Introductions to Computing

MJ Rutter
mjr19@cam

Lent 2009

Parallel Programming

The Need for Parallelism

Today it is hard to buy a single-core computer. The ultra-light notebook market is almost the only place they are found. 'Standard' notebooks are dual core, 'standard' desktops quad core, and purchasing eight core computers is not very expensive. In the future the core count will increase further.

Something called SMT or Hyperthreading might also effectively increase the core count by a factor of two. Intel's i7 does this, as did the Pentium 4.

Of course, clustering multiple computers using GBit/s ethernet is also now very cheap, and is another way of producing a single resource with many cores. If a better interconnect is required, Infiniband is relatively inexpensive.

Broken Philosophy

UNIX is not actually very good at letting separate processes communicate. It tries so hard to isolate them from each other, and forgot that sometimes a little communication can help.

In the UNIX world, resources, such as memory, file handles, and other finite things, are associated with particularly processes. When a process exits, for whatever reason, the kernel frees up all of the resources it was using. Nothing should ever escape, so reboots to free lost resources should never be needed.

System V introduced the antithesis of this. SYSV shared memory and semaphores are objects which are owned by no process, but to which multiple processes may attach themselves. They remain even when no process is currently attached, lest some future process should want to use them.

Uncontrolled Leaks

If something using SYSV shared memory crashes, and fails to release that memory, it will never be freed until the machine is next rebooted. Even if it simply forgets, the result is the same.

Unfortunately some implementations of MPI use SYSV semaphores to aid synchronisation. Almost nothing uses SYSV shared memory.

(On Linux, the most certain way of discovering that leaks have occurred is to examine the contents of the three files in `/proc/sysvipc`)

A Better Way

A different standard for dealing with shared memory, called POSIX shm, is saner. It keeps a count of the number of processes currently attached to a given area of shared memory. When the count reaches zero, the memory is freed automatically. However a process exits, the attachment count on any shm areas it was using will be decremented.

It is a pity that the SYSV way ever became popular, but it got there first, in 1983, beating POSIX by a decade.

Parallel Programming with Processes

Conceptually the simplest form of parallel programming is to have multiple independent processes which occassionally communicate in response to explicit instructions to do so.

The standard programming interface for this is MPI (Message Passing Interface), which was designed to be used with either C89 or Fortran77. It is thus usable with C99, C++ and F90/95/2003 too.

MPI consists of a library to be linked against at compile time, an include file defining all sorts of constants to be made available at compiler time, and a special command (usually `mpirun`) to launch the resulting program.

MPI is a nice open standard. For Linux, versions from HP, Intel, MPICH, LAM and OpenMPI exist, and maybe others too. One must use a consistent set of include file, library and mpirun commands. Failure to do so might not produce any warnings, but is unlikely to produce the desired behaviour.

MPI Hello World

```
      program hello
c
      include 'mpif.h'
      integer npe,mype,ierr
c
      call mpi_init(ierr)
      if (ierr.ne.0) stop 'MPI initialisation error'
c
      call mpi_comm_rank(mpi_comm_world, mype, ierr)
      call mpi_comm_size(mpi_comm_world, npe, ierr)
c
      write(*,101) mype,npe
101   format(' Hello F77 parallel world, I am process ',I3,
     &       ' out of ',I3)
c
      call mpi_finalize(ierr)
      end
```

Running MPI

```
pc0:~$ mpifort hello.mpi.f
pc0:~$ mpirun -np 2 ./a.out
 Hello F77 parallel world, I am process   1 out of   2
 Hello F77 parallel world, I am process   0 out of   2
pc0:~$ mpirun -np 4 ./a.out
 Hello F77 parallel world, I am process   1 out of   4
 Hello F77 parallel world, I am process   2 out of   4
 Hello F77 parallel world, I am process   3 out of   4
 Hello F77 parallel world, I am process   0 out of   4
```

Parallel Programming without Processes

Starting (and stopping) a process is an expensive operation. Tables of new virtual to physical address mappings need to be created, along with control blocks describing the process and its file handles etc. On exit all these need to be freed, and probably an entry written to the process accounting file too. the whole start/stop cycle may take the odd ms.

MPI runs with a fixed number of processes to avoid this overhead getting out of control.

There is an alternative: threads.

Threads

A thread is something which can be sceduled like an independent process, but shares everything else with its parent. It is much simpler to create and destroy, for the whole of its address space, and much else besides, is not duplicated. Communication between threads in the same process is also trivial, for they all automatically have access to all of the same memory. Keeping them apart is the harder bit.

The standard model for thread programming on Linux is that of POSIX threads (pthreads). However, people rarely use threads directly, instead using OpenMP to do so.

Note that if a signal is set to a threaded process, all threads receive it. Usually.

OpenMP

OpenMP, which supports Fortran and C, is implemented as a set of comments (pragmas in C) which the compiler can ignore, producing correct serial code, or follow, producing parallel code.

In general, the comments preceed loops, and are instructions to parallelise the following loop, with a list of which variables should be private to each separate version of the loop. The loop counter is automatically considered private, and everything else is shared. The threads are created with separate stacks, where the private variables are, but everything else is common.

What Goes Where

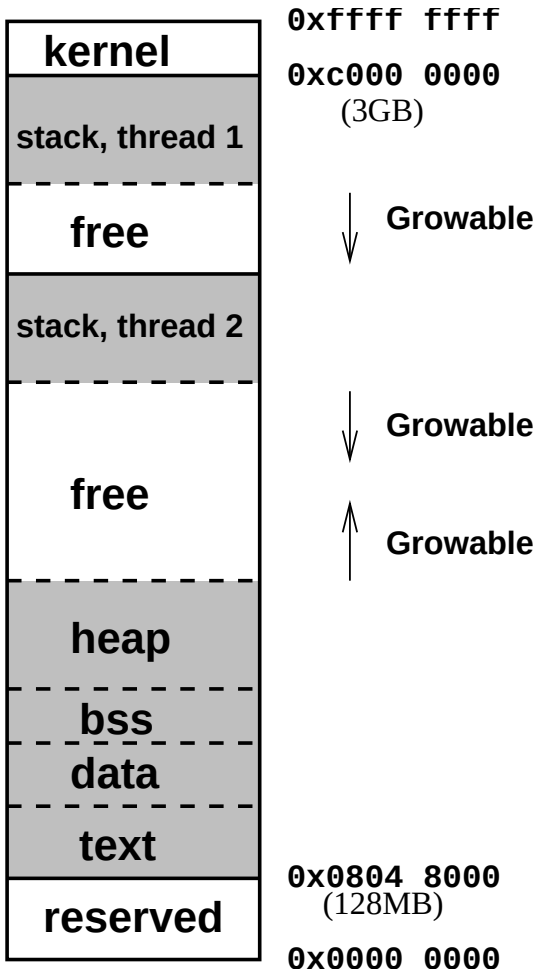
```
#include<stdio.h>

main(){
  int i,j;

#pragma omp parallel for
  for (j=0;j<4;j++){
    printf("j=%d Main at  %lx, i at %lx, j at %lx\n",
           j,(long)main,(long)&i,(long)&j);
  }
}
```

What Went Where

```
$ icc -openmp omp.c
$ ./a.out
j=0 Main at  80489a4, i at bffffcac, j at bffffb5c
j=1 Main at  80489a4, i at bffffcac, j at bffffb5c
j=2 Main at  80489a4, i at bffffcac, j at bffffb5c
j=3 Main at  80489a4, i at bffffcac, j at bffffb5c
$ OMP_NUM_THREADS=2 ./a.out
j=0 Main at  80489a4, i at bffffc9c, j at bffffb5c
j=1 Main at  80489a4, i at bffffc9c, j at bffffb5c
j=2 Main at  80489a4, i at bffffc9c, j at b7e1b15c
j=3 Main at  80489a4, i at bffffc9c, j at b7e1b15c
$ OMP_NUM_THREADS=4 ./a.out
j=0 Main at  80489a4, i at bffffc9c, j at bffffb5c
j=1 Main at  80489a4, i at bffffc9c, j at b7e1b15c
j=2 Main at  80489a4, i at bffffc9c, j at b7c1a15c
j=3 Main at  80489a4, i at bffffc9c, j at b7a1915c
```

One copy of `main` and `i` per process, one copy of `j` per thread.

What Where in Pictures



More Control

OpenMP does allow one to specify specific variables as being private to individual threads, rather than shared. It also allows one to specify how a loop is distributed amongst threads. It even allows an MPI-like parallel region with library calls for establishing how many threads exist, and the thread number of the current thread.

Generally the number of threads is set by the environment variable `OMP_NUM_THREADS`, or, if absent, it defaults to the number of processor cores in the machine.

OMP Integrates

```
program integrate

real (kind(1d0)) :: x, tot
integer :: i,n

write(*,*)'Input number of steps'
read(*,*)n

tot=0d0
do i=1,n
  x=2*(i-0.5d0)/n
  tot=tot+sqrt(4d0-x*x)
enddo

write(*,*)'Integral is ',2*tot/n

end
```

OMP for π

The above serial code integrates over one quadrant of a circle of radius 2. It can readily be timed:

```
$ echo 1e9 | time ./a.out
 Input number of steps
 Integral is    3.14159265358961
        8.22 real         8.19 user         0.01 sys
```

At 8s on a laptop for quite a few significant digits, it hardly needs parallelising. However, one could try a quick

```
!$omp parallel do
```

immediately in front of the loop.

Disaster!

The loop is now much faster, about 4.26s with two threads. However, the answer is now either
`1.91322295498106`,
or sometimes
`1.22836969860869`.

If we compile without optimisation (`-O0`), the answers become more varied. Picking three consecutive runs:

```
2.52073484901867
2.58922465376122
2.54830746007760
```

Yet worse, the time for one thread as `-O0` is 24.2s (and the answer is correct). For two threads it is 23.8s – almost no speed-up.

What Went Wrong

We should have written

```
!$omp do parallel private (x) reduction(+ : tot)
```

This is slightly faster for one thread (21.2s), much faster for two (10.8s), and now gives the correct answer with two threads. Remove the `-O0` and the answer is still correct, and the time is reduced to about $4\frac{1}{4}$s again.

The mistake is 'obvious' – each thread needs its own value of `x`, and its own value of `tot`. However, `tot` is special – on exit from the parallel region, the master thread must gather the sum of all the `tot`s on the slaves.

The default in Fortran is that all variables apart from the loop counter are shared.

The nervous may wish to try something like
`default(none) private(i,x) shared(n) reduction(+ : tot)`
which explicitly specifies everything, and will give an error if anything is omitted.

The result may be split over several lines as

```
!$omp do parallel default(none) private (i,x)
!$omp+  shared(n) reduction(+ : tot)
```

The Error in Detail

The loop is fairly trivial. With standard optimisation, both `x` and `tot` were held in registers. As registers are not shared between threads, each thread correctly calculated its own part of the integral. Which part finally ended up in `tot` was random, but the division of the integral between threads seems to have been constant between runs, so the sum of the two wrong answers does give the correct answer!

With no optimisation, both `x` and `tot` are written out to memory. The same location for each thread. The result is chaos. One thread reads `tot` and a few clock cycles later writes it out again updated, but the other thread may well have read it in the meantime, and will obliterate the first thread's attempt to update it.

The reason it is no faster is that the two CPU cores are updating the same memory locations. The locations cannot be held exclusively in the L1 cache of either core, and, with separate L1 caches per core, but a shared L2, the result is a lot of extra cache traffic. The time to deal with this almost exactly cancels the gain in dividing the work between two cores.

Private Fun

A variable declared as `private` in an OpenMP statement will not be initialised at the start of the parallel region, and will be undefined on exit from the parallel region. The sane therefore don't attempt to access private variables from the serial part of the code.

It is possible to change this behaviour using `firstprivate` (the variable is initialised in each thread to the value it held in the master thread), and `lastprivate` (the value written to the variable in the last iteration of the loop only is retained in the master thread). A variable may be listed in both clauses.

Genuine Varience

Of course, sometimes answers genuinely vary as the number of processors varies. Consider summing

$$\sum_{i=1}^{n} e^{-i/10\ 000} + 10^{-13}$$

The sum quickly reaches around 10,000, after which the further additions of 10^{-13} have no effect, and the exponential soon decays to below this level too. So, summing the first 2×10^9 terms on varying numbers of processors gives:

Processes	Sum
1	9,999.500 008 360 36
2	9,999.500 108 360 35
4	9,999.500 158 360 36
16	9,999.500 195 860 36

```
program series
include 'mpif.h'

integer npe,mype,ierr,i,n,n1,n2
real (kind(1d0)) :: sum,total

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

if (mype.eq.0) then
   write(*,*)'Input number of terms'
   read(*,*)n
endif

! mpi_bcast(buffer, count, type, source, communicator)
call mpi_bcast(n,1,mpi_integer,0,mpi_comm_world,ierr)

n1=(n/npe)*mype+1
n2=(n/npe)*(mype+1)
if (mype.eq.npe-1) n2=n

sum=0d0
do i=n1,n2
  sum=sum+exp(-real(i,kind(1d0))/10000)+1d-13
enddo
write(*,*)'Node ',mype,' total ',sum

! mpi_reduce(send, receive, count, type, operation, root, comm)
call mpi_reduce(sum,total,1,mpi_double_precision,mpi_sum,0,mpi_comm_world,ierr)
if (mype.eq.0) write(*,*)'Total is ',total

call mpi_finalize(ierr)
end
```

OpenMP Restrictions

The `do` loop following a `parallel do` directive must obey additional restrictions above those standard in Fortran:

- no data dependencies between iterations
- using `exit` or `goto` to end the loop are not permitted

Restrictions in C

The restrictions in C are much more numerous, as C does not really have a for loop. The C construction

```
for(expr1;expr2;expr3){expr4;}
```

is equivalent to

```
expr1; while (expr2) {expr4; expr3;}
```

For an OpenMP loop, `expr1` must simply initialise an integer loop counter, `expr2` must test that counter against a loop-invarient expression, and `expr3` must modify it by a constant stride. The loop body may not alter the value of the counter.

Common C constructions such as

```
while (*s++ = *t++);
```

fail at almost every level.

Thread Safety

As OpenMP is always implimented with threads, anything called from a parallel region of an OpenMP code must be *thread safe*. In other words, it must be able to cope with multiple copies of itself being run simultaneously, with each copy having a unique stack but sharing everything else.

To a first approximation, things written in C which avoid `static`, and things written in Fortran which are declared recursive and avoid `save` will be OK, but one often needs to tell the compiler at compile time to make something thread safe.

The current NAG library is not thread safe.

MPI avoids these issues.

OpenMP History

Version 1.0	1997
Version 1.1	1999
Version 2.0	2000
Version 2.5	2005
Version 3.0	2008

(Version 1.1 was Fortran only. The others are all Fortran and C/C++.)

MPI History

Version 1.0	1994
Version 1.1	1995
Version 1.2	1997
Version 1.3	2008
Version 2.0	1997
Version 2.1	2008

MPI-2 introduced one-sided communication and parallel I/O. It is treated as a separate product to MPI-1, of which it is a superset. MPI 1.3 is intended to be the last of the MPI-1 series.

The changes between 1.0 and 1.2 are very minor.

Parallel Problems

Parallel programs are often awkward to debug or optimise, because whereas serial code is by and large deterministic in its behaviour, parallel code is less so. Well-written parallel code is, but that does not need debugging...

Poorly written parallel code may behave differently depending on which order different threads/processes arrive at given points in the code. Behaviour may also vary according to the size of the messages sent.

Trivial Deadlock

Process 0	Process 1
Send to proc 1	Send to proc 0
Receive from proc 1	Receive from proc 0

If the messages sent are small, the above may work. At some point it is likely to fail, for the sends are to processes not attempting to receive, and eventually buffers will fill and the sends must be suspended until the receiving process accepts some data. Which it will never do because it is buried in a similar send itself.

Deadlock Solutions

MPI has a solution to this problem called non-blocking communication. It initiates a send (or receive), but returns without completing if completion is not immediate. The programmer must guarantee not to touch the buffer being used until he has checked that the communication has completed.

Of course, the lazy programmer posts a non-blocking receive, and assumes that a hundred lines later in the code the data are bound to have turned up, so uses the buffer without checking. Some/most of the time this will work.

Strong or Weak Progess?

Any *two-sided* communications model (two-sided: both send and receive required) must have a concept of progress. What is needed for a communication to progress?

Consider a two process code in which

Time	Process 0	Process 1
0s	Non-blocking send to 1	Waits for 10s
	Non-blocking recv from 1	
	Waits 20s	
10s	Still waiting	Matching non-block recv from 0
		Matching non-block send to 0
20s	Finish waiting	Waits 100s
	Wait for comms to end	
110s		Makes MPI call

One might naïvely expect the comms status to be complete after 20s. It might well not be.

The Weak

MPI communication does not occur by magic. In most implimentations, when any MPI call is made, the MPI library checks to see if any other outstanding work exists which can be progressed. If no MPI call is made, the MPI library has no chance to do anything, and nothing will happen. Running the above code under LAM 7.1.2 shows that process 0 does not return from its comms status check until process 1 calls MPI_Finalize and is able to progress its side of the communications. Thus process 0 takes 110s to finish. Under MPICH it takes just 20s. OpenMPI takes 110s.

MPICH is not magic. It just has bigger buffers. The above timings were for exchanging messages of 100,000 bytes. Increase this to 1,000,000 bytes and MPICH takes 110s too.

The lesson is that behaviour can change radically between MPI implimentations and message size, without either behaviour being wrong.

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

int main(int argc, char *argv[]){
  char *a,*b;
  int length=100000;
  int this_process, num_processes, error_code;
  MPI_Status status;
  MPI_Request r1,r2;
  double start_time;

  a=malloc(length); b=malloc(length);

  error_code = MPI_Init(&argc,&argv);
  if(error_code != MPI_SUCCESS){
    printf("MPI initialisation error\n");
    exit(1);
  }
  MPI_Comm_size(MPI_COMM_WORLD,&num_processes);
  MPI_Comm_rank(MPI_COMM_WORLD,&this_process);
  start_time=MPI_Wtime();

  printf("Hello, I am process %d\n", this_process);

  if (this_process==0){
    MPI_Isend(a,length,MPI_CHAR,1,1,MPI_COMM_WORLD,&r1);
```

```
    MPI_Irecv(b,length,MPI_CHAR,1,2,MPI_COMM_WORLD,&r2);
    printf("Process 0 has completed MPI calls, time=%f\n",
            MPI_Wtime()-start_time);
  }
  else if (this_process==1){
    sleep(10);
    MPI_Irecv(a,length,MPI_CHAR,0,1,MPI_COMM_WORLD,&r1);
    MPI_Isend(b,length,MPI_CHAR,0,2,MPI_COMM_WORLD,&r2);
    printf("Process 1 has completed MPI calls, time=%f\n",
            MPI_Wtime()-start_time);
  }

  if (this_process==0){
    sleep(20);
    printf("Process 0 waiting for MPI completion, time=%f\n",
            MPI_Wtime()-start_time);
    MPI_Wait(&r1,MPI_STATUS_IGNORE);
    MPI_Wait(&r2,MPI_STATUS_IGNORE);
     printf("Process 0 finished, time=%f\n",
            MPI_Wtime()-start_time);
  } else sleep(100);

  printf("All done, time=%f\n", MPI_Wtime()-start_time);
  MPI_Finalize();
}
```

MPI vs OpenMPI

MPI has one enormous advantage over OpenMP. Because MPI uses separate processes, with all communication being explicit and no shared memory, there is no need for the processes to run on the same computer. Provided a mechanism exists for launching processes on a remote computer, and exchanging data thereafter, MPI will work. So an ethernet-connected cluster of PCs can be treated as one MPI cluster.

OpenMP effectively requires all the threads to run on a single computer. Thus the job size is limited by the amount of memory (and CPU power) in a single box. And the cost of a computer scales faster than linearly with the memory size above a certain limit.

However, if one's ambitions are modest, OpenMP can be most useful at extracting an extra factor of two or more performance out of current computers, and maybe four or more out of computers which will shortly be available.

The author is aware of many projects to run OpenMP on distributed clusters, and is aware that some of these work in trivial cases.

One can even program in OpenMP in something more like an MPI-style. There are OpenMP function calls to determine how many threads exist, and what one's thread number is, so one can make decisions based on these results. Beware that variables will still default to being shared amongst threads unless one explicitly requests the contrary.

MPI vs F90

The MPI function calls in Fortran are ugly. Most need a buffer, a type, and a length. Surely the overloading and enquiry facilities in F90 should not require this?

```
MPI_Send(buffer, count, type, dest, tag, comm)
```

could surely be

```
MPI_Send(object, dest, tag, comm)
```

Unfortunately Fortran is rather too strongly typed, requiring an explicit interface for every combination. All Fortrans support seven data types (logical, char, int, two floats, two complexes), and most a couple more. They also support scalars, and arrays of up to seven dimensions. So that is $7 \times 8 = 56$ interfaces to `MPI_Send` as a minimum, and there are over 50 functions like `MPI_Send`. So there are around 3,000 interfaces required, and F90 requires all interfaces to be in a single source file. Almost doable until one discovers those MPI functions which take two buffers of independent type, and thus will require over 3,000 interfaces *each*.

Simple Parallel Programming

Various libraries are themselves parallelised using OpenMP. If one links against such a library, then though one's own code is serial, all the library routines may run in parallel.

The obvious example is Intel's MKL, which certainly parallelises LAPACK, Level 3 BLAS and multiple FFTs (or 2D and higher FFTs) efficiently.

Beware! If `OMP_NUM_THREADS` is not set, this library will produce one thread per core on the machine it is run on. So if you run an MPI job on a quad core machine, and each of the four MPI processes uses MKL, each is liable to split into four threads, giving a total of sixteen. MPI users with MKL probably wish to ensure that the product of the number of MPI processes and `OMP_NUM_THREADS` is the number of cores present.

The Mandelbrot Set

For a slightly more detailed example of parallel programming, we consider the generating the Mandelbrot set. This set is formed from the equation

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

Those points in the complex plain for which z_n does not tend to infinity as n increases are deemed to be in the set. Obvious examples include

$z_0 = 0, z_n = 0$ $\qquad$ $z_0 = -2, z_1 = 2, z_2 = 2\ldots$

$z_0 = -1, z_1 = 0, z_2 = -1, z_3 = 0\ldots$ $\qquad$ $z_0 = 0.25, z_1 = 0.375, z_2 = 0.39 \ldots z_\infty = 0.5$

It can be shown that once $|z_n|$ exceeds 2, then $|z_n|$ will increase without limit.

The usual way of producing a pretty picture is, for each z_0 to iterate until either $|z_n| > 2$, or some interation limit is reached. The resulting point is then coloured based on the number of iterations.

Fortran, Page 1

```
function mand(z0,n)
  complex (kind=kind(1d0)) :: z,z0
  integer i,n,mand

  z=z0
  do i=1,n
    z=z*z+z0
    if (abs(z).gt.4d0) exit
  enddo

  mand=512-512*(log(real(i))/log(real(n)))
end function
```

It is conventional to use a logarithmic scale for the colouring.

Fortran, Page 2

```
module pnm
contains
  subroutine ppm_write(set,unit,name)
! set(3,resx,resy) contains r,g,b components for each pixel
  integer :: i,j,k,resx,resy,unit
  integer, allocatable :: set (:,:,:)
  character(*) :: name

  resx=size(set,2)
  resy=size(set,3)

  open(unit,file=name)
! NB First two bytes must be "P3", an initial space is wrong,
! so write(unit,*)'P3' would be wrong
  write(unit,'(''P3'')')
  write(unit,*) resx,resy
  write(unit,*) 255

  do i=resy-1,0,-1
    write(unit,*)((set(k,j,i),k=1,3),j=0,resx-1)
  enddo
  close(unit)

  end subroutine
end module
```

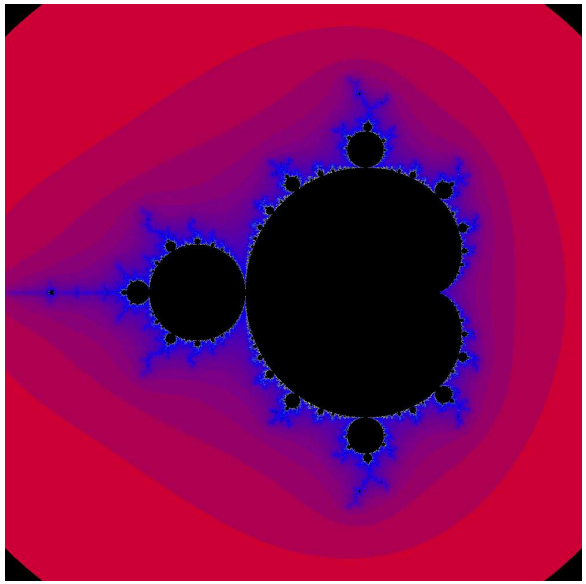
Fortran, Page 3

```
program mandel
  use pnm
  complex (kind=kind(1d0)) :: z
  real (kind=kind(1d0)) :: x,y
  integer :: ix,iy,res,i,pallette(3,0:512)
  integer, allocatable :: set(:,:,:)

  write(*,*)'Please input resolution'
  read(*,*) res
  allocate(set(3,0:res-1,0:res-1))

  pallette(:,0)=0
  pallette(:,512)=0
  do i=1,256
    pallette(1,i)=256-i
    pallette(2,i)=256-i
    pallette(3,i)=i
    pallette(1,255+i)=i
    pallette(2,255+i)=0
    pallette(3,255+i)=256-i
  enddo

!$omp parallel do private (x,y)
  do ix=0,res-1
    x=-2d0+ix*3d0/(res-1)
    do iy=0,res-1
      y=-1.5d0+iy*3d0/(res-1)
      set(:,ix,iy)=pallette(:,mand(cmplx(x,y,kind(1d0)),1024))
    enddo
  enddo

  call ppm_write(set,10,"m.ppm")
end
```

Check

First check that things work in serial mode.

```
$ ifort mandel.f90
$ echo 1000 | time ./a.out
 Please input resolution
5.26user 0.12system 0:06.56elapsed 82%CPU
$ xv m.ppm
```



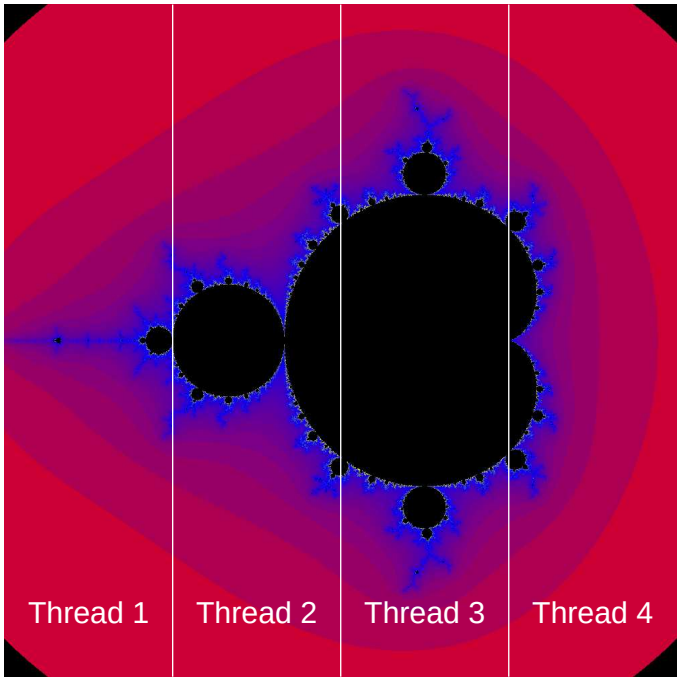
And in Parallel

```
$ mv m.ppm m_serial.ppm
$ ifort -openmp mandel.f90
mandel.f90(66): (col. 7) remark: OpenMP DEFINED LOOP WAS PARALLELIZED
$ echo 1000 | OMP_NUM_THREADS=2 time ./a.out
 Please input resolution
5.48user 0.15system 0:05.31elapsed 106%CPU
$ diff m.ppm m_serial.ppm
$ echo 1000 | OMP_NUM_THREADS=4 time ./a.out
 Please input resolution
5.53user 0.14system 0:04.90elapsed 115%CPU
$ diff m.ppm m_serial.ppm
$
```

Good news! Same answer, and faster.
But 4.9s on four cores compared with 6.6s on one is not very impressive.

A Black Hole

The inside of the set takes 1000 iterations per pixel, and the outside many, many fewer. The standard OpenMP division of the loop will result in one thread dealing with the first 250 columns, the next the next 250, etc. With four threads, the first and last will have very little work, and the third a large amount of work.



More Appropriate Scheduling

To change the division of the loop, we can add
`schedule(static,1)`
to the `!$omp` statement. Then with four threads the first thread will get the first column, and the fifth, and the nineth, the second thread the second, sixth and tenth, etc.

As the amount of work per column changes little between adjacent columns, this should be rather better.

	Before	After
Serial	6.56s	6.56s
Two Threads	5.31s	4.08s
Four Threads	4.90s	2.87s

Another scheduling option is `schedule(dynamic)`. This distributes the iterations between threads dynamically at run-time according to when threads become idle. It is no longer the case that each thread will perform the same number of loop iterations. The overhead of dynamic scheduling is higher. Here it is not measurably higher, for each outer loop contains 1000 iterations of the inner loop, so a reasonable amount of work.

Detecting Black Holes

No of Threads	Before	After
2	5.31s	4.08s
4	4.90s	2.87s
8	3.68s	2.86s
16	3.12s	3.03s

The computer used for the above timings has just four cores, so the program has no business getting faster as the thread count increases above four. However, whereas with the old loop division and four threads, two threads finish quickly and leave two cores idle, with sixteen threads thirteen must exit before CPU cores become idle.

Better I/O

The output file is huge.

```
$ head m.ppm
P3
         1000          1000
          255
 256    0    0 256    0    0 256    0    0 256
```

It is three integers taking four character per byte, or 12MB.

However, Fortran 2003 allows binary output using ‘streeam’ access, and there exists a binary form of the PPM output format.

Fortran 2003

```
module pnm
contains
  subroutine ppm_write(set,unit,name)
  integer :: i,j,k,resx,resy,unit
  integer, allocatable :: set (:,:,:)
  character(*) :: name
  character, allocatable :: line(:)

  resx=size(set,2)
  resy=size(set,3)
  allocate(line(3*resx))

  open(unit,file=name)
! NB First two bytes must be "P6"
  write(unit,'(''P6'')')
  write(unit,*) resx,resy
  write(unit,*) 255
  close(unit)

  open(unit,file=name,access='stream',position='append')
  do i=resy-1,0,-1
    do j=0,resx-1
      do k=1,3
        line(3*j+k)=char(set(k,j,i))
      enddo
    enddo
    write(unit) line
  enddo
  close(unit)

  end subroutine
end module
```

Faster

The wall-clock times are:

	ASCII	Binary
Serial	6.56s	5.40s
Two Threads	4.08s	2.92s
Four Threads	2.87s	1.64s

And the final output file is now 3MB, not 12MB.

(Of course, it would be foolish to store even this for long periods.

```
$ pnmtopng m.ppm > m.png
$ ls -lh m.???
-rw-r--r-- 1 spqr roma 150K Jan 17 16:16 m.png
-rw-r--r-- 1 spqr roma 2.9M Jan 17 16:16 m.ppm
```

Lossless, and 20× smaller.)

Predicting Times

A simple model would suggest that the time taken would be the time for the serial part, plus the time for the parallel part divided by the number of threads. Assuming that the serial part takes 1.6s, and the parallelisable part takes 5s, except on for the binary version, where the serial part ought to be $4\times$ faster, or 0.4s, gives:

	ASCII		Binary	
	Expt	Theory	Expt	Theory
Serial	6.6s	6.6s	5.4s	5.4s
Two Threads	4.1s	4.1s	2.9s	2.9s
Four Threads	2.9s	2.9s	1.6s	1.7s

Pretty good. This simple model is known as Amdahl's Law. It assumes perfect load-balancing and no overheads involved with parallelisation – ridiculously optimistic.

Amdahl's Law is often written as $t = t_s + t_p/n$.

To slow things down suitably, the output was to an NFS-mounted disk on a 100MBit/s network. So the I/O time would be expected to be about 1s per 10MB.

More Amdahl

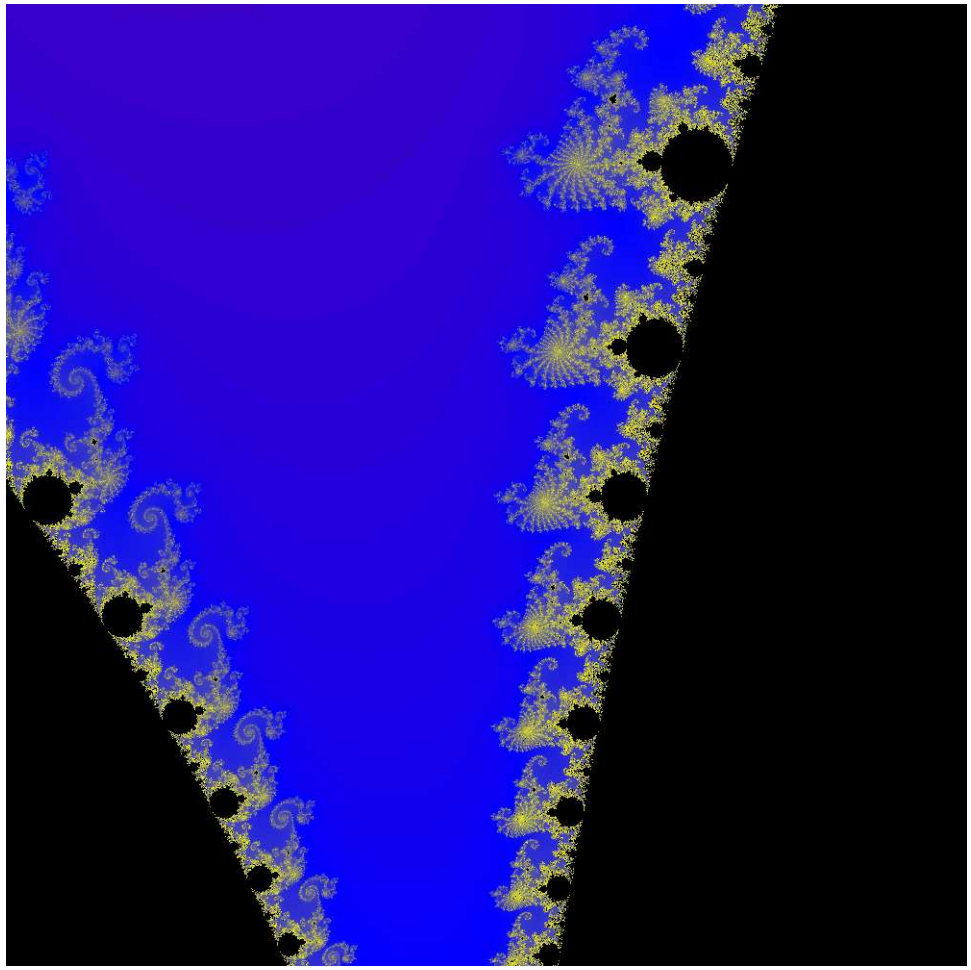
Amdahl's Law is hopelessly optimistic. It says that t decreases monotonically as n increases. However, even with Amdahl a decent 90% of the code being parallelised will lead to a speed-up of just 4.7 on eight cores, and 6.4 on sixteen. To get decent scaling up to hundreds of cores it is necessary to parallelise well over 99% of the code (by serial execution time).

In many cases, the parallel part has a worse scaling in time with problem size than the serial part (perhaps N^2 matrix initialisation vs N^3 diagonalisation). Then big problems will scale to higher core counts than small problems.

In other cases, an MPI broadcast, or an OpenMP/MPI reduce, lurks in the code. That must take a time which scales as $\log_2 n$, producing a term for which time increases as n increases.

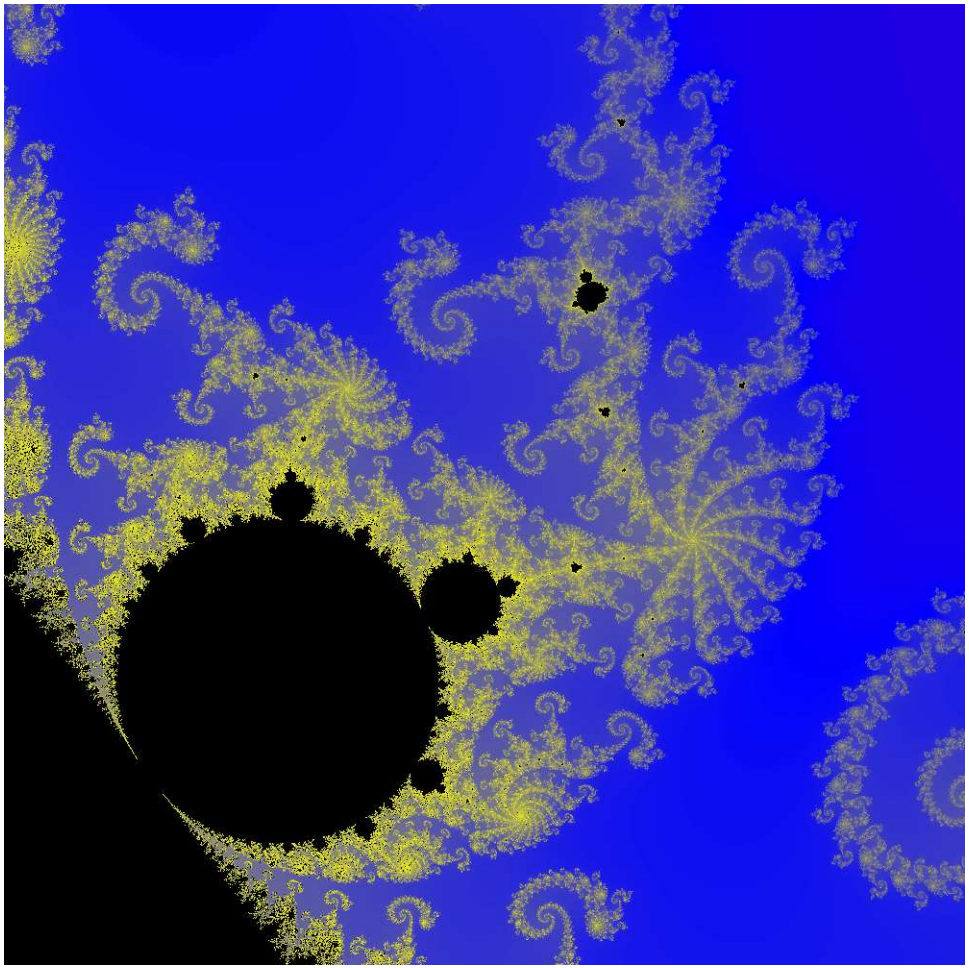
That HPC Centres are excellent for turning CPU-bound problems into I/O bound ones is well known, for parallelising I/O can be hard.

Fractal Fun



Lower left corner, (-0.8,0.1). Range 0.1.

More Fractal Fun



Lower left corner, (-0.8,0.144). Range 0.015.

As the pictures get more complex, so the compressed size increases. The whole set was 129K, the first detail 206K, and this 421K (all sizes as EPS).

Ping Pong

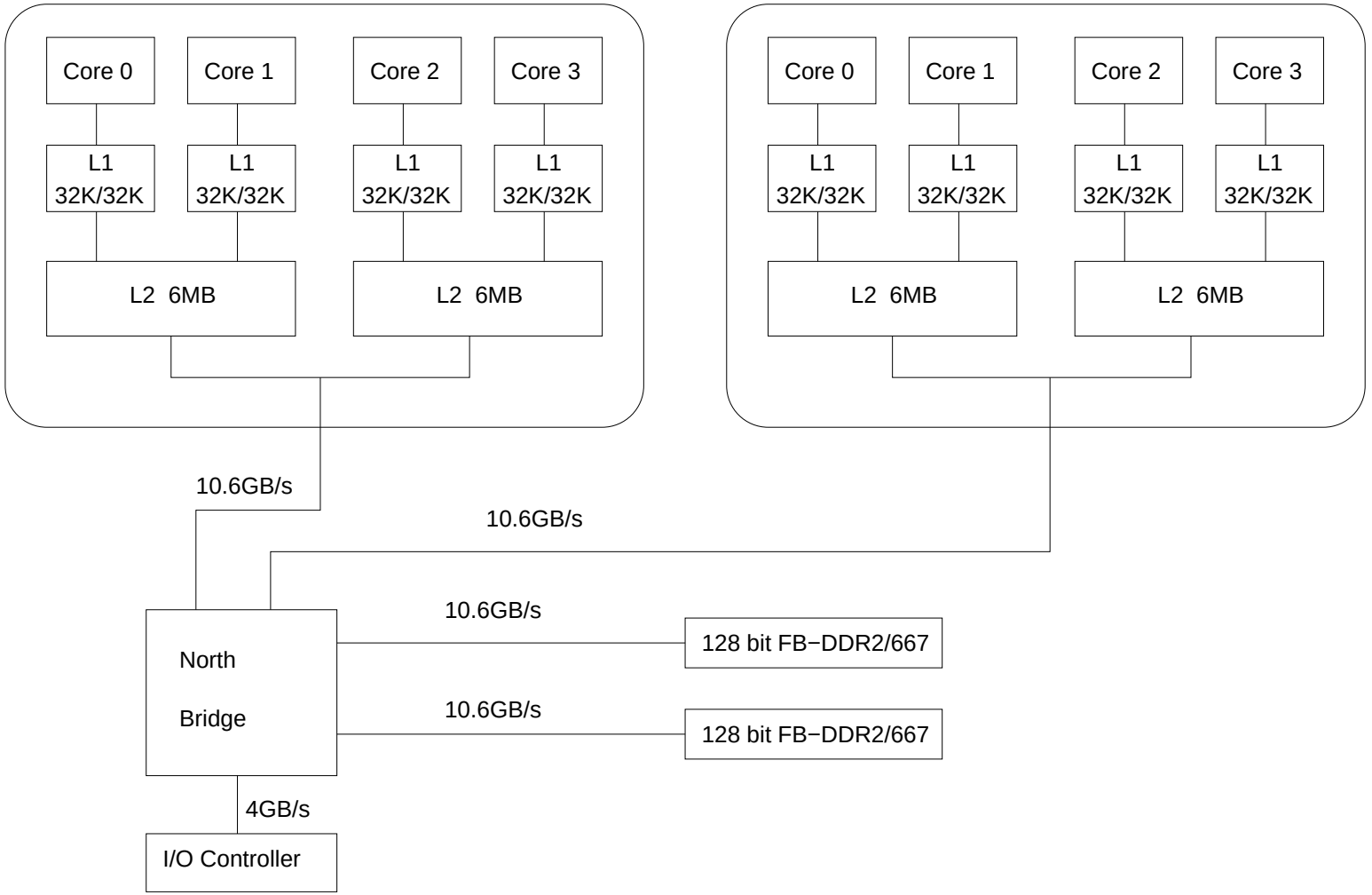
Another, more useful, game is ping-pong. Most relevant to MPI-like systems, it addresses the question of how long it takes for one process to send a message to another.

As for any other network, the obvious model is that there is simply a latency and a bandwidth. For a decent interconnect between separate nodes, the latency might be around 2μs and the bandwidth around 1GB/s. For MPI implimented within a single node, one might hope for 0.5μs and 2GB/s.

Note that sending a message containing a single double precision number will take 2.008μs in the first case. A message containing one hundred double precision numbers would take 2.8μs, whereas a hundred messages containing one number would have taken 200μs.

If possible, bundle small messages into big messages.

More Cans of Worms



The Worms

The above is an approximate diagram of a dual socket quad core Intel computer based on the Core2 processor. At the top are the two CPUs, each with four cores, and at the bottom right is the main memory.

It is immediately obvious that whilst all cores have identical connectivity to memory, the path to transfer data between core 0 and core 1 on the same CPU is shorter than that between core 0 and core 2 on the same CPU, and much shorter than that between cores on different CPUs.

If a parallel job uses little memory, and does much communication between processes, it will run fastest if on a pair of cores sharing an L2 cache on the same CPU. If the opposite, then it would run faster if the processes were on different CPUs.

If a process has exclusive access to core 0, anything running on core 1 (perhaps another job from another user) will cause contention on the L2 cache, and even things on cores 2 and 3 will cause contention on the main memory bus.

Keeping Worms Coherent

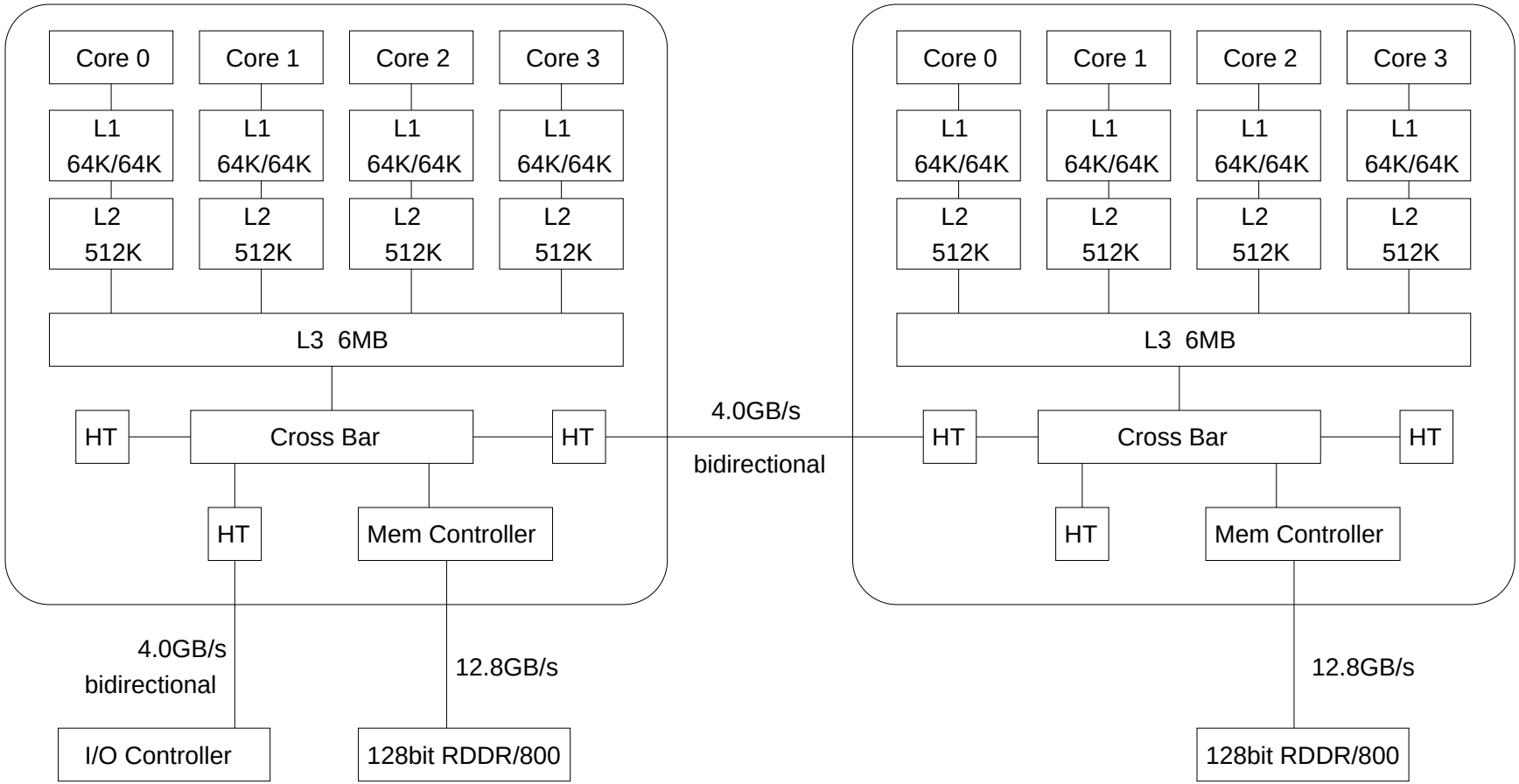
Suppose core 0 CPU 0 and core 1 CPU 1 both read the same item from memory, so there is a copy in both of their L1 caches. Then core 0 CPU 0 writes a new value. Will core 1 CPU 1 see the new value, or the old 'stale' value in its cache?

It had better see the new value, or data transfer will become awkward. Which means what looked like an L1 cache hit has become something involving sending a message via the North Bridge between the CPUs. Messy. (Or MESI, but that is another lecture!)

MESI: Modified, Exclusive, Shared, Invalid – a simple set of book-keeping bits for caches to help determine when problems like the above occur.

The ideal way of using this architecture is probably to place a four process MPI job on cores 0 and 2 of each CPU, and then let each MPI process use a threaded BLAS / LAPACK / FFT library with the second thread running on the other core connected to the same L2 cache. Can I show you the Fortran for that? No. There is no way of enforcing such detailed distribution from Fortran (or C).

Alternative Worms



The speed of the weak link between the two CPUs is doubled in some current, and some near-future, designs.

A More Interesting Collection

The above is AMD's solution, which will become more relevant because it is very similar to the solution Intel will adopt for the dual socket i7 CPUs.

The big difference is that each CPU has its own memory controller, and own bank(s) of memory. Whereas the Core 2 CPU had a single bus for all memory, cache coherency and I/O traffic, this design uses a dedicated memory bus.

Most importantly, if there are four CPUs, there are four memory controllers and buses, and four times the usual memory bandwidth.

Placement

This solution has a Non Uniform Memory Architecture. Accesses from the cores of the first processor to the memory attached to that processor are faster than accesses to memory on the other processor. However, the programmer sees a single pool of memory, and is unaware of the division. Indeed, the division is dynamic.

This causes a head-ache for the OS. Suppose two processes are running on each core, and each process has been allocated memory attached to its local CPU. Then suppose all eight processes on one CPU finish. Should the OS:

1/ Do nothing, leaving one CPU idle
2/ Migrate half the processes to the idle cores, so forcing them to access memory on the 'wrong' processor
3/ Migrate both processes and memory pages, even though that could be several GB?

More Parallel Problems

The last lecture commented that the result given by a buggy parallel program may vary depending on the detailed timing of a given run – which process or thread reached a given point first.

Now we see that the detailed timing may vary considerably depending how the OS decided to distribute processes and memory. This will depend on what else is running on the machine (including kernel and OS processes), and even the recent history of things running on it.

Optimising on current parallel machines is painful, because repeated identical runs can vary in time by several percent, so detect when a code change has made even a 5% improvement is hard.

But for ‘hard,’ read ‘challenging’ or ‘stimulating.’