Parallel Computing
CM30225

Russell Bradford

2023/24

### 1. OpenMP

There are several useful functions

* int omp\_get\_num\_threads(void) returns the number of threads in this parallel region
* int omp\_get\_thread\_num(void) returns a per-thread unique number
* int omp\_get\_max\_threads(void) the maximum number of threads available (often defaults to the number of cores)
* void omp\_set\_num\_threads(int) set the number of threads OpenMP can use
* int omp\_get\_num\_procs(void) number of processors in this system

### 2. OpenMP

And lots more functionality

For example, setting the environment variable OMP\_NUM\_THREADS before running the program sets the default number of threads

OMP\_NUM\_THREADS=7 ./prog

OpenMP is widely supported. For example, to compile under GCC:
cc -fopenmp -Wall -o prog prog.c

### 3. OpenMP

OpenMP is clearly naturally associated with shared memory

There is a distributed memory version from Intel, called *Cluster OpenMP*

There is an undercurrent of “if your program doesn’t work well on normal OpenMP, then it won’t work well on Cluster OpenMP”

### 4. OpenMP

OpenMP

* is an evolving standard
* is easy to use; you can modify existing programs incrementally
* hides messy threads fiddling
* needs compiler support, unlike pthreads (but is supported by the mainstream compilers, in particular GCC, Clang and MSVC)
* is dependent on good implementation of the compiler: if you pass control of the parallelism to a compiler you need that compiler to be good at it
* is very large and complicated in scope
* still allows trivially buggy programs

### 5. OpenMP

**Exercise** Would the coursework be easier using OpenMP?

### 6. Cilk Plus

Of course, OpenMP is not the only way of tweaking C

*Cilk Plus* is somewhat similar in that it adds annotations and is based on fork and join

But as new keywords in C, not as pragmas (mostly)

Cilk Plus is intended as an extension to C++, but works for C, too

You may come across other versions named “Cilk” and “Cilk”

We may have time to talk about Cilk later

### 7. Shared Memory

This concludes our discussion of the shared memory world

For now

### 8. Distributed Memory

We now turn to distributed memory programming

We could use interfaces like threads or OpenMP and have an underlying or virtualising infrastructure that converts them to message passing between processors over a network

Good programmers don’t like that as it hides the source of the cost of distributed parallelism from the programmer, making it harder to design and write efficient programs

So most distributed programs are explicitly message passing, or have some other way of making the cost of an operation more clear

### 9. Distributed Memory

The big player in this field is *Message Passing Interface* (MPI)

You may hear about

* PVM: Parallel Virtual Machine, a predecessor to MPI
* SHMEM: SHared MEMory, only on Cray (SGI) machines
* UPC: Unified Parallel C, a supposed successor to MPI

### 10. MPI

MPI is what Big Science uses, when terabytes of data crunching is needed

And remember distributed systems are not good for small programs due to the overhead of the messaging outweighing the parallelism gained

MPI runs the same program on multiple processors (SPMD), but definitely not in lockstep

The processes communicate via messages

### 11. MPI

MPI is “simply” a library of functions to do messaging; you can use it with normal (unmodified) C, Fortran, etc.

Even Java, Python and other languages less suited to high performance systems

MPI is actually a standard with several competing implementations

Code written to the standard should run on any implementation

But frequently doesn’t

The MPI standard specifies a huge number of functions, covering a wide range of different types of messaging

### 12. MPI

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

int main(int argc, char \*\*argv)
{
 int rc, myrank, nproc, namelen;
 char name[MPI\_MAX\_PROCESSOR\_NAME];

 rc = MPI\_Init(&argc, &argv);
 if (rc != MPI\_SUCCESS) {
 printf ("Error starting MPI program\n");
 MPI\_Abort(MPI\_COMM\_WORLD, rc);
 }

 MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myrank);
 MPI\_Comm\_size(MPI\_COMM\_WORLD, &nproc);

*continued*

### 13. MPI

 if (myrank == 0) {
 printf("main reports %
 }

 namelen = MPI\_MAX\_PROCESSOR\_NAME;
 MPI\_Get\_processor\_name(name, &namelen);
 printf("hello world %

 MPI\_Finalize();
 return 0;
}

### 14. MPI

Notes:

* MPI\_Init(&argc, &argv); Set up the system: you must always do this. A batch processing system (e.g., SLURM) starts the processes on all the processors, while MPI\_Init sets up the connections between them
* Later versions of MPI allow MPI\_Init(NULL, NULL) but the above is preferable as it provides more information to the MPI system
* rc Always check to make sure it worked
* MPI\_COMM\_WORLD The system can be sub-divided into subsets of processors called *communicators*. The WORLD communicator is all processors; MPI\_COMM\_SELF refers to just the calling processor

### 15. MPI

* MPI\_Comm\_rank Each process in a communicator has a unique rank within that communicator: this is just an integer from 0 to *size of the communicator* $−1$. So, for WORLD the rank ranges from 0 to *total number of processors* $−1$
* MPI\_Comm\_size Get the size of the communicator
* if (myrank == 0) All processors run the same code (SPMD). This is how we get different things happening on different processors
* MPI\_Finalize All procs must always call this to tidy up their MPI state

### 16. MPI

Compile using mpicc:

mpicc -Wall -o hellompi hellompi.c

### 17. MPI

Batch file runnit.slm:

#!/bin/sh
#SBATCH --account=cm30225
#SBATCH --partition=teaching
#SBATCH --job-name=HelloMPI
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=8

mpirun ./hellompi

### 18. MPI

The lines of note here are:

* –nodes=2 we want two nodes
* –ntasks-per-node=8 we will be using just 8 of the 44 cores on each node

Recall we had:

 if (myrank == 0) {
 printf("main reports %
 }

 namelen = MPI\_MAX\_PROCESSOR\_NAME;
 MPI\_Get\_processor\_name(name, &namelen);
 printf("hello world %

### 19. MPI

Output:

hello world 3 from 'ip-AC125409'
hello world 5 from 'ip-AC125409'
hello world 4 from 'ip-AC125409'
hello world 11 from 'ip-AC125408'
hello world 6 from 'ip-AC125409'
hello world 9 from 'ip-AC125408'
hello world 1 from 'ip-AC125409'
hello world 15 from 'ip-AC125408'
hello world 7 from 'ip-AC125409'
hello world 12 from 'ip-AC125408'
hello world 2 from 'ip-AC125409'
hello world 10 from 'ip-AC125408'
main reports 16 procs
hello world 0 from 'ip-AC125409'
hello world 14 from 'ip-AC125408'
hello world 13 from 'ip-AC125408'
hello world 8 from 'ip-AC125408'

### 20. MPI

Notes:

* ip-AC125408 and ip-AC125409 are the names of the two nodes that happened to be allocated; the next run may well get different nodes
* Processes 0–8 are on ip-AC125409 while processes 9-15 are on ip-AC125408, but it might happen the other way around
* ntasks-per-node is important here as sometimes you want fewer MPI tasks on a node than there are cores on that node: an MPI task can itself be multithreaded (not your coursework!)

### 21. MPI

* Output in a random order, even for the “main reports 16 procs” which we might think happens first!
* We *do* see “main reports” before “hello world 0”, though!
* MPI has a mechanism for routing prints on any node back via the network to a single point: this results in all kinds of timing variations in output

### 22. MPI

* MPI is SPMD, so this code is *not* synchronised across processors
* For example, when proc 0 is doing its printf the other processors may well already be doing MPI\_Get\_processor\_name
* Or perhaps still MPI\_Comm\_size
* But many MPI function calls do have a built-in synchronisation and block the calling processor until all processors involved in that call are done
* Each MPI “task” is a separate *process*, not sharing anything with any other task: in particular, not sharing any variables (e.g., myrank), even if the tasks happen to be on the same node

### 23. MPI

**Exercise** Does adding a MPI\_Barrier after the “main reports” conditional *ensure* the message comes out first?