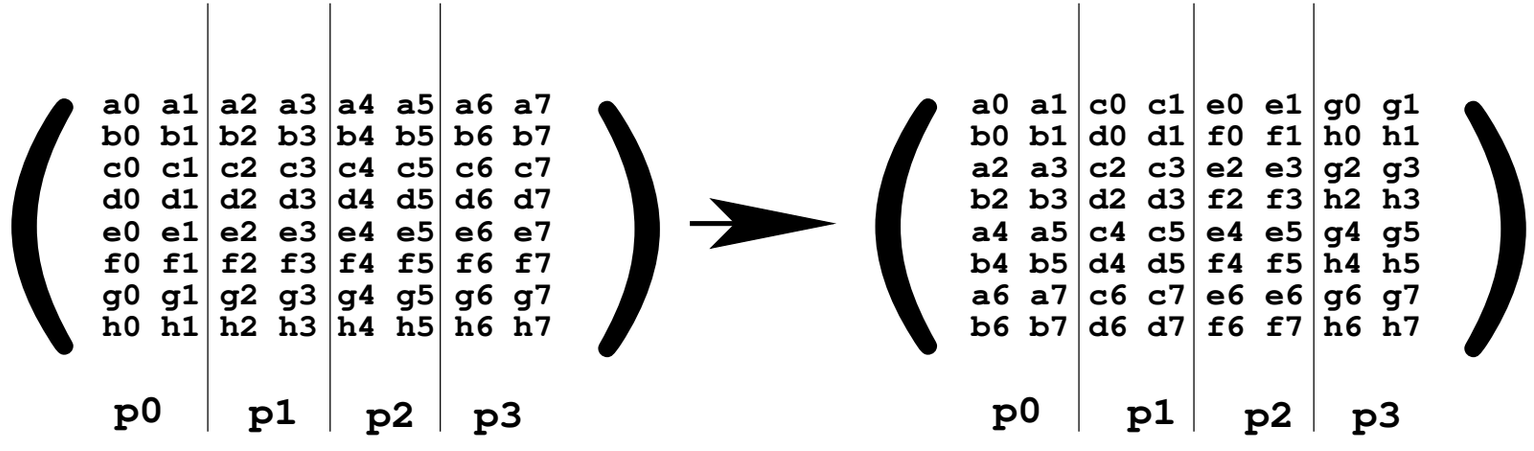
```
for(i=0;i<nproc;i++){
  MPI_Send(sendbuf+i*sendcount,sendcount,sendtype,i,...
  MPI_Recv(recvbuf+i*recvcount,recvcount,recvtype,i,...
}
```

with magic to avoid deadlock.

A simple `MPI_Alltoall` on `n` processes is rather like a transpose of an `n`×`n` matrix distributed across those processes.

```
MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype,
             void *recvbuf, int recvcount, MPI_Datatype recvtype,
             MPI_Comm comm);
```

`Alltoall` in pictures



Assume an 8×8 matrix is distributed across 4 processes so that rank zero has the first two columns, rank 1 the second two columns, etc. Further assume that each rank stores its columns in memory interleaved, that is, rank 0 stores `a0 a1 b0 b1 c0 c1 ...`

After a call to

```
MPI_Alltoall(m1, 4, MPI_DOUBLE, m2, 4, MPI_DOUBLE, MPI_COMM_WORLD);
```

if `m1` was the matrix on the left spread between the four processes, then `m2` will be set to the matrix on the right. This is not quite a transpose, there is still the local operation of transposing each 2×2 block, but the hard work has been done. Note that the '4' in the MPI call is the number of elements in one pairwise communication. (E.g. p0 sends (c0 c1 d0 d1) to p1.)

Final Comments on Alltoall

The ‘transpose’ can be done in place – simply replace `m2` above by `MPI_IN_PLACE`.

Matrix transposes are not simply part of traditional linear algebra, but they can also be a key component of 2D FFTs. A 2D FFT is a 1D FFT of each column, followed by a 1D FFT of each row. If one is storing a large 2D matrix columnwise spread across all ranks, that is a bunch of 1D FFTs on each rank to do the columns, a transpose via `MPI_Alltoall`, then another bunch of 1D FFTs on each rank to do the columns of the transposed matrix, which are the rows of the original matrix.

This is a key part of the parallelisation of CASTEP.

`MPI_Alltoall` is a bit inflexible if one’s matrix size does not divide exactly by the number of processes used. So there is also an `MPI_Alltoallv` which allows the send counts and displacements to vary. The details are beyond the scope of this course.

Wrong Buffer Sizes

```
if (rank==0)
  MPI_Send(ptr,4,MPI_CHAR,1,1,MPI_COMM_WORLD);
if (rank==1)
  MPI_Recv(ptr,1,MPI_CHAR,0,1,MPI_COMM_WORLD,&st);
```

A perfectly matched send and receive, except for the rather important detail that the send is of four chars, and the receive of just one.

This is an error, and one which an MPI library ought to detect. The opposite problem, a send having fewer data items than the receive, is *not* an error, and modifies just the first elements of the receive buffer.

Short Receives

As the standard permits the data sent to be shorter than the data expected, it provides a mechanism for detecting that this has occured.

```
if (rank==0)
  MPI_Send(ptr,1,MPI_CHAR,1,1,MPI_COMM_WORLD);
if (rank==1){
  MPI_Recv(ptr,4,MPI_CHAR,0,1,MPI_COMM_WORLD,&st);
  MPI_Get_count(&st,MPI_CHAR,&count);
  printf("Received %d chars\n",count);
}
```

This can sometimes be useful, particularly with master-slave programming.

Master: find me the first fifty primes between 200 and 300.
Slave: here they are, and there are not fifty. . .

However, it can also indicate a bug, but, because it is legitimate MPI, not the sort of bug which will be detected by the MPI library.

Wrong buffer types

```
if (rank==0)
  MPI_Send(ptr,1,MPI_DOUBLE,1,1,MPI_COMM_WORLD);
if (rank==1)
  MPI_Recv(ptr,8,MPI_CHAR,0,1,MPI_COMM_WORLD,&st);
```

This is also an error. It is never correct to mix different types in MPI. If one must use a 'void' type, then one should specify `MPI_BYTE` on both the send and receive.

The version of MPI I used to test the above failed to detect any error.

Mixing types is a serious error, for the MPI standard supports running MPI on a cluster so heterogeneous that different nodes use different binary representations for the same type. Though you are very unlikely to see such a machine, if some nodes use EBDIC for characters, and IBM's floating point format for reals, and others use ASCII and IEEE-748, then sending "Hello" or "42.0" will still result in "Hello" or "42.0" being received, despite the difference in bit representation. No conversion occurs with the `MPI_BYTE` type. An MPI library designed for a homogeneous computer is probably incapable of converting.

Performance

One cannot sum up the performance of an MPI system in a single number. One might be interested in the performance of large transfers, or small, or collectives, or synchronised sends and receives.

But it is still worth trying to get some basic measures, and the simplest, as befits an introductory course, is a test known as ‘pingpong.’ This simply sends messages of varying size between two processes, and times the result. There are various ways of writing the benchmark, and here we offer the following.

pingpong.c

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define MAXSIZE 1073741824

int main(){
  double *a,start,end;
  int i,size,rep,rank,nproc;

a=malloc(MAXSIZE);
if (!a) exit(1);

MPI_Init(NULL,NULL);
MPI_Comm_size(MPI_COMM_WORLD,&nproc);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

```
for(size=1;size<=MAXSIZE/sizeof(double);size*=2){
  rep=1000000/size+1;
  start=MPI_Wtime();
  for(i=0;i<rep;i++){
    if (rank==0){
      MPI_Send(a,size,MPI_DOUBLE,1,0,MPI_COMM_WORLD);
      MPI_Recv(a,size,MPI_DOUBLE,1,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }
    else if (rank==1){
      MPI_Recv(a,size,MPI_DOUBLE,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
      MPI_Send(a,size,MPI_DOUBLE,0,1,MPI_COMM_WORLD);
    }
    }
    end=MPI_Wtime();
    if (rank==0) printf("%10d bytes in %g sec at %g MB/s\n",
                        size*sizeof(double),(end-start)/(2*rep),
                        (2*rep*(double)size*sizeof(double))/(1e6*(end-start)));
  }
  MPI_Finalize();
  return 0;
}
```

Code Notes

There are some points to note about the above code.

It carefully defines MAXSIZE to be 2^{30} to avoid any problems with overflowing 32 bit integers. Naughty.

It fails to check whether `nproc` is at least two, though it will fail if it is one. Naughty.

It uses `MPI_STATUS_IGNORE` for the status returned by the recieve. Naughty. But at least here there is an excuse – filling in this structure will take time, so one could argue that it is better, for benchmarking purposes, not to do so.

MPI provides a convenient function for measuring Wallclock TIME. Fortran users may be particularly pleased, but even C users may find it more convenient and/or precise than many alternatives. It returns seconds in a double, and its resolution is given by the function `MPI_Wtick`. In Fortran these are also functions returning double precision.

The origin of time is not defined, to the extent that it is not guaranteed that all processes use the same origin (though they might).

Results

```
         8 bytes in 2.05394e-07 sec at 38.9496 MB/s
        16 bytes in 1.8471e-07 sec at 86.6224 MB/s
        32 bytes in 1.88001e-07 sec at 170.212 MB/s
        64 bytes in 1.90187e-07 sec at 336.511 MB/s
[...]
   1048576 bytes in 9.58145e-05 sec at 10943.8 MB/s
   2097152 bytes in 0.000184387 sec at 11373.6 MB/s
[...]
 536870912 bytes in 0.0753155 sec at 7128.29 MB/s
1073741824 bytes in 0.150609 sec at 7129.36 MB/s
```

A latency of 0.2μs, and a bandwidth of 7.1GB/s, or 11.3GB/s whilst within the CPU's last level of cache. This was run on an SMP machine (a 3.3GHz Ivy Bridge). The same using icc rather than gcc was a few percent slower.

More Results

```
         8 bytes in 3.35425e-05 sec at 0.238504 MB/s
        16 bytes in 3.42444e-05 sec at 0.467229 MB/s
        32 bytes in 3.46164e-05 sec at 0.924418 MB/s
[...]
 268435456 bytes in 2.46976 sec at 108.689 MB/s
 536870912 bytes in 4.56301 sec at 117.657 MB/s
1073741824 bytes in 9.12524 sec at 117.667 MB/s
```

A latency of 34μs and a bandwidth of 117MB/s. That is two computers connected via 1GBit/s ethernet, and those are the sort of figures one expects for MPI over TCP in such an arrangement.

Detail

There is a lot of structure in the pingpong results which causes deviations from the simple idea of time being latency plus packet size over bandwidth. Some are caused by hardware. In the SMP case we have mentioned the CPU's last level cache. Some are caused by software, as there are often 'fast' paths for small messages involving including all the data in the initial 'packet' of the message, whereas large messages cannot have all their data sent until there is confirmation of a receive buffer big enough to take them.

Ethernet switches have a subtle hardware feature. A switch will always refrain from starting to send on a packet until it has been completely received (and the ethernet checksum verified – if that fails the packet is dropped as invalid). For transfers of under 1KB, which may be a single ethernet packet, the effective bandwidth is halved as the packet gets sent to the switch at 1GBit/s, and only when wholly received is it sent on at 1GBit/s. For larger transfers ethernet packets get sent to the switch at 1GBit/s whilst simultaneously the switch is sending on previous packets at 1GBit/s.

(Ethernet packets have sizes between 64 bytes and 1518 bytes. This excludes overheads of TCP and MPI, so the amount of useful data in a packet is slightly less. Most modern network switches and interfaces can cope with larger packets, generally up to 9,000 bytes. Use of such 'jumbo' frames can improve throughput for large data transfers, but slow down small transfers. Historically jumbos reduced CPU overheads significantly. Today most of this work is done on the network card itself, so there is less benefit.)

Ethernet packets contain six byte source and destination addresses, four byte checksums, and two byte length fields. This immediately reduces the maximum payload to 1500 bytes.

Progress

Much is said about overlapping communication and computation in MPI codes. This assumes that separate hardware is responsible for communication, and is called ‘strong progress.’

In practice, most MPI implementations rely on the CPU for both communications and calculations. Worse, MPI transfers can be progressed only whilst MPI calls are being made. This results in ‘weak progress.’

Process 0	Process 1
Large, non-blocking send	Matching blocking receive
Lots of computation	[no data transferred]
Another MPI call	[more data transferred]

In Practice

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <time.h>
#define SIZE 4096

int main(int argc, char *argv[]){
  int rank, *a;
  MPI_Request req;
  time_t t1;

  a=malloc(SIZE);
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  printf("Process rank %d\n",rank);

  MPI_Barrier(MPI_COMM_WORLD);
  t1=time(NULL);
  if (rank==0){
    MPI_Isend(a,SIZE/sizeof(int),MPI_INT,1,123,MPI_COMM_WORLD,&req);
    sleep(10);
  }
  else if (rank==1){
    MPI_Recv(a,SIZE/sizeof(int),MPI_INT,0,123,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
  }
  printf("Rank %d reaching barrier at time %d\n",rank,(int)(time(NULL)-t1));
  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize(); return 0;
}
```

Progress Results

SIZE=2048

```
Process 0 out of 2
Process 1 out of 2
Process 1 reaching barrier at time 0
Process 0 reaching barrier at time 10
```

SIZE=4096

```
Process 0 out of 2
Process 1 out of 2
Process 1 reaching barrier at time 10
Process 0 reaching barrier at time 10
```

The behaviour of code like this will vary enormously depending on the details of one's MPI implementation. If one has fewer cores than MPI processes, so the scheduler might not be able to run sending and receiving processes simultaneously, other major slow-downs might occur.

In some cases, a slower interconnect as measured by pingpong is preferable if it has strong progress, and the alternative does not. Some versions of OpenMPI support the option of strong progress, at the cost of extra latency, by using an additional 'progress thread,' but in general this was not found to be beneficial.

MPI does not yet have a call which means 'do nothing, but progress anything which might be outstanding'. However, almost any MPI call will result in a check being made to see if any other communication can or should be progressed.

Eagerness

Small messages are typically sent using an ‘eager’ transfer mode. This means that the transfer to the destination process, though not to the final destination buffer, can happen before the destination process posts a receive. The main purpose of this is to reduce latency. Without it, a transfer looks like:

Sender: Dear receiver, I have a message for you of length x bytes.
Receiver: Dear sender. Jolly good, please send here.
Sender: Dear receiver, Please find enclosed. . .

With it the process is simply:

Sender: Dear receiver, Please find enclosed a short missive. I know it will fit on your doormat until you have time to pick it up.
Receiver: Dear sender, doormat now clear again.

The penalty for eager communication is the size of doormat needed. Given that the sender will know how many eager messages it has sent, but not how many anyone else has sent to the receiver, the size of doormat is
(eager message size) $\times$ nproc

Note that the sum of the sizes of doormats for the whole jobs scales as the square of the number of processes.

(Messages considerably larger than the eager threshold may be sent as multiple fragments.)

(For extra (or any) marks in CompSci, write ‘buffer’, not ‘doormat.’)

Performance Tips

Try to avoid sending lots of small messages, and consider bundling them up instead. Even sending 128 bytes is generally quicker than sending 8 bytes twice, and sending 16 bytes is much quicker than sending 8 bytes twice.

Do use collectives where possible.

Perhaps worry about the number of processes to use. Don't run more than there are cores, but, in some cases, using just half the cores in each socket is preferable (less contention for caches and the memory controller). If the code is limited by memory bandwidth, it might like this approach.

In other cases leaving one core free per node can be advisable (for any kernel I/O threads). If one's MPI library performs strong progress in software, using separate thread(s), again leaving some cores free for those threads might be advisable.

If one's MPI library does not offer strong progress, attempts to overlap calculation and communication by using non-blocking calls are mostly doomed, and do make the code much more complicated.

I/O: Availability

MPI does not guarantee that all nodes can perform I/O (though we have assumed this). According to the standard, one should call

```
MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val,
                  int *flag)
```

with `comm_keyval` set to `MPI_IO`. The value returned in `attribute_val` could be:

`MPI_ANY_SOURCE` if all ranks can do I/O.
One's own rank if one's own rank can do I/O (but not all).
Another rank if some rank can do I/O.
`MPI_PROC_NULL` if no rank can do I/O.

If more than one, but not all, ranks can perform I/O, different ranks might get different results from this function.

Many, many codes do not check this function, and assume all ranks can perform I/O. In many cases, this is quite reasonable.

The natural assumption that rank zero can perform I/O can be checked by calling this function on rank zero, and checking for a return value of either `MPI_ANY_SOURCE` or zero.

Those wondering what to do if `MPI_PROC_NULL` is returned should note that this refers to the ability to do standard language I/O. At that point one hopes that one's MPI system supports MPI-IO.

I/O: Availability continued

Note that we can chose `MPI_COMM_SELF` not `MPI_COMM_WORLD` as the communicator. The communicator `MPI_COMM_SELF` contains just the calling process, so we quickly answer the question 'can this process do I/O?'

The `flag` argument will be returned set to false if the attribute exists but is unset. The call will return an error if the attribute does not exist. Arguably in this case there is little reason to check `ierr` or `flag` for `MPI_IO` really should exist and be set.

The mechanism for calling `MPI_Comm_get_attr` from Fortran is:

```
logical :: flag
integer(kind=MPI_ADDRESS_KIND) :: result
integer :: ierr

call MPI_Comm_get_attr(MPI_COMM_SELF, MPI_IO, result, flag, ierr)

if ((result.eq.0).or.(result.eq.MPI_ANY_SOURCE)) then
   [Our rank will be 0 in MPI_COMM_SELF, and we can do I/O!]
endif
```

Note the use of `MPI_ADDRESS_KIND` for the type of integer which this call will return. As its name suggests, this is an integer big enough to hold an address, so is probably 64 bits.

More Advanced Features

Communicators

So far every example of a communication operation has had an argument of a communicator which we have set to `MPI_COMM_WORLD`. There is further flexibility here, which in some cases can be useful.

Communicators tend to be particularly useful when one has multiple levels of parallelism: the program has been split into a number of subtasks, each of which can be further parallelised.

Plain Wave DFT Codes

```
do over k-points
   do over bands
      do over g vector columns
         Some basic potential operations
         Some 3D FFTs involving exchanges with all other columns
      enddo
      find gradient direction ensuring bands are kept orthogonal
   enddo
enddo
Build density from all k-points
```

There is very little communication between k-points, so it is an obvious loop to parallelise. But typically there are 1 to 20 kpoints, and fewer the larger the system.

There is a little more communication between bands, and rather more, typically 20 to 1,000, and more in larger systems. But the product of the number of bands and k-points might still be less than the number of processors available.

Parallelising the g-vector column loop is the last resort, but there could be 1,000 to 10,000 of these (or more), so one is unlikely to run out.

The Need

The part of the code which is working on the g-vector columns probably wishes to perform collective operations for the 3D FFT which it may wish to perform independently of any collectives occuring for other bands and k-points.

MPI allows for this.

One can create user-defined communicators. Each MPI process must be in `MPI_COMM_WORLD`, but may be in any number of other communicators. Not all members of `MPI_COMM_WORLD` need be in the same number of other communicators. Each new communicator must have a rank zero process, which need not be the rank-zero process in any other communicator.

If you wish to remain sane, you will construct communicators which nest. The MPI standard does not require sanity.

A need might also arise in a master-slave model with a very large number of slaves. The poor master could be overwhelmed, so one can create a hierarchy of masters: one global master, one hundred sub-masters each in its own communicator with a thousand slaves might make more sense than one master with 100,100 slaves. One could also create a communicator containing the global master and the hundred submasters, thus ignoring advice about sanity and communicators nesting.

Splitting a Communicator

```
MPI_Comm_split(MPI_Comm comm, int colour, int key, MPI_Comm *newcomm)
```

(In Fortran both `MPI_Comm` and `MPI_Comm*` are integers.)

The `comm` argument is the current communicator from which new communicators are being made. All ranks in this communicator must make this call.

The `colour` argument is a non-negative integer specifying which new communicator the given rank should join – all ranks calling with the same colour join the same new communicator. It may be set to `MPI_UNDEFINED` to mean join none.

The `key` argument is used to determine the rank of the process in the new communicator.

And `newcomm` is the new communicator and is filled in by this call.

(The value of the communicator handle / structure / integer does not necessarily make sense globally. Two non-overlapping communicators may have the same integer communicator value in Fortran, for instance. The combination of global rank and communicator 'value' uniquely identifies a communicator.)

An Example

Old rank	`colour`	`key`
0	42	0
1	12	5
2	`MPI_UNDEFINED`	0
3	12	1
4	42	0

This creates two new communicators. One contains old ranks 0 and 4, and one old ranks 1 and 3. Rank 2 chose not to participate beyond the requirement that it makes the collective call.

In the communicator formed by old ranks 1 and 3, old rank 3 has rank 0, for its key sorts lowest. In the communicator formed by old ranks 0 and 4, old rank 0 has rank 0, for keys sorted equally, at which point the ordering of the old rank is used to break ties.

On rank 2 the value of the communicator returned will be `MPI_COMM_NULL`. This value must never be used to specify a communicator in an MPI call – to do so is an error.

More on Communicators

An MPI process may participate in communications (collective or point-to-point) in any communicator of which it is a member.

Its rank in each of the communicators of which it is a member will generally be different.

Communicators can be removed with

```
MPI_Comm_free(MPI_Comm *comm)
```

which will set `*comm` to `MPI_COMM_NULL`. This should be called by all ranks within the communicator. After any rank calls it, no further communication may be initiated, although unfinished communication (to other ranks) should complete normally. It is an error to attempt to free `MPI_COMM_WORLD` or `MPI_COMM_NULL`.

If one wishes to split a communicator into two equal pieces, setting the colour on each rank as the split is done to `2*rank/nproc` or `mod(rank,2)` (for C programmers, `rank%2`)
should both work. They result in a different distribution of the new communicators amongst the old ranks, which may result in different performance, depending on the topology of the underlying hardware.

Yet More on Communicators

If one wishes to use user-defined MPI communicators, then `MPI_Comm_split` as above has all the functionality one needs.

One can also use it to create a new communicator with the same membership as an existing communicator, but with ranks assigned differently.

And, for completeness, we note that the communicator `MPI_COMM_SELF` exists. It contains the caller (only), and thus the caller's rank in it is zero. It cannot be freed. It is of minimal use.

MPI often seems good at providing multiple routes for achieving much the same end. We have seen how to split a communicator, potentially into several non-overlapping comminicators in a single call, using `MPI_Comm_split`.

The alternative approach is via MPI groups, a concept not covered in this course, and `MPI_Comm_create` or `MPI_Comm_create_group` which creates a communicator from a group.

Castep mostly uses `MPI_Comm_split`.

MPI I/O

The following slides give a brief overview of MPI I/O, and why one might wish to use it.

In many cases, one doesn't. One already knows the non-MPI, standard, I/O routines in one's language, and one just uses those.

MPI I/O has the concept of a non-blocking (asynchronous) read: one can request a read, return immediately, and then check later to see if the read has actually occured.

MPI can use local, or shared (in a communicator), file pointers.

MPI assumes that a file contains entries of just a single datatype, and measures offsets in units of that datatype. It does not mandate this, and it is possible to use MPI I/O on files which contain a mixture of datatypes.

MPI I/O does not do formatted I/O (the conversion of numbers to strings).

MPI I/O was not part of version 1 of the standard. It was developed by IBM and adopted by version 2 of the MPI standard (1997). That is not to say that in 1997 every installed version of MPI instantly supported the whole of MPI-2.

Native I/O Calls (in pseudocode)

```
open(file)
if (rank==0)
  write(file)

call mpi_barrier

if (rank==1)
  read(file)
```

```
open(file)
if (rank==0)
  write(file)
  flush(file)

call mpi_barrier

if (rank==1)
  read(file)
```

```
if (rank==0)
  open(file)
  write(file)
  close(file)

call mpi_barrier

if (rank==1)
  open(file)
  read(file)
```

Using native I/O, when can rank one reasonably expect to see what rank zero wrote?

Same node	N	Y	Y
Different node, NFS	N	N	Y?

Consistency Problems

Because file access is slow, much buffering and caching automatically occurs.

Until one calls `flush` or `close`, there may not be a guarantee that one's run-time library has actually made an OS I/O call: it may be trying to coalesce multiple calls.

> *The three types of buffering available are unbuffered, block buffered, and line buffered. When an output stream is unbuffered, information appears on the destination file or terminal as soon as written; when it is block buffered many characters are saved up and written as a block; when it is line buffered characters are saved up until a newline is output.*
>
> *Normally all files are block buffered.*

(From '`man setbuffer`')

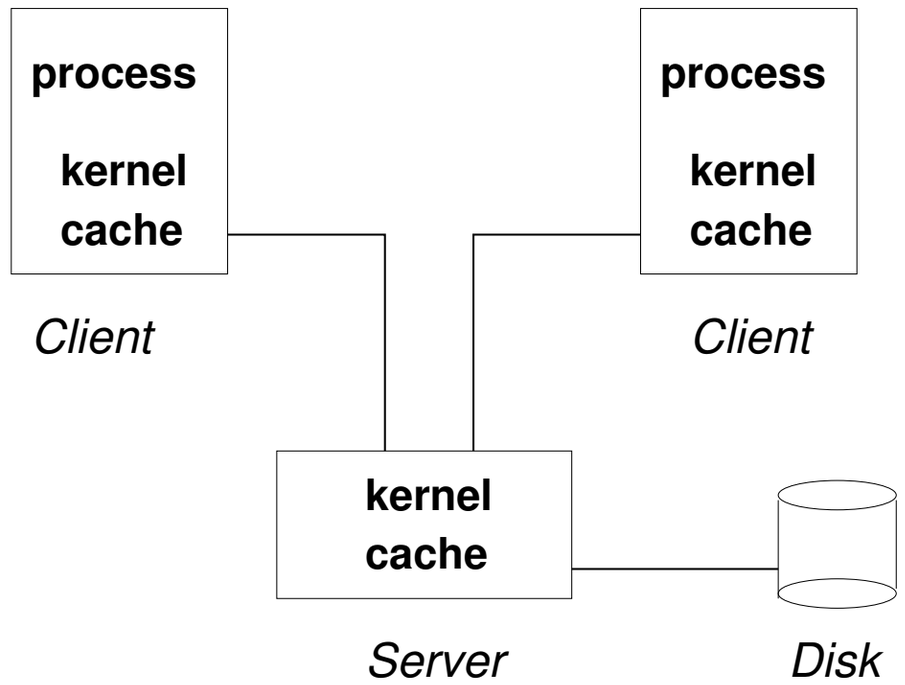
Even if the kernel on the local node has received the I/O request, NFS does not guarantee that it is immediately passed to the server, much less that all other clients are immediately sychronised.

Fast or Consistent

Caching local disks is easy. All access to them must go via the local kernel, so the kernel can cache aggressively knowing that nothing else can modify the disk.

Caching remote disks is hard. Caching will occur on the remote fileserver, where the disks are local anyway, but that cache is slow compared to a local cache which can be accessed without network overheads and latencies.

A local cache will become inconsistent if another client updates the cached file. The approach of NFS up to version 3 inclusive is not to care. The protocol does not permit a client to say it is caching a file, and broadcasting news of all file updates to all clients which could potentially be holding local copies would destroy performance and scalability. So clients do cache, with time-outs of a few seconds, and one lives with the consequences.



Fast or Consistent (2)

NFS tends to assume that cached data is valid if a check on the file's modification time shows that it has not been updated since the cached version was stored. Its approach to modification times (and other file attributes) can be tuned for different circumstances. The default is to cache this locally for a few seconds to tens of seconds.

This local caching of attributes can be turned off. This makes operations slower, in general, as almost every operation becomes a check of the file modification time on the remote server, but it does improve data consistency.

With attribute caching on, almost anything is possible. The bottom right example in the table on side 219 could return 'file not found' if the directory, with the file absent, were locally cached, and the directory's last modification time had been checked sufficiently recently that no new check was done for the open.

Clients can also cache writes locally, and may fail to pass writes on to the server until `close()` is called. This means that `close()` may return any of the errors expected of a write, such as no space on device, or quota exceeded.

Even turning off attribute caching is not perfectly safe. Consider what happens if a file is modfied, a client reads it and caches the result, and the file is modified again. The client will notice if the last modified timestamp is now different. If one is using ext4, which stores timestamps in nanoseconds, all should be well. If using ext3, which stores timestamps in seconds, there could be a problem.

NFS v4

Version 4 of NFS does offer one significant improvement for caching. That is the idea of file delegation. A client can obtain a read, or read/write, delegation on a file. This allows it to cache it, with a guarantee that should something happen that it needs to know about, it will be told explicitly, and the operation will be delayed until it has had time to respond. (If a client with a write delegation is disturbed by another client requesting a read, any locally-modified data needs to be sent back to the server before the read request can be fulfilled.)

Delegations suffer from the same problems as file locks in the face of unreliable networks and clients – what does one do with a client with a delegation and which is now uncontactable for reasons unknown? The NFS v4 delegation answer is that one times it out after a few seconds, and cancels the delegation. In some circumstances (network faults) this will leave the client thinking it still has a valid delegation.

Delegations also can worsen performance if activity from many different clients on the same file causes many revocations to occur.

Not NFS

CIFS, Lustre, GPFS, ...

All make different compromises between performance, reliability, simplicity, consistency guarantees, etc.

One advantage of MPI I/O is that it provides a set of guarantees which should be independent of the underlying filesystem. So one should get consistent behaviour, and reasonably good performance too.

Other Native Consistency Problems

```
open(file)
if (rank==0)
  write(offset=0, file)

call mpi_barrier

if (rank==1)
  write(offset=0, file)
```

Whose data end up in file? Not defined. It could even be a mixture. However, convert this to using MPI I/O, and the result is defined (as rank 1's data).

Safety with native I/O

Native I/O is completely safe if one either:

reads from any number of ranks

or

writes from just one rank (and reads from no other rank).

The only very mildly adventurous may try writing non-overlapping regions from multiple ranks, or switching between the above regimes with close / barrier / open.

Anything else is really quite adventurous, in that it is likely to work as naïvely expected on a code running on a single node, but on multiple nodes and an underlying parallel filesystem. . .

MPI I/O and files

Files are opened in a given communicator. This might be `MPI_COMM_SELF` for purely local access, or it might be `MPI_COMM_WORLD` (or any other communicator).

When opened, all processes must give a filename which maps to the same physical file. If not passing info, set the `info` parameter to `MPI_INFO_NULL`. (The info structure contains performance hints.)

```
MPI_File_open(MPI_Comm comm, const char *name, int amode,
                  MPI_Info info, MPI_File *fh)
```

Once opened, a file *must* be closed before `MPI_Finalise` is called.

```
MPI_File_close(MPI_File *fh)
```

The mode is generated by oring constants such as `MPI_MODE_RDONLY`, `MPI_MODE_RDWR`, `MPI_MODE_WRONLY`, `MPI_MODE_CREATE`, `MPI_MODE_EXCL`, `MPI_MODE_APPEND`. Fortran users can simply add these constants, rather than using `IOR`, and everything except the filename is a default integer in Fortran.

The above constants will be familar from C/POSIX. Three new ones are added: `MPI_MODE_DELETE_ON_CLOSE` (automatic removal of temporary files), `MPI_MODE_SEQUENTIAL` (forbids any form of seeking), and `MPI_MODE_UNIQUE_OPEN` (claims file will not be accessed by any other means during program's execution, including not being read by external programs/backup systems).

Reading and Writing

People used to C will wish to think in terms of bytes and seeks.

```
MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
MPI_File_read(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Status *status)
MPI_File_write(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Status *status)
```

And to determine how many elements were read/written:

```
MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
```

In these cases each process maintains its own independent file pointer, and the calls are synchronous.

Note that we have seen `MPI_Get_count` used on an `MPI_Status` object before, when dealing with MPI messages.

Unlike other MPI calls, MPI I/O calls default to returning, not aborting, on errors. So one should check the function return value (or, in Fortran, the value of `ierr`).

For `MPI_File_seek` whence may be `MPI_SEEK_SET` (from beginning), `MPI_SEEK_CUR` (from current), `MPI_SEEK_END` (from end).

Non-blocking

```
MPI_File_iread(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Request *req)
MPI_File_iwrite(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Request *req)
```

These are analogous to `MPI_Isend` and `_Ireceive`. One must test the request data structure, using `MPI_Test` or `MPI_Wait`, before one can assume that the operation has completed, and, in the case of `_iwrite`, before one can reuse its buffer.

C programmers beware!

`MPI_File_open` takes as an argument a pointer to an allocated (but uninitialised) `MPI_File` structure. C's `fopen` returns a pointer to a structure it allocates and initialises.

The MPI read and write calls pass the structure, not a pointer.

`MPI_File_open` with a mode of `MPI_MODE_WRONLY|MPI_MODE_CREATE` does not truncate an existing file, whereas C's `fopen()` would. Either call `MPI_File_open` followed by `MPI_File_set_size` or `MPI_File_delete` followed by `MPI_Barrier` followed by `MPI_File_open`.

C (and Fortran) programmers may be used to leaving files open and relying on the program's exit to close them. MPI says this is not correct for MPI I/O.

Deleting

```
MPI_File_delete(char* name, MPI_Info info)
```

The info argument will probably be `MPI_INFO_NULL`.

If called on all ranks, most will return an error of `MPI_ERR_NO_SUCH_FILE`. There is no need to do this.

Behaviour if any process has the file open is implementation dependent. It may even cause the deletion to fail and `MPI_ERR_FILE_IN_USE` or `MPI_ERR_ACCESS` to be returned.

`MPI_File_delete` is best used with barriers.

Truncating

```
MPI_File_set_size(MPI_File fh, MPI_Offset offset)
```

must be called on all ranks of the communicator with the file open, and all must call with the same offset.

It may be implemented by unsynchronised calls to `ftruncate` on all ranks, so explicit synchronisation before further writes are made is necessary in most cases.

Fortran programmers will wish to know that `MPI_Offset` is an integer of kind `MPI_OFFSET_KIND`, so

```
call MPI_File_set_size(fh, int(0,MPI_OFFSET_KIND), ierr)
call MPI_File_set_size(fh, 0_MPI_OFFSET_KIND, ierr)
```

or

```
integer(kind=MPI_OFFSET_KIND) :: offset
offset=0
call MPI_File_set_size(fh, offset, ierr)
```

are three valid approaches.

Seeking Combined

```
MPI_File_seek(MPI_File fh, MPI_Offset offset, MPI_SEEK_SET)
MPI_File_read(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Status *status)
```

can also be written as

```
MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
     MPI_Datatype datatype, MPI_Status *status)
```

and analogously for `write_at`, `iread_at` and `iwrite_at`.

But in this form no file pointer is modified.

MPI calls this form of read or write 'explicit offset.'

Again, Fortran programmers need to know that the offset is an integer of kind `MPI_OFFSET_KIND`.

A Conversion (cf. slides 115 and 117)

```
FILE *img;

img=fopen("mandel.pam","w");



MPI_Barrier(MPI_COMM_WORLD);


if (rank==0)
  hdr=fprintf(img,"P6\n%d %d 255\n",
              SIZE,SIZE);


MPI_Bcast(&hdr,1,MPI_INT,0,
              MPI_COMM_WORLD);
for(i=...
  ...
  fseek(img,hdr+3*SIZE*i,SEEK_SET);
  fwrite(line,1,3*SIZE,img);
}

MPI_Finalize();
```

```
FILE *img;          char str[100];
MPI_File mpi_img;   MPI_Status st;

MPI_File_open(MPI_COMM_WORLD,"mandel.pam",
   MPI_MODE_WRONLY|MPI_MODE_CREATE,
   MPI_INFO_NULL,&mpi_img);
MPI_File_set_size(mpi_img,0);
MPI_Barrier(MPI_COMM_WORLD);

if(rank==0){
  hdr=sprintf(str,"P6\n%d %d 255\n",
                  SIZE,SIZE);
  MPI_File_write(mpi_img,str,hdr,
                       MPI_BYTE,&st);
}
MPI_Bcast(&hdr,1,MPI_INT,0,
                MPI_COMM_WORLD);
for(i=...
  ...
  MPI_File_write_at(mpi_img,hdr+3*SIZE*i,
    line,3*SIZE,MPI_BYTE,&st);
}
MPI_File_close(&mpi_img);
MPI_Finalize();
```

The Shared File Pointer

Whereas `MPI_File_read` (and friends) update a file pointer local to that MPI rank, and `MPI_File_read_at` (and friends) update no file pointer, there is another option. MPI also maintains a shared file pointer, one per open file, which certain operations update and follow.

It can be used only if all ranks have the same file view (see later), which, by default, they will.

```
MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)
MPI_File_get_position_shared(MPI_File fh, MPI_Offset offset)
MPI_File_read_shared(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Status *status)
MPI_File_write_shared(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Status *status)
MPI_File_iread_shared(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Request *req)
MPI_File_iwrite_shared(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Request *req)
```

These operations do not move the per-process pointers, just as the non-shared operations do not update the shared file pointer. `MPI_File_seek_shared` is a collective operation – it must be called on all ranks, and with the same offset and whence.

Ordering and Consistency

If n ranks call a shared operation, it is guaranteed that the effect is as though the operations were atomic – one read or write will not be interrupted by another read or write. But the ordering is not guaranteed unless one explicitly puts lots of barriers or similar in one code.

So if the aim is to get each rank to read a complete record from a file, but the mapping of the records to ranks does not matter, then these shared routines will do the job well.

Note that standard operating system I/O with a file opened for writing in append mode looks a little bit like using a shared pointer, but not completely like using a shared file pointer. With the same file open on two processes, both of which call something like

```
write(fd,"Hello World\n")
```

one may see

```
HellHello World
o World
```

(With such a short message, this is extremely unlikely, but the principle stands. In theory the interleaving could be even worse.)

Replace `write` by `MPI_File_write_shared` and this won't happen.

Consistent Ordering of Output

```
char msg[100];
MPI_File fh; MPI_Status status;

sprintf(msg,"I am process %d out of %d\n",rank,nproc);

MPI_File_open(MPI_COMM_WORLD,"hello.out",MPI_MODE_WRONLY|
          MPI_MODE_CREATE|MPI_MODE_APPEND,MPI_INFO_NULL,&fh);
MPI_File_write_ordered(fh,msg,strlen(msg),MPI_CHAR,&status);
MPI_File_close(&fh);
```

`MPI_File_write_ordered` must be called by all processes in a communicator, and the final effect will be as though the ranks had called it in their natural order. So this time there is a guarantee of the ordering

```
I am process 0 out of 4
I am process 1 out of 4
I am process 2 out of 4
I am process 3 out of 4
```

It is implementation-dependent whether processes will return from calling `MPI_File_write_ordered` before all processes have reached it (i.e. whether it acts like `MPI_Barrier`), and whether the writes occur in time in their natural order, or whether some combination of seeks and writes produces the correct ordering of data in the file in an unexpected time ordering.

Unsuprisingly, the corresponding read exists.

```
MPI_File_read_ordered(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Status *status)
MPI_File_write_ordered(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Status *status)
```

These operations do update the shared file pointer.

There is no `iread` or `iwrite` for `ordered` or `shared` operations.

More Collective I/O

Collective MPI I/O is useful because it is easier for the MPI library to optimise. There is a guarantee that all ranks within a communicator will make a call, although counts and offsets may differ from rank to rank. The calls must match just like data-moving collective calls, but note that the communicator is not passed explicitly. This is because an MPI filehandle is passed, and an MPI filehandle exists within a given communicator, so it would be superfluous to restate that communicator.

Not all collective I/O routines use the shared file pointer.

```
MPI_File_read_all(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Status *status)
MPI_File_write_all(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Status *status)
```

are collective operations, but use the local file pointers. If all ranks are going to read or write approximately simultaneously, these collective routines may give better optimisation, encouraging the MPI library to accumulate data into chunks of at least file system block size before making any OS I/O calls.

(There exist non-blocking collective I/O calls using 'split collectives' and `MPI_File...all_begin` and `MPI_File...all_end` calls. They are not covered in this course.)

File Views

Each MPI process has its own 'view' of a file. By default, the 'view' is that a file is sequential bytes, starting at the beginning. However, a view could be that a file is sequential integers, with all the ranks interleaved. In other words, if there are eight processes, process zero could ask for a view in which it sees the zeroth, eighth, sixteenth, twenty-fourth integer of the underlying file as being the first four integers in its view. (Though one might question the resulting efficiency.)

This gets complicated, and fortunately if no view is specified, the default is what one might imagine it to be.

Particularly in Fortran it may be useful to change the default view.

```
MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
  MPI_Datatype ftype, const char *datarep, MPI_Info info)
```

Which must be called by all processes in the communicator which has the file open. If this is not called, the default is

```
MPI_File_set_view(fh,0,MPI_BYTE,MPI_BYTE,"native",MPI_INFO_NULL)
```

The `disp` argument is measured in bytes, always, and is used to skip any header. It may be different on different processes.

The `etype` and `ftype`, if the same, set the units for calls to `MPI_File_seek`. The `etype` must be the same on all processes. Reading or writing things which are not a multiple of `etype` in length is not at all recommended.

More views

```
integer(kind=MPI_OFFSET_KIND) :: disp
disp=0
call MPI_File_set_view(fh,disp,MPI_DOUBLE_PRECISION,MPI_DOUBLE_PRECISION, &
                       "native",MPI_INFO_NULL,ierr)
```

would be a reasonable way for a Fortran programmer to access a file in terms of double precision variables (of unknown length). If, after a while, the file changes to containing integers, then one approach is:

```
integer :: fh,ierr
integer(kind=MPI_OFFSET_KIND) :: offset,disp
! Get position in current view in units of etype
call MPI_File_get_position(fh,offset,ierr)
! Convert to absolute displacement from start of file
call MPI_File_get_byte_offset(fh,offset,disp,ierr)
! Set a new view
call MPI_File_set_view(fh,disp,MPI_INTEGER,MPI_INTEGER, &
                       'native',MPI_INFO_NULL,ierr)
```

For C programmers:

```
MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset, MPI_Offset *disp)
```

which takes an offset and returns a byte displacement, and

```
MPI_File_get_position(MPI_File fh, MPI_Offset *offset)
```

Native

The `datarep` argument takes one of three (string) values:

`native`: fast, and readable on the machine which generated it. The format used in memory is simply transfered to the file.

`internal`: compatible with heterogeneous hardware running the same version of MPI, so conversions may need to occur. (With this format, OpenMPI will read files written by OpenMPI on big-endian or little-endian machines, for instance, but it might not read anything from MPICH.)

`external32`: a representation defined by the MPI standard, so readable by any MPI implementation on any machine. Slow.

MPI I/O and User-Defined Datatypes

User-defined datatypes can be useful, and an oft-cited example is extracting columns from a matrix stored by row, or vice versa.

They can be complicated, as the operations for building them up permit both repeating an existing type a number of times, and joining two existing types.

They are not equivalent to C's structures, or anything similar in Fortran. The precise arrangement in memory of items in a C structure, or in Fortran, is not very well defined, so attempting to map these onto MPI is, except in simple cases, unreliable.

One could easily give a whole lecture on MPI datatypes. Here we will briefly scratch the surface. The use of user-defined datatypes is not restricted to MPI I/O – standard MPI communication (send, receive, broadcast) can use them too.

Simple Construction

We will look at just one way of creating a new MPI type.

```
MPI_Type_vector(int count, int blocklength, int stride,
  MPI_Datatype oldtype, MPI_Datatype *newtype)
MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb,
  MPI_Aint extent, MPI_Datatype *newtype)
```

(In Fortran, `MPI_Datatype` is an integer, and `MPI_Aint` an integer of kind `MPI_ADDRESS_KIND`.)

To create a datatype which is made up of two doubles, and then padding until one reaches $2\times$`nproc` doubles:

```
MPI_Type_vector(1,2,2,MPI_DOUBLE,tmp_type);
MPI_Type_create_resized(tmp_type,0,2*nproc*sizeof(double),my_type);
MPI_Type_commit(my_type);
```

`MPI_Type_commit` must be called before the type is used in any communication (including I/O). An uncommitted type can be used by the `MPI_Type` building operations.

The recipe for a block of `n` doubles followed by `m` doubles worth of space is:

```
MPI_Type_vector(1,n,n,MPI_DOUBLE,tmp_type);
MPI_Type_create_resized(tmp_type,0,(n+m)*sizeof(double),my_type);
```

and clearly there is further flexibility not described here.

Fortran Fun

Type checking is rather strong with the Fortran mpi modules, and there is no `sizeof()`, so

```
call MPI_Type_create_resized(tmp_type,0,2*nproc,my_type,ierr)
```

won't work. It should be

```
integer :: dbl_size

call MPI_Type_size(MPI_DOUBLE_PRECISION,dbl_size,ierr)
call MPI_Type_create_resized(tmp_type,0_MPI_ADDRESS_KIND, &
     int(2*nproc*dbl_size,MPI_ADDRESS_KIND),my_type,ierr)
```

Remember that, in Fortran, a constant of the form 123.4_DP is the value 123.4 of kind DP (a declared and initialised variable). It looks odd with all the underscores in `0_MPI_ADDRESS_KIND` but that is simply `int(0,MPI_ADDRESS_KIND)`

User Defined Datatype with I/O

```
MPI_Offset disp;

disp=2*my_rank*sizeof(double);

MPI_File_set_view(fh,disp,MPI_DOUBLE,my_type,"native",MPI_INFO_NULL);

MPI_File_read(fh,buf,6,MPI_DOUBLE,MPI_STATUS_IGNORE);
```

This will read six doubles (and should be used for reading multiples of two doubles only, as the underlying type contains two doubles). On rank zero those six doubles will be numbers 0, 1, 2*`nproc`, 2*`nproc`+1, 4*`nproc`, 4*`nproc`+1 in the file. On rank one they will be at 2, 3, 2*`nproc`+2, 2*`nproc`+3, 4*`nproc`+2, 4*`nproc`+3.

If one must interleave data in this fashion, these sort of calls offer a much faster way of reading, and especially writing, than the alternative, particularly if one is able to use the collective `_all` variants. But one should check how well your MPI implementation optimises this, particularly on remote filesystems. I have seen fine-grained interleaving cause write speeds to collapse to under 150 bytes/second on a system capable of over 100MB/s, so a factor of almost a million lost!

Types can be freed with `MPI_Type_free(MPI_Datatype *oldtype)`

MPI I/O advantages

MPI I/O needs to run on top of another filesystem, maybe NFS, maybe Lustre, or GPFS, or some other network filesystem.

It offers a consistency model which combines utility with ease of implementation. The use of `MPI_MODE_UNIQUE_OPEN` and of non-blocking reads and writes have obvious performance advantages if the underlying filesystem is able to exploit them. Using `MPI_MODE_UNIQUE_OPEN` with a communicator of `MPI_COMM_SELF` should be particularly advantageous for caching.

It can often combine what would be several native I/O calls into a single MPI I/O call. Seeks can be combined with read or write, and the use of derived datatypes can combine small scattered reads into larger I/O calls.

The collective operations can be optimised by the library, to coalesce small I/O operations, or to combined strided operations (arising from user-defined data types) into contiguous ones.

MPI I/O: would I use it?

Probably not! I would if:

I/O was taking significant time *and*

I was intending to run on a machine with something better than a shared NFS filesystem.

For most of the code I run, neither is true. At which point doing all I/O on the root node and transferring data via MPI calls seems reasonable, as does careful use of seeks for multiple nodes to write to a common file.

Another scenario in which MPI I/O might be bad is if each node has its own local scratch disk, and most I/O can be done to those scratch disks. Simple and cheap, and the complete lack of sharing at hardware or software level should make scaling perfectly linear and caching easy.

On the other hand, it might be very good if it were possible to exploit its non-blocking reads. This does rely on the implementation being able to perform a usefully non-blocking read (perhaps by starting a blocking read in a separate thread). If the application lends itself to the use of collective I/O there could also be advantages, even with NFS (as the library, in theory, could realise that NFS was involved, and decide to do all the I/O from a single rank).

Hardware

Parallel Computers: the Concepts

Modern supercomputers are generally *parallel computers*. That is, they have more than one CPU. So are desktops and laptops now that almost all processors have multiple cores.

Some tasks are clearly suited to being done by a 'farm' of 'workers' working simultaneously, whilst others are not. As two examples:

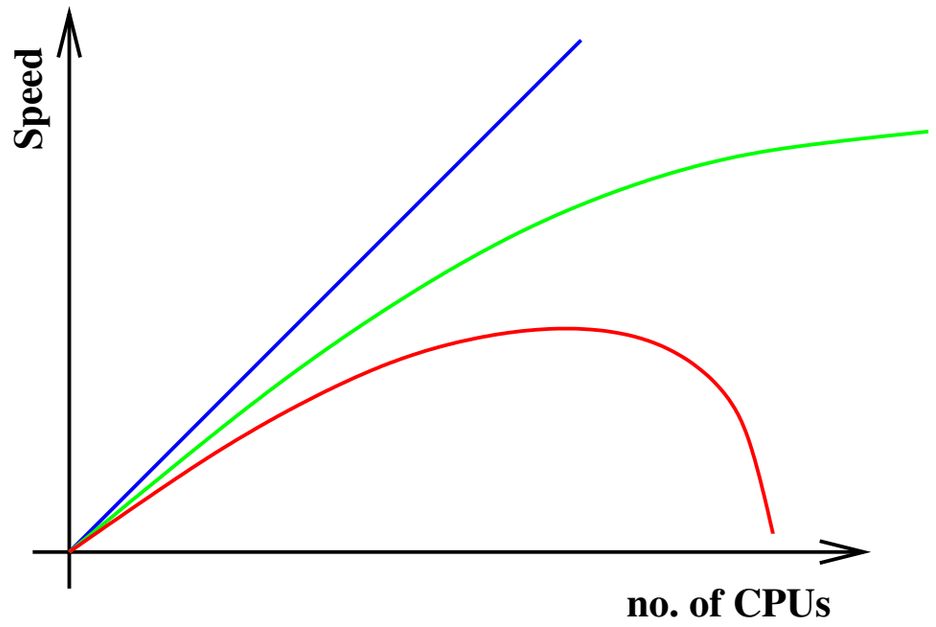
Integration of differential equation over very many timesteps. Clearly one cannot start the 5,000th timestep until the 4,999th has been finished. The process is fundamentally serial.

Dumb Factorisation of a large number. The independent trial factors from 2 to $\sqrt{n}$ are readily distributed amongst multiple processors.

A simple example of parallelisation has alrady been seen in the various 'multimedia' instructions. This is known as SIMD parallelism: Single Instruction Multiple Data. The parallelism discussed in this section is MIMD (Multiple...).

Scaling

How much faster does a code run when spread over more CPUs?



From top to bottom: Linear scaling (rare!), Amdahl's Law (see below), The Real World

Notice that the speed is not monotonic in the number of CPUs

Amdahl's Law

Amdahl was a pioneer of supercomputing and an employee of IBM.

This law assumes that a program splits neatly into an unparallelisable part, and a completely parallelisable part. It claims:

$$t_n = t_s + t_p/n$$

The total run time on n processors is the time for the serial part of the code, plus the time the parallel part would take on a single processor divided by the number of processors.

Consider $t_s = 0.2$ and $t_p = 0.8$. Then $t_1 = 1.0$, $t_{32} = 0.225$ and $t_\infty = 0.2$.

On 32 processors the speedup is $4.5\times$ and the efficiency is just 14%.

Bigger is better

Suppose t_s and t_p scale differently with problem size.

Assume t_s scales as N and t_p as N^3 and consider a problem $4\times$ as large as before. Now

$t_s = 0.8$ and $t_p = 51.2$ giving $t_1 = 52$ and $t_{32} = 2.4$.

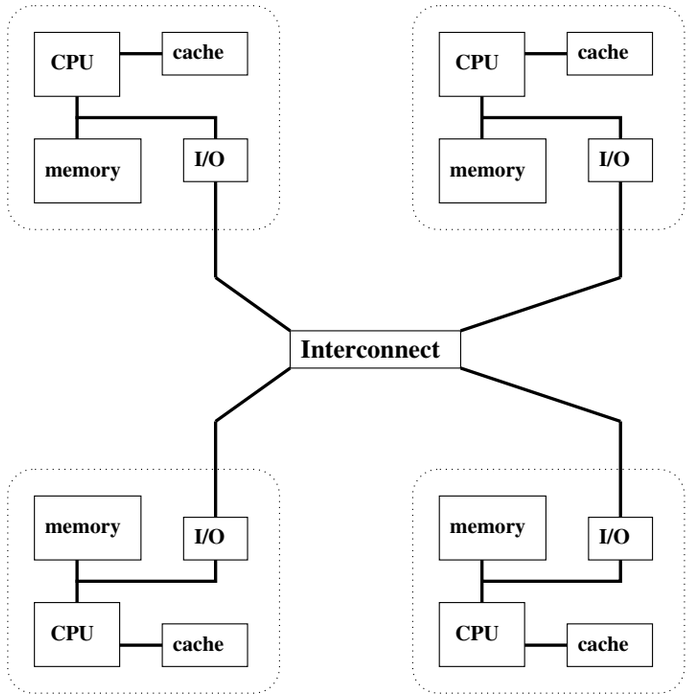
Now the speedup on 32 processors is $21\times$, and the efficiency is now over 67%.

Supercomputers like *big* problems.

Conversely, workstations hate big problems, as their various caches become less effective and their overall efficiency falls.

MPP and SMP

We first consider the MPP design of parallel computer. It is simple, consisting of lots of separate single-processor computers with a fast network between them. Each separate sub-computer, or node, has its own memory, and, in some cases, even its own disk drive. Such a parallel computer is called a *distributed memory computer* or *massively parallel processor*



Topologies

There are many different ways of connecting nodes together, as ever governed by cost and practicality.

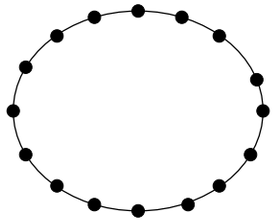
Two useful ways of characterising a network are the ‘diameter’, the maximum number of hops from one node to another, and the bisectional bandwidth, the bandwidth between two halves of the machine.

	Bandwidth	Diameter
Ring	2	$N/2$
2D Grid	$\sqrt{N}$	$2\sqrt{N}$
2D Torus	$2\sqrt{N}$	$\sqrt{N}$
Hypercube	$N/2$	$\log_2 N$
Tree	2	$2\log_2 N$
Fat tree	$N/2$	$2\log_2 N$
X-bar	$N/2$	1
3D X-bar	$N/2$	3

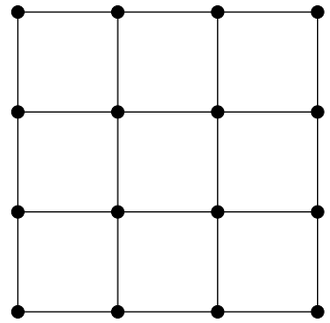
The Cray T3D was a 2D torus, the IBM SP2 a fat tree, the SGI Origin2000 a form of hypercube, and the Hitachi SR2201 a 3D X-bar. Today fat trees are most common.

Ideally the network topology should not be apparent to the user.

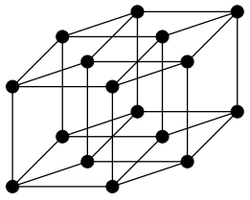
16 Nodes...



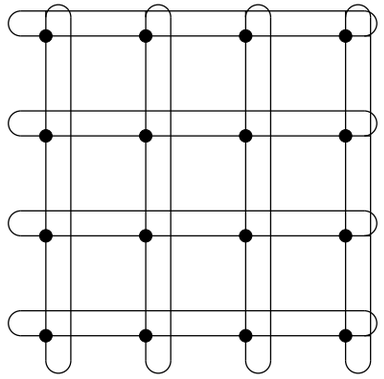
Ring (1D torus)



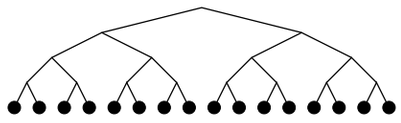
2D mesh



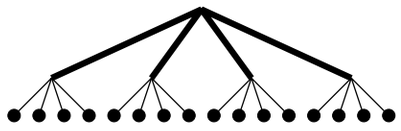
Hypercube



2D torus



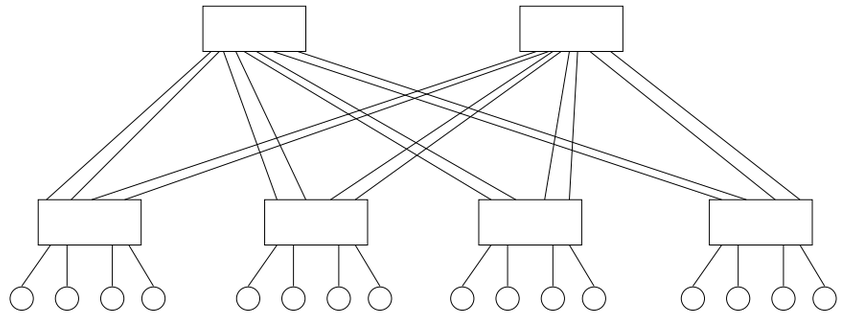
Tree (log 2)



Fat Tree (log 4)

Clos Networks

The obvious problem with the fat tree is the switch at the root of the tree. It is big and therefore expensive and rare. So in practice a slightly different configuration is commonly used, invented by Charles Clos (Bell Labs, 1952). Assuming that eight-port non-blocking switches exist cheaply, can we build a network for 16 ports? Yes:



This may look unimpressive: six eight-port switches have become one sixteen-port switch. On the other hand, if any eight nodes wish to talk to any other eight nodes, there is a way of routing packets which results in no link being congested, and full bandwidth being obtained. There is also no single point of complete failure.

The difficulty is that the routing depends on the traffic pattern. If one attempts to build a static routing tree (e.g. traffic from node 1 to node 5 always goes via the left-hand top switch), then one can only ever support some sets of eight pairs communicating with no blocking.

Static routing is attractive: cheap and low latency. Many switches will do this.

(Note that with eight-port switches and two layers the most ports we can make is 32: eight leaf switches at the bottom, each with a single link to each of four switches at the top. Twelve switches in total. Then we can recurse: two layers of 32 port switches give 512 ports, two layers of n port switches give $0.5n^2$ ports.)

Performance

Another important characteristic of the interconnect is its raw performance, both bandwidth and latency. These are most usefully measured using a standard interface such as MPI, and not using the hardware directly.

Ideally the time to transmit a packet is simply

latency + size / bandwidth

If size $<$ latency $\times$ bandwidth, then the latency will dominate.

Also ideally communication between a pair of nodes is unaffected by any other communications happening simultaneously between other nodes. Such a network is called *non-blocking*.

Typical figures are 1 to 3 GB/s bandwidth and 1 to 3 μs latency. Clusters using 1GBit/s ethernet typically run at around 100 MB/s and 20 μs.

Parallelisation Overheads

Amdahl's law assumes that there are no overheads associated with parallelisation. This is certainly a gross approximation.

Consider the case where each node must exchange data with every other node at some point in the program: some sort of rearranging of an array spread over all the nodes. E.g. an FFT

Each node must send $n-1$ messages of size a/n^2 where a is the size of the distributed array. Even assuming that the nodes can do this simultaneously, the time taken will be

$$(n-1) \times \left(\lambda + \frac{a}{n^2\sigma}\right) \approx n\lambda + \frac{a}{n\sigma}$$

where λ is the latency and σ the bandwidth.

Amdahl revisited

A better form of Amdahl's law might be

$$t_n = t'_s + t'_p/n + c\lambda n$$

where $t'_p > t_p$ and $t'_s > t_s$.

Now t_n is no longer a monotonically decreasing function, and its minimum value is governed by λ.

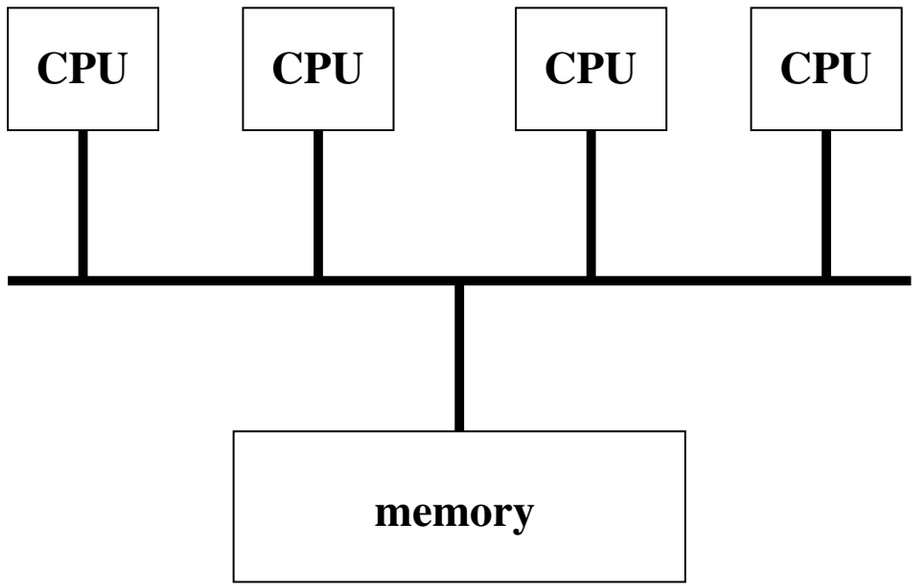
This form stresses that the quality of the interconnect can be more important than the quality of the processors.

Hence 'cheap' PC clusters work well up to about 16 nodes, and then for many applications their high latency compared to 'real' MPPs starts to be significant.

SMP: Bused Based

SMP (Symmetric Multi Processor, Shared Memory Processor) describes another class of multi-CPU computer.

The original, bus-based, SMP computer simply has multiple CPUs attached to a single system bus.



The architecture is *symmetric* (all CPUs are equivalent), and the memory is *shared* between them.

Shared memory

This is precisely what a modern multi-core CPU looks like, as a single core is equivalent to the old idea of a CPU.

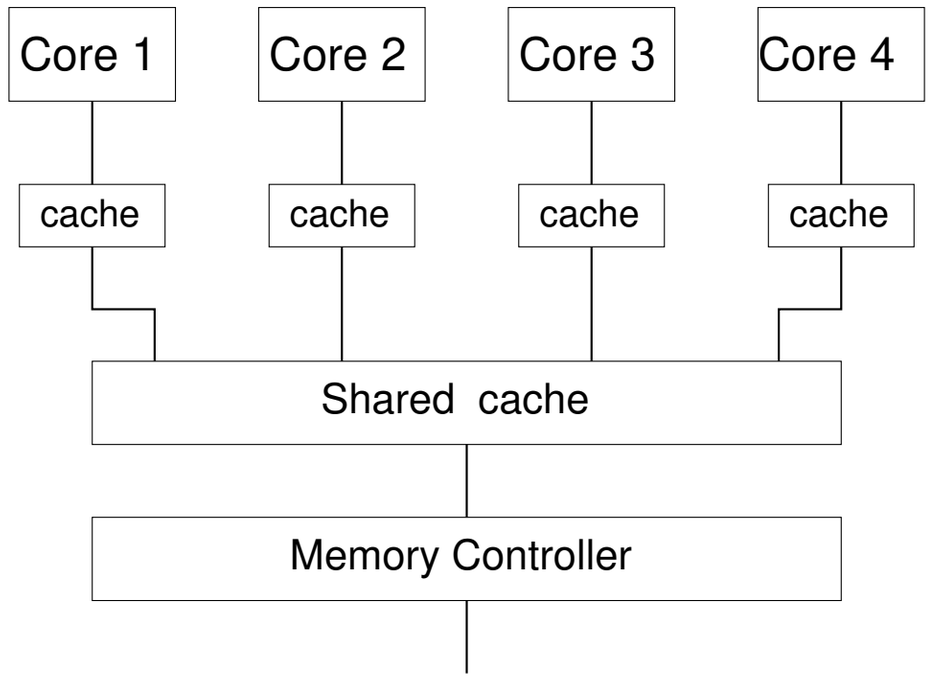
As all processors access the same main memory, it is easy for different parts of a program executing on different processors to exchange data. One CPU can write an array into memory, possibly from disk, possibly as the result of a calculation, then all other CPUs can read it with no further effort.

Programming is thus simple: all the data are in one place, and there is merely the little matter of dividing up the millions of instructions to be executed in a long loop between the multiple eager processors – a job so simple that the compiler can do it automatically.

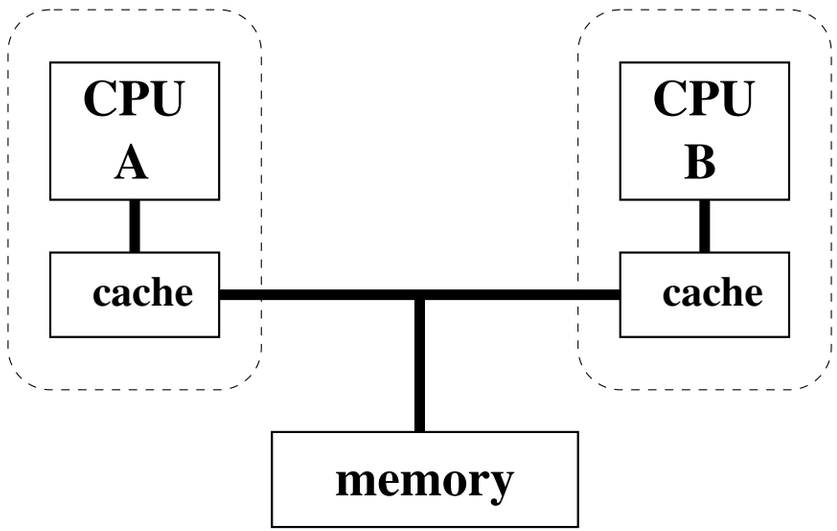
Except it is not quite that simple.

Two Heads are Better than One?

As in a conventional, single-CPU computer, the single processor typically spends between 75 and 95% of its time waiting for memory, trying to 'feed' two or more CPUs from one memory bank is clearly crazy. The memory was, and is, the bottleneck. The CPU was not. However the design is cheap, and is now ubiquitous within a CPU, as the diagram below illustrates, and is common in multi-socket designs.



Cache coherency



Processor A reads a variable from memory. Later, it reads the same variable, which it can now get directly from its cache, without troubling the system bus.

Only it can't. For what if processor B has modified that variable, and processor A needs the new value? If processor B has a write back cache, the new value may not even have reached the main memory, with the current value being held in processor B's cache only.

Snoopy caches

The trivial solution is to abandon all caches.

An easy solution is to ban write-back caches, and to ensure that each cache '*snoops*' the traffic on the system bus, and, if it sees a write to a line it is currently caching, it must either update itself automatically, or mark its copy as being invalid.

These solutions severely compromise one's cache architecture, and often lead to a SMP machine generating more traffic to the main memory than a uniprocessor machine would running the same code. Thus a SMP machine can fail to reach the performance of a single-processor workstation based on the same CPU.

With either of these solutions, the definitive data are always those in the main memory.

Even single core single CPU workstations have a lesser version of this problem, as it is common for the CPU *and* the disk controller to be able to read and write directly to the main memory. However, with just two combatants, the problem is fairly easily resolved.

More Complexity: NUMA

Most modern SMP machines are not bus based. Internally they are configured like MPPs, with the memory physically distributed amongst the processors. Much magic makes this distributed memory appear to be global.

This (partially) addresses the poor memory bandwidth of the bus based SMP machines.

However, there are problems...

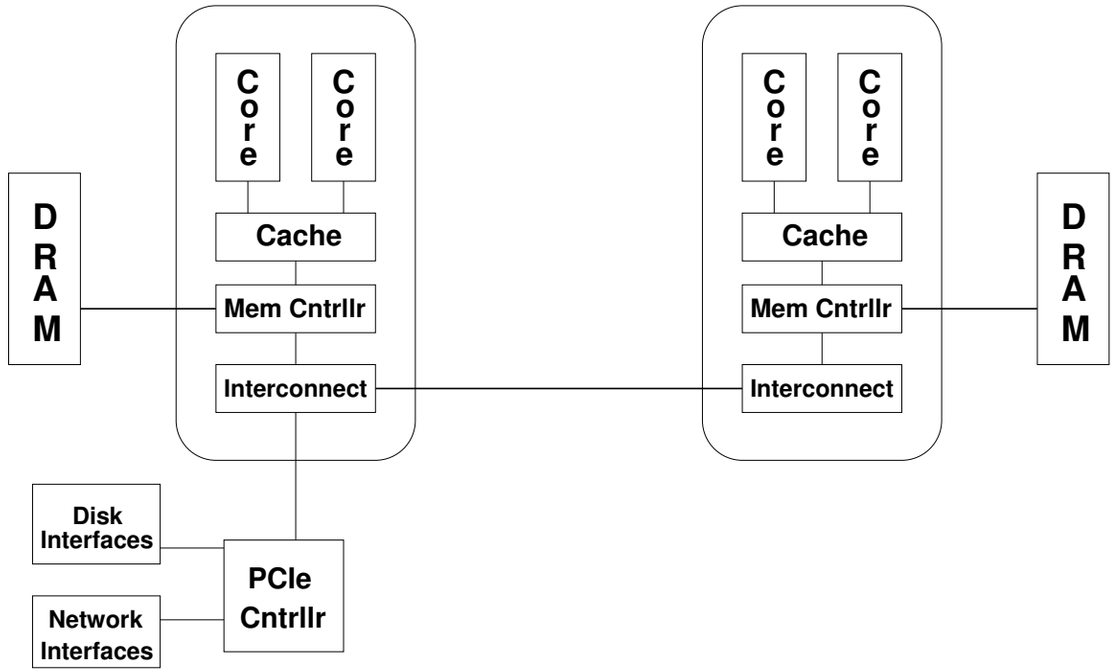
Not least that some memory is now local to a CPU, some attached to its neighbour(s), some its next nearest neighbour(s). Access times and bandwidths will vary depending on the relationship between the physical location of the memory and the core on which a process is executing. Hence the acronym Non Uniform Memory Architecture.

And magic costs money, and, in this case tends to degrade performance over an MPP, providing instead increased flexibility.

To emphasise that magic has been used to make even caches work correctly, one sometimes sees the acronym cc-NUMA, cache-coherent NUMA. Fortunately the alternative of non cache coherent NUMA is extremely rare.

Modern, small SMPs

Modern processors not only contain multiple cores, but also often contain the interconnect needed to construct SMP machines of two or four sockets. (The below diagram should really show four or six cores per CPU.)



AMD's HyperTransport (HT) interconnect and Intel's rather later Quick Path Interconnect (QPI) are quite similar. In both cases a CPU has about three links, which can connect either to other CPUs, or to I/O controllers (e.g. PCIe bus controllers). The above diagram shows the left-hand processor using two links, and the right-hand one. Not only is the memory NUMA, but so is access to disk and network.

NUMA in Action

A version of the Streams benchmark written in MPI gives a measure of memory bandwidth. The command `taskset` can be used to specify which cores it runs on. Here a dual socket machine with quad core processors:

Process count	cores used	bandwidth
1	1	9.8 GB/s
2	1&3	12.3 GB/s
2	1&5	12.3 GB/s
2	1&2	19.6 GB/s
4	1,3,5&7	13.0 GB/s
4	1,2,3&4	24.1 GB/s
4	1,2,5&6	24.1 GB/s
8	1–8	26.0 GB/s

Conclusion: each core has maximum b/w of c.10 GB/s
each socket has maximum b/w of c.13 GB/s

This was a dual socket 2.4GHz quad core Intel 'Nehalem'. Each processor has a 24 byte wide bus to DDR3/800 memory, so theoretically 19.2GB/s per socket. It seems that cores 1, 3, 5 & 7 are on one CPU (socket), and the even numbers on the other. Note that here the performance gain in moving from one process to eight on an eight core machine for a code with no inter-process communication is a factor of merely 2.7. The gain from using one core per socket to all four is a factor of just 1.3.

The Consequences of NUMA

If a processor is mangling an array, it now matters crucially that that array is stored in the memory on directly attached to that processor, and not on memory the other side of the machine. Getting this wrong can drop the performance by a factor of three or more instantly.

Whereas with MPP all memory accesses are guaranteed to be local, as one cannot access remote memory except by explicit requests at the program level, with SMP the compiler has many ways of getting things wrong.

```
for(i=0;i<10000000;i++)
   t+=x[i]*y[i];
```

Consider this on a two processor NUMA machine. If the code is split so that one processor stores the first 5000000 elements of each array in its directly attached memory, and does the first half of the loop, the other the second half of each array and does the second half of the iterations, then optimal performance is obtained. If the whole of `x` is stored in the memory local to one processor, the whole of `y` the other, then much reduced performance will result.

MESI solutions for caches

A typical SMP has extra bits associated with each cache line, which mark it as being on one of four states:

- Modified (i.e. dirty) – this state exists for uniprocessor machines too
- Exclusive (in no other cache)
- Shared (possibly in other caches too)
- Invalid

Modified implies exclusive, and a line must be exclusive before it can be modified.

A line fill for a read ensures that no other cache has the line modified or exclusive, then loads the line marked as 'shared.' A fill for a write also ensures than any caches with that line shared mark it invalid. In either case any cache with it 'modified' (there can be only one) writes it back to memory.

Thus a line can be:
In no caches
In one cache and marked as modified (or exclusive)
In one or more caches and modified (or exclusive) in none

More Messes

It may seem as though ensuring that each thread works on its own data, and rarely exchanges data with other threads, is sufficient to ensure performance. It isn't, for there is the overhead of checking to ensure that one core is not trying to update data held in another core's cache. All modern computers do this in hardware. Even if the compiler knows that two data items are distinct, the hardware will still check, and may need to do so as a thread may migrate from one core to another during its execution.

One method of checking simply broadcasts details of all line fills to all cache controllers, and the fill does not progress until the other controllers have had an opportunity to reveal that they held the line. The amount of broadcast traffic tends to scale as the square of the number of caches, so this works poorly for large numbers of CPU – in practice, beyond about four.

A significant improvement uses a 'directory'. A directory entry is associated with each line in memory, and records which caches have copies of the line. Then a fill need simply check the directory, contact only those caches listed (probably none), and proceed, updating the directory as it does so. In practice a directory which only provides partial coverage of the main memory can be used, falling back to broadcasting when the directory is incomplete.

Secondly, the important concept is not the sharing of data, but the sharing of cache lines. If two threads attempt to write to adjacent items in the same cache line, this is no better from the point of view of copying data around in a MESI system than if they were writing to the same element. This is sometimes called 'false sharing'.

Broadcast Failures

In 2006 I had the fun of testing an 8-socket Opteron server with dual core processors. The board design was cheap, and not quite up to AMD's recommendations, so the following results are not a fair reflection on AMD...

The measured bandwidth using the Streams benchmark was 2.0GB/s for a single process, peaked at 8.0GB/s for six, and 7.4GB/s for sixteen. Why am I convinced that this reflects a severe broadcast problem? The server design allowed me to remove physically four of the CPUs. The numbers I then measured were 4.4GB/s for a single process, and 16.3GB/s for eight.

Simply having the extra four CPUs present, and informed of what was happening, even if one did not use them, halved the performance of this machine for Streams! A single process memory latency benchmark moved from a poor 190ns to a very poor 290ns just by having the extra CPUs present.

Even Linpack, normally forgiving of poor memory subsystems, managed 26.4 GFLOPS running on eight cores (four CPUs present), and only 22 GFLOPS on 16 cores (8 CPUs present) with an array size of 40,000. The cores had a theoretical peak performance of 4.4 GFLOPS each.

Sharing, True and False

Naturally things get worse if multiple processors really are trying to update the same memory location. Not only does this need detecting, but once detected, it needs action to ensure that correct behaviour is observed. Corrective action tends to involve the automatic transfer of cache lines between CPUs. Not fast, as lines are big.

However, the important concept is not the sharing of data, but the sharing of cache lines. If two threads attempt to write to adjacent items in the same cache line, this is no better from the point of view of copying data around in a MESI system than if they were writing to the same element. This is sometimes called ‘false sharing’.

A False Sharing Example

```
#pragma omp parallel for private(j,ptr1,ptr2)
  for(i=0;i<=1;i++){
    if(i==0){
      ptr1=line;
      ptr2=line+2*OFFSET;
    }
    else{
      ptr1=line+OFFSET;
      ptr2=line+3*OFFSET;
    }
    for(j=0;j<(1<<28);j++){
      *ptr1+=*ptr2;
    }
  }
```

The above takes 2s to execute in a serial fashion on a certain dual core machine with a particular compiler. In parallel, it takes 1s. Unless OFFSET is one or two, in which case it takes over 12s in parallel, and still 2s in serial.

Inclusive Levels

There are three common approaches to a cache hierarchy:

- Data in one level is guaranteed to be in no other level.
- Data in level n is guaranteed to be in all levels $> n$.
- Neither of the above guarantees

Intel likes the second scheme, *inclusive* caches. In response to cache coherency requests, it need only check the last level cache, for if the data are not there, they can be in no other level.

AMD likes the first, *exclusive* caches, for the total amount of data cached is then the sum of the sizes of the levels, not simply the size of the last.

In Theory

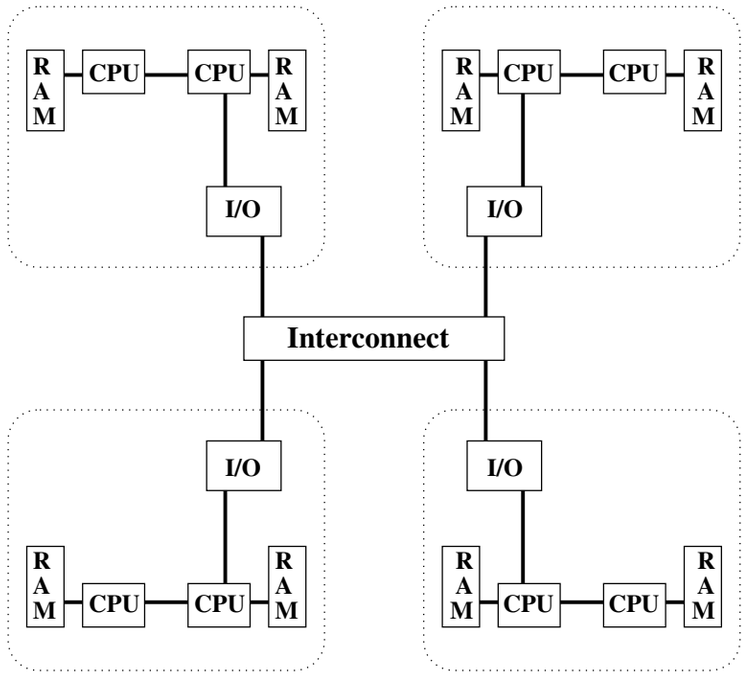
In theory a parallelised code has access to a greater number, and capacity, of caches, especially L1 caches. If the total memory requirement of the parallelised code is similar to that of the serial code, then the ratio of memory footprint to cache size is improved.

(Alternatively one can consider the memory footprint for an individual process decreasing by a factor of the number of processes now executing the parallel code.)

This can significantly improve the cache hit ratio, and thus parallelised code can run faster than expected by even linear scaling. Such *superlinear* scaling is regrettably rare, but is sometimes observed over small ranges of the number of processors, and for specific job sizes.

Modern, large MPPs

Modern MPP designs join SMP nodes like the above. Such a machine is awkward to program optimally, as one has both internode and intranode parallelism to address, with two very different interconnect speeds. A program which is merely correct, but not necessarily optimal, can ignore the SMP nature of the nodes, and consider the machine to be an MPP of all the cores.



MPI's Scaling Problem

Most MPI implimentations have a small receive buffer on each process for every other process. This gives an easy way of dealing with eager transfers and incoming envelopes. Suppose one has a couple of racks containing 64 nodes, each with 16 cores. That is 1024 cores in total, and, if the 'small' buffer is 4KB, each MPI process will need 4MB of buffer space. On a 16 core node that is 64MB, and the node probably has 64GB. So no-one cares.

If the supercomputer is more exotic, for instance 32 nodes each containing four Xeon Phi cards with 60 core Phis, then one has 7,680 cores. So 30MB of buffer space per process, and 1.8GB per Xeon Phi card. As the 5100 series of Phis have only 8GB of memory per card, this is somewhat embarrassing, even more so if that 'small' buffer is 8KB not 4KB.

If one's code can be rewritten to use 128 MPI processes, one per Phi card, and OpenMP within a card to make use of the 60 cores, then this N^2 MPI memory scaling problem gets reduced by a factor of 3,600. Or one could hope for a smarter MPI library.

Differing Speeds

The link between a CPU and its directly-attached memory is typically a 128 bit (or 256 bit or 384 bit) 2666MHz DDR4 bus. Theoretical bandwidth 42GB/s (or 85GB/s or 128GB/s), unidirectional.

The link between CPUs is typically one or two 16 bit 9.6GT/s HyperTransport or QPI links. Bandwidth 19GB/s (or 38GB/s), supporting traffic in both directions simultaneously.

The link from the I/O controller to the interconnect controller is typically 16 bit PCIe 3.1. This has a theoretical bandwidth of 16GB/s, and is bidirectional.

The interconnect itself is often EDR Infiniband 4X (or Intel Omni-Path). These have a theoretical bandwidth of 12GB/s, and are bidirectional.

Measured latencies vary more widely, from under 0.1μs for accessing memory within a node, to just over 1μs for an MPI transfer between nodes.

The cheap and nasty option is simply to use 10GBit/s ethernet, for a theoretical bandwidth of 1GB/s and a latency of around 10μs.

(All data current in 2018. The 2013 version of this slide said 1666MHz DDR3, 26GB/s (or 53GB/s), 6.4GT/s HT/QPI, PCIe 2.0 at 8GB/s, QDR Ininband 4X at 4GB/s, and 1GBit/s ethernet.)

Does Speed Matter?

The current trend seems to be for nodes to be increasing in internal bandwidth, and in peak MFLOPS, rather faster than (cheap) inter-node interconnects are increasing in speed. The first MPP I used seriously, a Hitachi SR2201 installed in 1997, had a per node performance of 300MFLOPS, and an interconnect peak bandwidth of 300MB/s, so one byte of interconnect bandwidth per FP operation.

A current node would probably have two processors, each of ten cores running at 2.5GHz and capable of eight FP ops per clock cycle. With QDR Infiniband 4X this is one byte of interconnect bandwidth per 100 FP operations.

For some algorithms this is still so much more than is needed that one need not care. For others, careful division of one's code to reflect the different performance of intra-node and inter-node transfers can be beneficial (if time-consuming). One approach is to use OpenMP within a node, and MPI between nodes, often called 'hybrid' coding. Another is to attempt to get topology information into the MPI system.

Hybrid MPI/OpenMP

The early MPI and OpenMP standards ignored each other's existence, but this changed in MPI 2.0.

If mixing MPI with OpenMP (or anything else which uses threads), one should replace the `MPI_Init` call with

```
C: int MPI_Init_thread(int *argc, char **argv, int required,
                       int *provided)
Fortran: call mpi_init_thread(required, provided, ierr)
```

(the Fortran arguments are all integers). The two new integer parameters can take one of four values:

`MPI_THREAD_SINGLE`: no threading
`MPI_THREAD_FUNNELED`: only the master thread makes MPI calls
`MPI_THREAD_SERIALIZED`: multiple threads do not call the MPI library simultaneously
`MPI_THREAD_MULTIPLE`: no restrictions

One requests a level in the `required` argument, and the `provided` argument returns what is actually being provided. It may be less than what was requested: there is no requirement for an implementation to support more than `MPI_THREAD_SINGLE`.

A call to `MPI_Init` is equivalent to calling `MPI_Init_thread` with `required` set to `MPI_THREAD_SINGLE`.

Hybrid MPI/OpenMP continued

`MPI_Init_thread` should be called once on every process, not on every thread.

The thread which calls it becomes the master thread of that process from the point of view of the MPI library.

The values of the parameters are guaranteed to be ordered as above, so
`if (*provided<required) MPI_Abort(MPI_COMM_WORLD,1);`
is sensible, if terse.

Two information routines are provided relevant to thread support. From MPI 3.1 these are callable from any thread at any time, regardless of the current thread support level for other MPI calls. They are:

```
int MPI_Is_thread_main(int *flag)
int MPI_Query_thread(int *provided)
```

(in Fortran `flag` is of type logical).

These mean that any thread, perhaps existing as part of a library linked into an MPI code, can always find out whether it is the master thread from the MPI perspective (i.e. whether it initialised MPI for that process), and what level of thread support exists. The value returned by `MPI_Query_thread` will always be the same that `MPI_Init_thread` returned.

Zero Copy

'Zero Copy' is a term applied to all sorts of I/O (disk, network, MPI), which refers to the idea that the data go from where they are, to where they are wanted, without being copied through intermediate buffers by the CPU. It is a Good Thing, being fast, low-latency and low-overhead.

So how likely are we to achieve it?

Not at all if we use `MPI_Bsend`, but are there better ways?

Zero Copy: It Might Work

Disk controllers, network cards and MPI interconnect controllers can all write data to memory (or read it) without going via the CPU – so called DMA – Direct Memory Access.

But DMA is a little too direct – it goes to physical addresses, not virtual. And it happens at an unspecified time, so the physical to virtual address mapping needs to be frozen until it has happened.

The simple way of dealing with a DMA device is for the kernel to arrange a buffer into which the DMA transfers occur, and then for the kernel to copy the data to/from the user program as appropriate. This is still much better than non-DMA, but it is not zero copy.

(The alternative to DMA is for the CPU to copy data into the controller's own memory. This is bound to be slower than the computer's memory, not least because it is accessed via some 'slow' PCIe bus (slow compared with the memory bus). So having the CPU do a quick copy into its fast memory, and then the slow device do a slow copy from there without involving the CPU, is an improvement.)

Zero Copy: Shared Memory

Even MPI transfers of data between processes running on the same node might not be zero copy. Shared memory exists: an area of memory to which both processes can read and write. But, in general, the 'shared' attribute needs to be specified as the memory is allocated, and needs to be on the granularity of pages (so start addresses on 4KB boundaries for Intel). Also it is generally not possible to have a region which is shared for reading, but writable by the original owner only.

So the expectation is that an MPI library will set up some shared memory as buffer space for transfers between processes, and then a send copies to the shared buffer, and receive copies from it.

OpenMP is better: then it is known when a variable is created whether it should be shared or not. With MPI any variable may be used in a send or receive call, so unless everything is marked as shared life is difficult. If everything is marked as shared, any process can overwrite memory in any other MPI process, and this will lead to debugging nightmares.

Omissions

This course was intended as a brief introduction to MPI. It has covered everything that many MPI programs use, and enough to write real MPI programs. There is plenty of scope for further reading if one wishes, on subjects such as use of communicators, user-defined data types, and one-sided communications.

However, there is often nothing wrong in using a simple subset of MPI. Simple may mean comprehensible, less buggy, and no slower than something more sophisticated.

One is encouraged to read further. The MPI standard can be found at `http://www.mpi-forum.org/docs/`.

Nothing much has been said about debugging either. Debugging is best done interactively, and large computers often do not provide for interactive use. With luck they will provide some form of MPI-compatible debugger, and notes on how to use it.

A late update for MPI 4.0 (2021). One significant change is the introduction of routines to deal with counts and offsets which do not fit in a standard integer. In C these suffix `_c` to the corresponding standard function, and use types of `MPI_Count` for counts and general offsets, `MPI_Offset` for file offsets, and `MPI_Aint` for memory offsets. For the `mpi_f08` interface functions are overloaded, and for earlier Fortran interfaces this extension is not available.

MPI 4 also adds a new way of initialising and finalising MPI (MPI sessions), removing the restriction that this can be done only once per process.

Further Work

This short section is intended to be neither prescriptive nor limiting. MPI is best learnt by practice, and this section is intended to provide a small amount of inspiration for alternatives to thumb-twiddling.

Hello, World

Can you successfully compile and run an MPI program in C (and Fortran)?

Does `mpiexec`[1] work with non-MPI executables such as `/bin/echo` or `hostname`? If it does work with `hostname` and you can submit to a cluster of multiple nodes, this might tell you on which node(s) your code ran.

It might even be possible to use MPI run to run multiple copies of a shell script in a helpful manner for those occasions when one wishes to run large number of a serial job. With OpenMPI, one can try things like:

```
$ mpiexec -n 2 /bin/bash -c 'echo I am rank $OMPI_COMM_WORLD_RANK of $OMPI_COMM_WORLD_SIZE'
I am rank 0 of 2
I am rank 1 of 2
```

Other MPIs are unlikely to set environment variables which begin with 'OMPI', but trying

```
$ mpiexec -n 1 env
```

might be a quicker way of finding out if they do set any environment variables than reading the documentation!

Does an MPI executable run on its own, without being launched by `mpiexec`?

Try the more complicated 'hello' program on page 51. Will `mpiexec` allow you to specify fewer / more cores than your computer has? Is the order of the output repeatable? Can the output be redirected with the shell's standard redirection (>)? What happens if you omit the `MPI_Finalize` call?

Note that most of the answers to the above are not specified in the MPI Standard – different implementations are allowed to behave differently.

If you have a cluster of computers sharing the same filesystem and with MPI installed on all, and passwordless ssh access between them, you may find that something like

[1] For '`mpiexec -n`' read '`mpirun -np`' if necessary throughout

```
$ mpiexec -H pc1,pc2 -n 4 hostname -s
pc1
pc1
pc2
pc2
```

works. (People in TCM will find that running MPI between its workstations in this fashion is strongly discouraged, and any attempt to do it with 'real' MPI code will result in the error message

```
At least one pair of MPI processes are unable to reach each other for
MPI communications.  This means that no Open MPI device has indicated
that it can be used to communicate between these processes.
```

So don't do that there then.)

Simple Quadrature

Try the simple quadrature example from slide 58 (or the Fortran version on the following slide). It does not take long to run, so perhaps increase the number of integration steps, but be careful not to overflow a 32-bit integer when doing so.

If one attempts to time this with something like

```
$ time mpiexec -n 2 ./a.out
```

what CPU time is actually reported? That for all processes, that for a single process, or that for just the `mpiexec` command, ignoring all of the processes it launches? (For those who use OpenMP, how does this compare to OpenMP?)

What about

```
$ mpiexec -n 2 time ./a.out
```

Does `mpiexec` even work when a non-MPI program is placed between it and the MPI program it is trying to launch?

(Users of `bash` should note that `time` is a shell built-in command, but also a command in `/usr/bin`. If `time` is invoked directly from `bash`, the built-in is used; if it is called by another process, such as `mpiexec`, the `/usr/bin` version is used. The default formatting of their output differs.)

Quadrature

Fortran

An example of using Fortran to call the C GSL routine is available on this course's website. Fortran's interface definitions have always seemed to me to be verbose at the best of times, and dealing with a C function which has function pointers and structures as arguments is not the best of times.

Taking the serial code and adding MPI is straightforward though, and the GSL module is not touched by this process.

There is one point to notice. Although the MPI datatypes `MPI_DOUBLE` (C's `double`) and `MPI_DOUBLE_PRECISION` (Fortran's `real(kind(1d0))`) are almost certainly identical, this is not guaranteed. Mix them, and your code will almost certainly run correctly, but it will contain a hidden bug which may bite on some future system on which these types are not the same. So the example code has

```
use, intrinsic :: iso_c_binding

real (kind(1d0)) :: total
real (c_double) :: integral

...

call MPI_Reduce(real(integral,kind(1d0)),total,1,MPI_DOUBLE_PRECISION, &
     MPI_SUM,0,MPI_COMM_WORLD)
```

There are other possible correct approaches, such as

```
use, intrinsic :: iso_c_binding

real (kind(1d0)) :: total,subtotal
real (c_double) :: integral

...

subtotal=integral
call MPI_Reduce(subtotal,total,1,MPI_DOUBLE_PRECISION, &
     MPI_SUM,0,MPI_COMM_WORLD)
```

or even

```
use, intrinsic :: iso_c_binding
```

```
real (c_double) :: total,integral

...

call MPI_Reduce(integral,total,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD)
```

but deliberately calling `MPI_Reduce` from Fortran with C datatypes like this seems stylistically flawed and likely to cause confusion. Whatever is done, the two arguments to `MPI_Reduce` must be of the same type, and that type must be correctly specified in the call.

$\pi/4$

One might reasonably wish to perform quite a large number of iterations in this program for calculating $\pi/4$, as the iterations are fast and convergence is slow. Can you write something which will run with more than 2^{32} iterations, remembering that the total iteration count will no longer fit into a default 32-bit integer?

In C and C++ one can use the `MPI_LONG` datatype in MPI calls, and this is probably 64 bits. In Fortran there is no simple MPI call for sending non-default integers. One solution is to remember that a double precision variable can reliably store integers up to about 2^{53}, so one can convert Fortran's longer integers to doubles before sending them without losing accuracy provided that they are less than 2^{53}.

```
integer(selected_int_kind(12)) :: count
```

in Fortran gives an integer capable of storing integers in the range $\pm 10^{12}$. This cannot be a 32 bit integer, so will probably be a 64 bit one.

Do your results actually keep improving with larger iteration counts, or do you suffer from a random number source with a short periodicity?

Broadcast, Scatter and Gather

Note that the gathering example given is guaranteed to write the partial sums in rank order, rather than in random order as the unmodified code would.

Consider the broadcast example for calculating $\pi/4$. It isn't very good. If run on four processes with an argument of 10, count will be calculated as 2, broadcast to all processes, and just eight iterations will occur.

One could calculate on the rank zero process an array of the required counts for the other processes, then scatter this array. There are many ways of doing this correctly, and many more of doing it incorrectly, so, as a check one could perform an `MPI_Reduce` on the scattered counts, reducing by summation, and check that the sum returned is the original count. Try modifying the example to do this.

Mandelbrot Set

The area of the Mandelbrot Set is unknown – there are merely numeric approximations to it, and some analytic upper and lower bounds. Can you modify the code so that each rank calculates the area of Mandelbrot set that it has been asked to calculate, and then sum these with an `MPI_Reduce`?

(The accepted answer is in the region of $1.506\,592$. We would expect a slight bias to overestimating, for the mathematical definition of z_∞ being finite is stricter than our definition of $|z|$ not exceeding two after 320 iterations.)

For the master-slave version, consider letting each slave keep count of the number of rows it has calculated, and print this, together with its rank, immediately before exiting. Are they all the same? Are they all the same if the resolution is not divisible by the number of slaves? Does the algorithm adapt well when the resolution is not divisible by the number of slaves?

Can you modify the Mandelbrot set generator to make it easier to specify a region of interest without recompiling? If exploring the set, one may find that setting the bottom left corner to $-0.75 + 0.15i$ with a range of about 0.015, or the bottom left at $-0.42 + 0.57i$ with a range of about 0.12, produces something pretty. One may well wish to improve on the colour map used. As one zooms in further, it can be useful to increase the maximum number of iterations. However, eventually the precision of double precision arithmetic becomes an issue, and the set stops appearing fractal but becomes smoothed out. (If tempted to print something, try to avoid printing large areas of dark colour, as toner is not free…)

Laplace

Fortran arguably handles multi-dimensional arrays more neatly than C. However, for efficiency one needs to reverse the order of the indices, for, in Fortran, `a(i+1,j)` is stored next to `a(i,j)`, whereas in C `a[i][j+1]` follows `a[i][j]`. For the benefit of those who believe that Fortran can't do pointers, a Fortran version of this code might look like:

```
program poisson
  real (kind(1d0)),pointer,dimension(:,:) :: g1,g2,p
  integer :: i,j,k,img, ht, n, niter
  integer, parameter :: SIZE=400

  ht=SIZE+2
  niter=40000

  allocate (g1(SIZE,ht),g2(SIZE,ht))

  g1=0.0
  g1(SIZE/8+1:7*SIZE/8,1)=1.0

! Top
  g2(:,1)=g1(:,1)
! Bottom
  g2(:,ht)=g1(:,ht)

  do n=1,niter
```

```
    do j=2,ht-1
! Left
      g2(1,j)=(g1(1,j-1)+g1(1,j+1)+g1(2,j))/3
! Body
      do i=2,SIZE-1
        g2(i,j)=0.25*(g1(i,j-1)+g1(i,j+1)+g1(i-1,j)+g1(i+1,j))
      enddo
! Right
      g2(SIZE,j)=(g1(SIZE,j-1)+g1(SIZE,j+1)+g1(SIZE-1,j))/3
    enddo

    p=>g1
    g1=>g2
    g2=>p

  enddo

  img=10
  open(unit=img,file='poisson.pgm')
  write(img,'(a2)')"P2"
  write(img,'(I0,x,I0,x,I0)')SIZE,SIZE,255
  do j=2,ht-1
    do i=1,SIZE
      write(img,'(I0)')int(255*g1(i,j))
    enddo
  enddo
end
```

(Note the use of `g1=0.0` to set all the array elements, and `g2(:,1)` to address a whole row at once. Note also the explicit format statement on the line
`write(img,'(a2)')"P2"`
without which Fortran is likely to add a leading space, which would mean that the resulting file is not a valid pgm file. For the other write statements the addition of whitespace will not cause a problem, but the formating has been adjusted to be identical to that expected from the C version, so it may be possible to compare the output files using `diff` or similar trivially.)

Try testing the Poisson solver, verifying that the serial and parallel codes give the same answers, regardless of the number of processes used. For debugging, it can be useful to use very small grid sizes, such as 8×8. Consider adding a convergence check, perhaps the greatest change in value of a single element being less than 0.01.

It is always good practice to test a numerical code with a problem for which the solution is known. It can readily be shown that the function

$$F(x, y) = A(\exp(ky) - \exp(-ky)) \cos(kx)$$

satisfies Laplace's equation, and is zero at $y = 0$. Also a constant trivially satisfies Laplace's equation.

Choosing $k = 4\pi/400$, $A = 0.5/(\exp(4\pi) - \exp(-4\pi))$ and a constant of 0.5 gives

$$F(x, y) = \frac{(\exp(0.01\pi y) - \exp(-0.01\pi y))\cos(0.01\pi x)}{2(\exp(4\pi) - \exp(-4\pi))} + 0.5$$

This has

$$F(x, y = 0) = 0.5$$

and

$$F(x, y = 400) = 0.5 + 0.5\cos(0.01\pi x)$$

Can you confirm that the results of this solver match this solution? Note that this analytic solution has periodic boundary conditions in x, that one would not expect to match it exactly with a grid of finite size, and that increasing the grid size will increase the number of iterations before convergence is obtained.

Master Slave revisited

An enthusiastic master could send a slave his next task using `MPI_Isend` before receiving any indication that the current task is completed. An enthusiastic slave could then return his results using an immediate send too, so that he could receive his next task and start work before the master has had time to receive the previous results.

Can you re-write the master–slave Mandelbrot code so that the next task of work is sent in this fashion? If doing the same for the slave, remember that one cannot write to the buffer sent by `MPI_Isend` until one has confirmation that the send if complete. So perhaps use a `Bsend` (having first attached a suitably-sized buffer) instead, or perhaps carefully alternate between two buffers (results currently being calculated, and results previously sent), remembering that swapping pointers is quicker than copying the contents of arrays.

What does your version of MPI do if you attempt to call `MPI_Bsend` without having attached any buffer? without having attached a buffer of adequate size?

Miscellaneous

Mistakes

For what values of `SIZE` does the deadlock code complete? If it fails to complete, does it sit using no CPU time, or using CPU time?

Does the version of MPI you are using detect mismatched datatypes on a send/receive pair? Does it detect receive buffers which are too small?

Have you noticed that `MPI_Gather` looks a bit like

```
MPI_Send(sendbuf,sendcount,sendtype,root,...)
if (rank==root)
  for(i=0;i<nproc;i++)
    MPI_Recv(recvbuff+i*recvcount,recvcount,recvtype,i,...)
```

The 'real' MPI_Gather should be happy to receive data in any order, amongst other differences.

Suppose the amount of data (`sendcount`) differed on the different ranks, so that one wished to write something like

```
MPI_Send(sendbuf,sendcount,sendtype,root,...)
if (rank==root)
  for(i=0;i<nproc;i++)
    MPI_Recv(recvbuff+offsets[i],recvcounts[i],recvtype,i,...)
```

Even that is no excuse for DIY collectivism, for that is the functionality of `MPI_Gatherv`.

Try rewriting the Mandelbrot set generator which used `MPI_Gather` so that it uses `MPI_Gatherv`, and thus works if the number of rows is not the same on all processes. (To do this you will have to use the poor load balancing solution of having rank 0 calculate the first image height / nprocs rows, then rank 1 the next image height / nprocs rows, save that unless the image height is divisible by the number of processes, those totals should vary by one in some cases.)

(Those who are proficient in C might wonder about passing the image to libpng so that it is written out in a sensible compressed format automatically. To call libpng you will need the whole of the image data to be on a single process.)

Performance

Try modifiying the ping-pong code so that rather than sending from between rank 0 and rank 1, it sends between all pairs of even and odd ranks. Does the performance change as the number of processes used changes?

If you have access to a heterogeneous system, try measuring the different intranode and internode performance. You may need to read some documentation on how to launch processes in odd fashions, or you may choose to vary which pairs one tries to send between. (E.g. if you have access to a machine with 16 core nodes, the chances are that submitting 32 process jobs and sending between processes rank 0 and 1, and then 0 and 16, will show inter node and intra node behaviour in some order.)

If you are not tied to a queueing system, then one often has more flexibilty, and commands such as

```
s8:~$ mpiexec -H s7,s8 -N 1 -n 2 ./a.out
```

can be interesting. (Run on hosts s7 and s8, with no more than one process on each, total number of processes two.)

How do the latency and bandwidth change if you replace `MPI_Send` by `MPI_Bsend`? (Remember to attach a suitably large MPI buffer first!) Does latency or bandwidth change if you replace `MPI_Send` by `MPI_Ssend`?

Progress

Your MPI implementation almost certainly shows weak progress. At what transfer size in the example given does this become clear? Can you modify this value? If using OpenMPI, you may find the following instructive:

```
$ mpiexec -n 2 ./a.out
Process rank 0
Process rank 1
Rank 0 reaching barrier at time 10
Rank 1 reaching barrier at time 10
```

```
$ OMPI_MCA_btl_sm_eager_limit=6144 mpiexec -n 2 ./a.out
Process rank 0
Process rank 1
Rank 1 reaching barrier at time 0
Rank 0 reaching barrier at time 10
```

The default value for `btl_sm_eager_limit` is 4096, which is insufficient for a transfer of precisely that size. Does changing this parameter affect pingpong performance for packets of around this size?

Can you write a code which runs to completion if the message size is smaller than the eager limit, and deadlocks (never completing) if it is larger? This could lead to interesting bugs, where code runs on small test cases, then fails on larger jobs...

Does your MPI implimentation show strong or weak progress? If weak, at what size message does it change from isends completing when the call is first made, to needing further entry into the MPI library to progress them?

Advanced Features

Communicators

Write a code in which the root process randomly initialises an 4×4 matrix. It then scatters the four 2×2 corners to itself and ranks 1 to 3. Communicators are then set up using `MPI_Comm_split` so that `MPI_Reduce` can be used to sum the rows and columns of the scattered matrix, and the sums printed from whichever processes seem convenient. Print the sums also from the root process, which has the full matrix, without using MPI, to demonstrate that the answers are the same!

Fortran users may wish to be reminded of Fortran's own reduction operations, such as
`sum(matrix(:,i))`

MPI I/O

The Mandelbrot set example is an obvious example of something waiting to be converted to MPI I/O. Indeed, Fortran users might consider writing binary files without recourse to F2003 features. (Or, 2003 being well over a decade ago, Fortran users might feel that F2003 is fair game.)

Rewrite the Mandelbrot code so that the master writes the file's header, broadcasts its size to all nodes, and all nodes write to the file using `mpi_file_write_at`.

Rewrite the Mandelbrot code so that each process writes one row every `nproc` (i.e. interleaved for half-reasonable load balancing) and define an MPI Datatype so that each process appears to be writing to the file without gaps. Some useful code fragments would be:

```
  integer :: row_type,tmp_type
  integer(kind=MPI_OFFSET_KIND) :: disp

  [...]

  if (irank.eq.0) then
    write(*,*)'Please input resolution'
    read(*,*) res
  end if

  call mpi_bcast(res,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)

  call mpi_file_open(MPI_COMM_WORLD,"mandelf.pam",   &
    MPI_MODE_WRONLY+MPI_MODE_CREATE,MPI_INFO_NULL,mpi_img,ierr)
! Ensure any existing file is truncated
  call mpi_file_set_size(mpi_img,int(0,MPI_OFFSET_KIND),ierr)
  call mpi_barrier(MPI_COMM_WORLD,ierr)

  if (irank.eq.0) then
    nl=new_line('x')
    write(str,'(''P6'',a,i5,i5,a,''255'',a)')nl,res,res,nl,nl
    hdr=len(trim(str))
    call mpi_file_write(mpi_img,str,hdr,MPI_BYTE,stat,ierr)
endif

call mpi_bcast(hdr,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)

call mpi_type_vector(1,3*res,3*res,MPI_CHARACTER,tmp_type,ierr)
call mpi_type_create_resized(tmp_type,0_MPI_ADDRESS_KIND,  &
                int(3*res*nproc,MPI_ADDRESS_KIND),row_type,ierr)
call mpi_type_commit(row_type,ierr)
disp=hdr+irank*3*res
write(*,*)'rank',irank,'disp',disp
call MPI_File_set_view(mpi_img,disp,MPI_CHARACTER,row_type, &
                    "native",MPI_INFO_NULL,ierr)

[...]

do i=irank,res,nproc
  y=-1.5d0+i*3d0/(res-1)
```

```
  do j=0,res-1
    [...]
  enddo
  call mpi_file_write(mpi_img,line,3*res,MPI_CHARACTER,stat,ierr)
enddo

call mpi_file_close(mpi_img,ierr)
```

The result is quite impressive in that each process appears to be writing a continuous stream to the file via the `mpi_file_write` statements, but what actually appears in the file from each process is not contiguous, but has large gaps. One might question the effort of setting this up...

This had each process writing a full row, then skipping `nproc-1` rows, etc. An alternative approach would be to write an element in a row, then skip `nproc-1` elements. Try coding this. It should be possible to do so with a single call to `mpi_file_write` which writes all elements of a row that the calling process will calculate. (You may assume that the resolution is divisible by the number of processes, and abort if not.)

Beware that this fine-grained interleaving may be very inefficient on remote filing systems. I recorded this example as taking under 0.5s to run on a local disk, and about 900s on the same machine on a remote disk, with an output filesize of just 120KB!

Hardware: NUMA

If you can find exclusive access to a NUMA machine (i.e. almost any current dual-socket machine) it can be instructive to run a few benchmarks on it. Precisely how one controls the placement of processes on processors will vary. Here we rely on the fact that, with OpenMPI on a single node, the MPI processes are children of the `mpiexec` command. So commands such as

```
taskset 3 mpiexec -n 2 ./a.out
```

make sense as the processes will inherit the mask of their parent. This runs on cores 0 & 1, as the mask, 3, given in hex, is 11 expressed in binary. Rerunning our pingpong code, the results are approximately

bytes	MB/s	MB/s
8	8	22
16	17	44
32	34	88
64	68	175
128	110	305
256	200	550
512	341	960
1024	555	1540
2048	1030	2310
4096	1230	3000
8192	1670	4120
16384	2040	4200
32768	2140	4660
65536	2500	5670
131072	3050	6650
262144	3360	6500
524288	3450	6700
1MB	3300	6800
2MB	2800	5500
4MB	3100	4900
8MB	2620	3500
16MB	2520	3470
32MB	2540	3470

The second column is for taskset masks of 11, 21, 41 and 81, whereas the third column is for 3, 5 and 9. The computer has two quad core processors, each with 10MB of last level cache shared between the cores. In all cases core number zero was used, and in the third column one of the other three cores in the same socket was used. With a shared last level cache, latencies were around 0.35μs with a bandwidth of around 6.8GB/s whilst within that cache. When the MPI transfer was between different sockets the latency is increased almost threefold, and the bandwidth reduced almost twofold.

Even without access to a dual socket machine, we can also set the mask to 1, meaning run both MPI processes on the same core. This is a disaster. Because the processes spin wait when waiting to receive, the latency jumps from 1μs to 12ms, and the bandwidth for large transfers falls from 3,000MB/s to 4MB/s. To run the pingpong code in a sensible time it is necessary to reduce the value of `rep` by at least a factor of 1000.

A program which is likely to perform in the opposite fashion to pingpong is the Streams benchmark (developed by John McCalpin obtainable from http://www.cs.virginia.edu/stream/). You will find an unaltered copy of the Fortran MPI version of this benchmark on this course's site, together with a helper timer routine. (Nearly unaltered – the default array size has been increased by a factor of ten.)

This can be compiled using a command similar to

```
$ mpifort stream_mpi.f mysecond.f90
```

This time, considering runs using four processes, we find that

```
$ taskset f mpiexec -n 4 ./a.out
Function     Rate (MB/s)  Avg time   Min time  Max time
Copy:           25447.3   0.050375     0.050300     0.050500
Scale:          22456.1   0.057088     0.057000     0.057200
Add:            26193.7   0.073525     0.073300     0.073700
Triad:          26301.4   0.073237     0.073000     0.073600
```

(and the same for `taskset f0`), but

```
$ taskset 33 mpiexec -n 4 ./a.out
Function     Rate (MB/s)  Avg time   Min time  Max time
Copy:           29836.8   0.043038     0.042900     0.043200
Scale:          25858.6   0.049600     0.049500     0.049900
Add:            33160.6   0.058125     0.057900     0.058700
Triad:          33449.5   0.057475     0.057400     0.057600
```

Not a huge difference, but on the triad test over 25% faster when the code is spread across two sockets than when it is confined to a single socket. On some machines one might expect a difference of rather more.

BEWARE! If using `taskset`, not that Linux does not specify how CPU core numbers map to sockets. One can find out:

```
~$ grep -E '^processor|^physical' /proc/cpuinfo
processor       : 0
physical id     : 0
processor       : 1
physical id     : 0
processor       : 2
physical id     : 0
processor       : 3
physical id     : 0
processor       : 4
physical id     : 1
processor       : 5
physical id     : 1
processor       : 6
physical id     : 1
processor       : 7
physical id     : 1
```

where physical id refers to socket, so that is the system described. However, equally valid is

```
$ grep -E '^processor|^physical' /proc/cpuinfo
processor       : 0
physical id     : 0
processor       : 1
physical id     : 1
processor       : 2
physical id     : 0
processor       : 3
physical id     : 1
processor       : 4
physical id     : 0
processor       : 5
physical id     : 1
processor       : 6
physical id     : 0
processor       : 7
physical id     : 1
```

on which `taskset f` means use both sockets (and triad says 19,140 MB/s), whereas now `taskset 55` means use one socket (processors 0, 2, 4 and 6), and triad then says (10,300 MB/s). This second machine shows an 85% difference in these triad results, and could be said to be more strongly NUMA than the first.

A similar command to `taskset` is `numactl`.

Total memory bandwidth can also be a problem. As the number of processes for the streams benchmark is varied from one to eight, the first machine produces triad scores of 8,290, 16,100, 22,430, 26,000, 33,100, 39,700, 46,200 and 53,000 MB/s. This is unusually good, even if the last figure is merely $6.4\times$ the first, and no communications are involved. The first machine, which is rather older, produces scores of 7,450, 9,840, 10,340, 10,200, 12,800, 15,340, 17,770 and 20,240 MB/s. So the last figure is just $2.7\times$ the first, and though it started just 10% behind its newer rival, it finished more than a factor of two behind. It would also seem that the default behaviour of MPI on both machines was to run four process jobs on just a single socket, rather than splitting them between two.

It can even be advantageous on some machines not to use half the cores on each socket, as this can give similar total memory bandwidths to using all of them, and less parallelisation overheads.

Timers

Timing code is very useful, but there are a couple of issues to watch out for. Firstly one generally wants wall-clock time, not CPU time, and attempting to time code on non-idle nodes is pretty pointless. (Even if one has exclusive access to a core, the memory within a node is shared, and memory activity from other processes can disrupt timings significantly.)

Secondly, the resolution of the timer may not be as high as one had hoped. It might return seconds in a double precision variable, but it is unlikely that it is doing so to full precision. Try to write something to test the timer functions available to you, in order to return their resolution if it is worse than a microsecond. (Determining resolution that is significantly better than one microsecond becomes quite hard due to function call overheads etc.)

Fortran users may wish to consider `system_clock` and `MPI_Wtime`. C users may wish to consider `gettimeofday`, `time` and `MPI_Wtime`. Fortran users may wish to consider that `system_clock` will be provided by their compiler's Fortran runtime library, so the resolution may differ between different compilers. A quick test of compilers currently installed in TCM shows that Intel's Fortran compiler gives a resolution ten times better than NAG's, Sun's or Gnu's (the last three all being 1ms). C compilers will almost certainly all use the same libc library, so one would not expect different C compilers to differ.

Reference

MPI Datatypes

C: `MPI_CBOOL MPI_CHAR MPI_SHORT MPI_INT MPI_LONG MPI_LONG_LONG`
`MPI_UNSIGNED_CHAR MPI_UNSIGNED_SHORT MPI_UNSIGNED ...`
`MPI_INT8_T MPI_INT16_T MPI_INT32_T MPI_INT64_T` also `MPI_UINT8_T` etc.
`MPI_FLOAT MPI_DOUBLE`
`MPI_C_COMPLEX MPI_C_DOUBLE_COMPLEX`

C++: As C, and add `MPI_CXX_BOOL`
`MPI_CXX_FLOAT_COMPLEX MPI_CXX_DOUBLE_COMPLEX`

Fortran: `MPI_LOGICAL MPI_CHARACTER MPI_INTEGER`
`MPI_REAL MPI_DOUBLE_PRECISION`
`MPI_COMPLEX MPI_DOUBLE_COMPLEX`

`MPI_C_FLOAT_COMPLEX` is a synonym for `MPI_C_COMPLEX`.

Fortran `mpi` vs `mpi_f08`

Those using the `mpi` module will find that many things which were types are plain integers in the older Fortran `mpi` module. In particular:

```
                          use mpi                          use mpi_f08
MPI_Aint        integer(kind=mpi_address_kind)    integer(kind=mpi_address_kind)
MPI_Comm                  integer                          type(mpi_comm)
MPI_Datatype              integer                       type(mpi_datatype)
MPI_File                  integer                          type(mpi_file)
MPI_Info                  integer                          type(mpi_info)
MPI_Offset      integer(kind=mpi_offset_kind)     integer(kind=mpi_offset_kind)
MPI_Request               integer                       type(mpi_request)
MPI_Status       integer st(mpi_status_size)           type(mpi_status)
```

This makes conversion from the old Fortran 90 module to the new less easy than one might hope, although the stronger type checking in the new module may catch more bugs. Note that the MPI 'constants' such as `MPI_COMM_WORLD` will be of the appropriate type for the module in use, and need not be changed. If `st` is an MPI status, then `mpi` has `st(MPI_TAG)` and `st(MPI_SOURCE)` whereas `mpi_f08` has `st%MPI_TAG` and `st%MPI_SOURCE`.

Summary of MPI-I/O commands in this booklet

```
int MPI_File_close(MPI_File *fh)
int MPI_File_delete(char* name, MPI_Info info)
int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,
              MPI_Offset *disp)
int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)
int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)
int MPI_File_iread(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Request *req)
int MPI_File_iread_shared(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Request *req)
int MPI_File_iwrite(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Request *req)
int MPI_File_iwrite_shared(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Request *req)
int MPI_File_open(MPI_Comm comm, const char *name, int amode,
                  MPI_Info info, MPI_File *fh)
int MPI_File_read(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Status *status)
int MPI_File_read_all(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Status *status)
int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
     MPI_Datatype datatype, MPI_Status *status)
int MPI_File_read_ordered(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Status *status)
int MPI_File_read_shared(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Status *status)
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)
int MPI_File_set_size(MPI_File fh, MPI_Offset offset)
int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
  MPI_Datatype ftype, const char *datarep, MPI_Info info)
int MPI_File_write(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Status *status)
int MPI_File_write_all(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Status *status)
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
     MPI_Datatype datatype, MPI_Status *status)
```

```
int MPI_File_write_ordered(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Status *status)
int MPI_File_write_shared(MPI_File fh, void *buf, int count,
     MPI_Datatype datatype, MPI_Status *status)
MPI_Type_commit(MPI_Datatype type)
MPI_Type_create_resized(MPI_Datatype oldtype,MPI_Aint lb,
  MPI_Aint extent, MPI_Datatype *newtype)
MPI_Type_free(MPI_Datatype *type)
MPI_Type_size(MPI_Datatype type, int *size)
MPI_Type_vector(int count, int blocklength, int stride,
  MPI_Datatype oldtype, MPI_Datatype *newtype)
```

In Fortran an `MPI_Offset` is an integer of kind `MPI_OFFSET_KIND`, `MPI_Aint` an integer of kind `MPI_ADDRESS_KIND`, and an `MPI_Datatype` and `MPI_File` is a default integer. The Fortran notes on the following page also apply.

Summary of MPI commands in this booklet

```
int MPI_Abort(MPI_Comm comm, int error)
int MPI_Allgather(void *sendbuff, int sendcount, MPI_Datatype sendtype,
                  void *recvbuff, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
int MPI_Allreduce(void *sendbuff, void *recvbuff, int count,
                MPI_Datatype type, MPI_Op op, MPI_Comm comm)
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm);
int MPI_Barrier(MPI_Comm comm)
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm)
int MPI_Bsend(void *buff,int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm)
int MPI_Buffer_attach(void *buff, int len)
int MPI_Buffer_detach(void *buff, int *len) /* NB really void **buff */
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errh)
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_split(MPI_Comm comm, int colour, int key, MPI_Comm *newcomm)
int MPI_Init(int *argc, char ***argv)
int MPI_Init_thread(int *argc, char **argv, int required, int *provided)
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *st)
int MPI_Irecv(void *buff, int count, MPI_Datatype type, int source, int tag,
              MPI_Comm comm, MPI_Request *req, MPI_Status *st)
int MPI_Is_thread_main(int *flag)
int MPI_Isend(void *buff,int count, MPI_Datatype type, int dest,
              int tag, MPI_Comm comm, MPI_Request *req)
int MPI_Finalize(void)
int MPI_Gather(void *sendbuff, int sendcount, MPI_Datatype sendtype,
               void *recvbuff, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm)
int MPI_Gatherv(void *sendbuff, int sendcount, MPI_Datatype sendtype,
                void *recvbuff, int recvcounts[nproc], int offset[nproc],
                MPI_Datatype recvtype, int root, MPI_Comm comm)
int MPI_Get_count(MPI_Status *st, MPI_Datatype type, int *count)
int MPI_Get_version(int *major, int *minor)
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *st)
int MPI_Query_thread(int *provided)
int MPI_Recv(void *buff, int count, MPI_Datatype type, int source,
```

```
              int tag, MPI_Comm comm, MPI_Status *st)
int MPI_Reduce(void *sendbuff, void *recvbuff, int count,
               MPI_Datatype type, MPI_Op op, int root, MPI_Comm comm)
int MPI_Request_free(MPI_Request *req)
int MPI_Scatter(void *sendbuff, int sendcount, MPI_Datatype sendtype,
                void *recvbuff, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm)
int MPI_Scatterv(void *sendbuff, int sendcounts[nproc],
                 int offsets[nproc], MPI_Datatype sendtype,
                 void *recvbuff, int recvcount, MPI_Datatype recvtype,
                 int root, MPI_Comm comm)
int MPI_Send(void *buff,int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm)
int MPI_Ssend(void *buff,int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm)
int MPI_Sendrecv(void *sendbuff, int sendcount, MPI_Datatype sendtype,
                 int dest, int sendtag,
                 void *recvbuff, int recvcount, MPI_Datatype recvtype,
                 int source, int recvtag, MPI_Comm comm, MPI_Status *st)
int MPI_Sendrecv_replace(void *buff, int count, MPI_Datatype datatype,
                         int dest, int sendtag, int source, int recvtag,
                         MPI_Comm comm, MPI_Status *st)
int MPI_Test(MPI_Request *req, int *flag, MPI_Status *st)
int MPI_Wait(MPI_Request *req, MPI_Status *st)
int MPI_Waitall(int n, MPI_Request req[n], MPI_Status st[n])
double MPI_Wtick(void)
double MPI_Wtime(void)
```

In Fortran all integer functions become subroutines with an extra integer argument of the return code as their last argument (optional with `mpi_f08`), and `MPI_Init` and `MPI_Init_thread` lose the `argc` and `argv` arguments. Also the `flag` argument in `MPI_Test`, `MPI_Iprobe` and `MPI_Is_thread_main` is of type `logical`, not `integer`.

In Fortran the `MPI_Status` type is an integer array of length `MPI_STATUS_SIZE`, and an `MPI_Comm` is an integer, as is an `MPI_Request`, `MPI_Op` or an `MPI_Datatype`. (Different in `mpi_f08` – see above.)

In Fortran or C/C++, arguments must not overlap (or be the same). (The send and receive buffers of `MPI_Reduce` cannot be the same, for instance.) One can use the argument `MPI_IN_PLACE` for one buffer in order to perform in place reductions, scatters, gathers and alltoalls. `MPI_IN_PLACE` is generally used for the send buffer on the root process only, but for scatter it is the receive buffer on the root process only.

aliasing of arguments, 123
Amdahl's law, 253, 260, 261
asychronous, 168

bandwidth, 199–201
bandwidth, interconnect, 259
bisectional bandwidth, 256
blocking, 128
broadcast, 89–92
buffered sends, 160

C++, 37
cache
    exclusive, 276
    inclusive, 276
    memory, 265
cache coherency
    broadcast, 272, 273
    directory, 272
    snoopy, 266
cache line, 271
`calloc()`, 73
cc-NUMA, 267
Clos network, 258
collective, 125, 188
collective I/O, 237–239, 246, 247
collectives
    ordering of, 139
compiler optimisation, 149
computer, 20
core, 20, 24, 40
CPU, 40
crossbar, 256

datatype, 62, 99, 194, 304
    user defined, 243–246, 287, 297
deadlock, 171, 184, 185, 295, 297
diameter, 256
distributed memory computer, 255
DMA, 285

eager, 297
eager sends, 172, 206
error accumulation, 82
error handling, 102, 228
errors, 54
ethernet, 27, 200, 201

false sharing, 272, 274
FFT, 2D, 191
file pointer, 218, 228, 233
    shared, 235, 238
file views, 240–242
`free()`, 69, 70

gather, 93–95
ghost, *see* halo
GSL, 78

halo region, 156, 176
hybrid, 281–283
hypercube, 256
HyperTransport, 280

idle wait, 187
images, 148
in place, 100, 101, 191
Infiniband, 27, 33, 280
interconnect quality, 27, 261

Java, 19

Laplace's equation, 142
latency, 33, 199–201, 205
latency, interconnect, 259
Linpack, 273
load balancing, 118
lock step, 56

`malloc()`, 69–74
Mandelbrot Set, 105
master, 63, 90, 129, 130, 132, 137, 138, 295
MESI, 271
MPI 2.2, 65
MPI 3.0, 65

MPI 4.0, 287
MPI groups, 217
MPI standard, 287
MPI_Abort, 56
MPI_ADDRESS_KIND, 209
MPI_Allgather, 97
MPI_Allreduce, 63
MPI_Alltoall, 189–191
MPI_Alltoallv, 191
MPI_ANY_SOURCE, 127, 208
MPI_ANY_TAG, 127
MPI_Barrier, 116, 139
MPI_Bcast, 90, 91, 116
MPI_Bsend, 160–162, 177, 295, 296
MPI_Buffer_attach, 161, 163
MPI_Buffer_dettach, 163
MPI_BYTE, 194
MPI_Comm_free, 216
MPI_Comm_get_attr, 208, 209
MPI_Comm_rank, 55
MPI_COMM_SELF, 209
MPI_Comm_set_errhandler, 102
MPI_Comm_size, 55
MPI_Comm_split, 214, 215
MPI_COMM_WORLD, 54, 55
mpi_f08, 131, 304
MPI_File_delete, 231
MPI_File_get_byte_offset, 241
MPI_File_get_position, 241
MPI_File_get_position_shared, 235
MPI_File_iread, 229
MPI_File_iread_shared, 235
MPI_File_iwrite, 229
MPI_File_iwrite_shared, 235
MPI_File_open, 230
MPI_File_read_all, 239
MPI_File_read_at, 233
MPI_File_read_ordered, 238
MPI_File_read_shared, 235
MPI_File_seek_shared, 235
MPI_File_set_size, 232
MPI_File_set_view, 246
MPI_file_set_view, 240, 241
MPI_File_write_all, 239
MPI_File_write_at, 233
MPI_File_write_ordered, 237, 238
MPI_File_write_shared, 235, 236
MPI_Finalize, 48
MPI_Gather, 94, 100, 122, 123, 295
MPI_Gatherv, 97, 98, 296
MPI_Get_count, 179, 193
MPI_Get_version, 65
MPI_IN_PLACE, 100, 101, 191
MPI_Init, 48, 54, 282
MPI_Init_thread, 48, 80, 282, 283
MPI_Iprobe, 181
MPI_Irecv, 170
MPI_Is_thread_main, 283
MPI_Isend, 160, 177, 295
MPI_OFFSET_KIND, 232, 233
MPI_Probe, 179, 180
MPI_Query_thread, 283
MPI_Reduce, 61, 100, 101
MPI_Request, 165–167
MPI_Request_free, 170
MPI_Request_get_status, 170
MPI_REQUEST_NULL, 165–167
MPI_Scatter, 94, 100
MPI_Scatterv, 97, 98
MPI_Send, 177
MPI_Sendrecv, 174
MPI_Sendrecv_replace, 174
MPI_Ssend, 175, 177
MPI_Status, 127, 130, 131, 180, 181
MPI_STATUS_IGNORE, 198
MPI_SUCCESS, 102
MPI_Test, 169, 229
MPI_Type_commit, 244
MPI_Type_create_resized, 244
MPI_Type_free, 246

MPI_Type_size, 245
MPI_Type_vector, 244
MPI_Wait, 169, 229
MPI_Waitall, 166
MPI_Wtick, 198
MPI_Wtime, 198, 303
MPICH, 67
mpiexec, 41, 49, 50, 64, 92, 288
mpirun, 41, 49, 288
MPP, 12, 255

NFS, 221, 222
NFS v4, 223
node, 20, 40
non-blocking, 207, 259
NUMA, 267–270
numactl, 302

Omni-Path, 280
one-sided communication, 287
OpenMP, 16, 18, 38, 63, 64, 279, 281, 282
OpenMPI, 67
overlap, communication and computation, 202, 207

pam images, 107, 148
parallel computer, 40, 251
parallel efficiency, 25, 27
PCIe, 280
pgm, 148
pingpong, 195, 205, 296
point to point, 125, 126
pointers, 62
Poisson's equation, 142
PostScript, 144
processor, 20
progress, 202–205, 207, 296
python, 19

QPI, 280

rand(), 86, 87
random numbers, 86–88
rank, 55
realloc(), 71
receive buffer, 192, 193
receive, types of, 177
reduction operators, 60, 61
request id, 165, 167, 170
root, 61

scaling, 25, 27, 119, 138, 155, 252–254, 261
    memory, 279
    superlinear, 277
scatter, 93–95
send, types of, 177
shared memory, 38
shared memory processor, 262
shepherd, 64
SIMD, 251
sizeof(), 69, 74
slave, 129, 130, 132, 137, 138, 295
SMP, 12, 262, 267
spin wait, 187
status, 127, 130, 131, 179, 181, 193
Streams, 269, 273
synchronisation, 116, 139
synchronous, 168

tag, 127, 139
taskset, 269, 299
threading, 80, 282
time, 289
timers, 302
timing, 198
Top500, 23–25
topology, 256
transpose, 189, 191
tree, 256

versions of MPI, 65, 67, 287