An Introduction of High Performance Computing

Lecture Notes

jerakrs

January 2018

Abstract

This document is the lecture note of *An Introduction of High Performance Computing*. The course code is *COMS30005* and the unit director is *Simon McIntosh-Smith*.

1 BlueCrystal

BlueCrystal is the University's High Performance Computing machine. BlueCrystal Phase 3 (*bluecrystalp3.bris.ac.uk*), is available to all users. Phase 4 (*bc4login.acrc.bris.ac.uk*) is primarily intended for large parallel jobs and for work requiring the Nvidia P100 GPUs.

1.1 Logging In

BlueCrystal only allows the user who is inside the University firewall to access directly.

Logging in the BlueCrystal Phase 3 by ssh command:

```
$ ssh username@bluecrystalp3.bris.ac.uk
```

Changing password, run this command and follow the prompts to type the old and new password:

```
$ passwd
```

It can use graphical tools remotely by adding **-X** flag when connecting:

```
$ ssh -X username@bluecrystalp3.bris.ac.uk
```

Passwordless access, it allows user to set up the SSH keys so that the user can connect BlueCrystal without typing password:

```
$ ssh-copy-id username@bluecrystalp3.bris.ac.uk
```

This command will copy the content of the *public key* file (**id_rsa.pub**) to BlueCrystal's **~/.ssh/authorized_key**. If there no SSH key in local machine, run this command to make a new key:

```
$ ssh-keygen
```

1.2 The Queuing System

BlueCrystal Phase 3 is made up of 4 head nodes and 341 computing nodes. The user will work on the headnodes, but the tasks should run on the computing nodes. Therefore there is a queuing system to manage jobs.

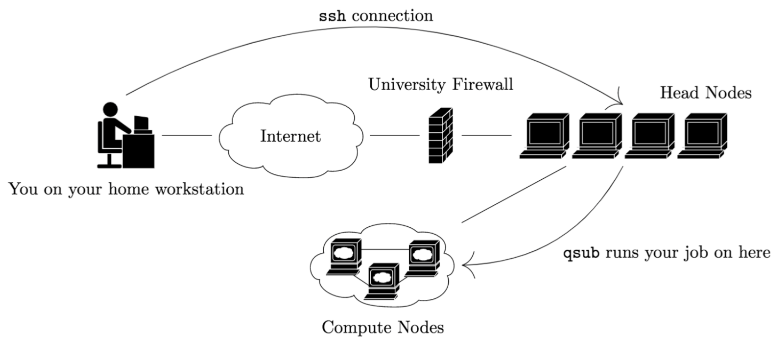


Figure 1: The Structure of Queuing System

Submitting the jobs by qsub command:

```
$ qsub example.job
```

The **example.job** is a script file, the content of this file is shown below.

Listing 1: Example Script File

```
#!/bin/bash

#PBS -N lab1
#PBS -o lab1.out
#PBS -joe
#PBS -q teaching
#PBS -l nodes=1:ppn=16
#PBS -l walltime=00:01:00

cd $PBS_O_WORKDIR

echo Running on host `hostname`
echo Time is `date`
echo Directory is `pwd`
echo PBS job ID is $PBS_JOBID

./lab1
```

Viewing the status of submitted jobs:

```
$ qstat -u $USER
```

The S column gives the job status. **R = running, C = complete, Q = queued**.

Deletingthe a submitted job:

```
$ qdel jobid
```

1.3 Environment Modules

BlueCrystal has lots of extra software, it allows users to load the modules which they need. When user submits a new job, it need to load the modules that this task will use in the script file, or put them into the user's **.bashrc** file so they are loaded when user log in and for every job the user run.

View available software:

```
$ module avail
```

Combining the grep command, it can search the module:

```
$ module avail 2>&1 | grep intel
```

Load and unload the module:

```
$ module load languages/gcc-4.8.4
$ module unload languages/gcc-4.8.4
```

List loaded modules:

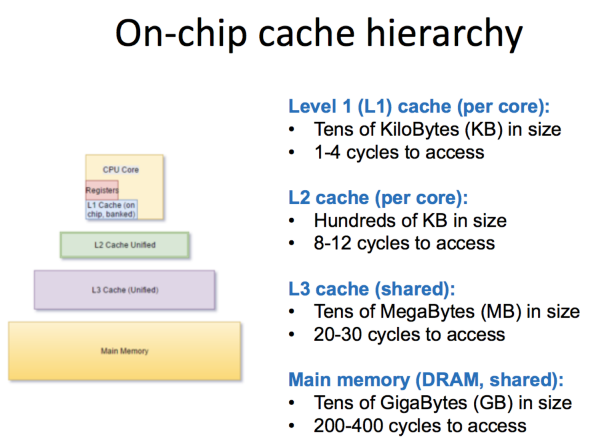
```
$ module list
```

2 Processors

The most important HPC trends:

- Microprocessor performance $\sim 55\%$ per annum.
- Memory capacity $\sim 49\%$ per annum.
- Memory bandwidth $\sim 30\%$ per annum.
- Memory latency $<< 30\%$ per annum.

Cache Hierarchy:



On-chip cache hierarchy

Level 1 (L1) cache (per core):
- Tens of KiloBytes (KB) in size
- 1-4 cycles to access

L2 cache (per core):
- Hundreds of KB in size
- 8-12 cycles to access

L3 cache (shared):
- Tens of MegaBytes (MB) in size
- 20-30 cycles to access

Main memory (DRAM, shared):
- Tens of GigaBytes (GB) in size
- 200-400 cycles to access

Figure 2: The Structure of Cache Hierarchy

3 Serial Code Optimisations

- Algorithm Matter: different algorithms have different complex rate.
- Examine the within-core performance: finding the critical code and only attempting to optimise the critical code.
- Compiler and Flags:
  - gcc flags: -O2, -O3, -ffast-math
  - icc flags: -fast
- The memory hierarchy:
  - Row vs. Column Major Order
  - don't re-visit memory
  - 'Blocked' loop
- Vectorisation: making use of wide registers is important on modern processors (SIMD: SSE, AVX, AVX2, AVX512).

4 OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++ and Fortran on most platforms.

Threads vs. Processes: Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources.

Two Method for Multi-threads:

- The old way: Posix Threads, it is tedious and error prone.
- OpenMP: serial code with *#pragma* compiler directives.

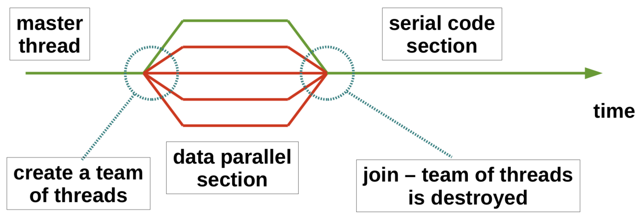


Figure 3: Open MP

4.1 Key Features

- Compiler directives: *#pragma*
- Thread team creation: *parallel*
- Data attributes: *shared*, *private*
- Team operations: *reduction*
- Work sharing: *for* in loops; *section*, *task* in less structured.
- Syncronisation:
  - Implicit at end of work sharing loops
  - Mutex (one processor at a time): *critical*
  - Wait for all threads: *barrier*
  - One thread executes: *single*, *master*

4.2 Runtime Library

- Get the number of threads in team: *omp_get_num_threads()*
- Get the thread id: *omp_get_thread_num()*
- Control the number of threads via an environment var: *OMP_NUM_THREADS*
- Control the number in the program: *omp_set_num_threads(N)*

4.3 Performance

Accumulator

- Shared Accumulator lead to a long critical bocking time.
- Array of Accumulators leads to thrashing cache.
- Private Accumulators.
- Reduction will make a copy of the reduction variable per thread, and after the loop, the local variable will be combined into the global variable using the reduction operator.

Schedule

- *static*: dividing the loop into equal-sized chunks or as equal as possible.
- *dynamic*: using a work queue to give a chunk- sized block of tasks to each thread.
- *guided*: same to dynamic scheduling, but the chunk size is decreasing.

The *nowait* Clause allows program to avoid the implicit barrier at the end of a worksharing for loop.

4.4 Potential Benefits

Amdahl's Law

$$speedup = \frac{T_1}{T_p} \tag{1}$$

$$speedup_{max} = \frac{1}{1-P} \tag{2}$$

$$speedup_{ideal} = \frac{1}{\frac{P}{N} + S} \tag{3}$$

Notice that:

- T_1 is the execution time on a single processor.
- T_p is the execution time on a parallel computer.
- S is the serial fraction of the program.
- P is the parallelisable fraction of the program.
- N is the number of processors available.

Gustafson's Law

$$speedup = \frac{T_1}{T_p} = \frac{T_a(x) + N \times T_b(x)}{T_a(x) + T_b(x)} \to N \tag{4}$$

Notice that:

- x is a measure of problem size.
- N is the number of processors.
- $T_a(x)$ is fraction of time spent executing the serial part of the program.
- $T_b(x)$ is fraction of time spent executing the parallel part.

4.5 New features in OpenMP v4

NUMA: Under NUMA, a processor can access its own local memory faster than non-local memory, which also provides higher overall memory bandwidth. It uses 'first touch' policy to control placement of items across the banks of memory.

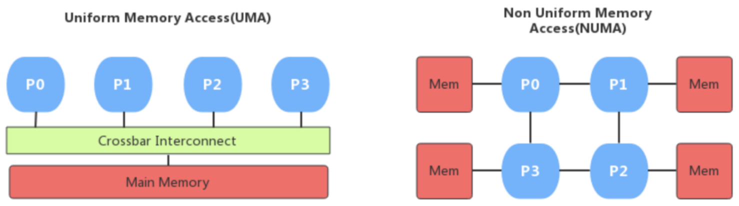


Figure 4: UMA vs. NUMA

binding policy:

- threads: corresponding to a single hardware thread.
- cores: corresponding to a single core.
- sockets: corresponding to a single socket.

SIMD Directive: a work sharing loop can be executed using SIMD lanes *#pragma omp parallel for simd.*

5 Roofline Model

The Roofline model is an intuitive visual performance model used to provide performance estimates of a given compute kernel or application running on multi-core, many-core, or accelerator processor architectures, by showing inherent hardware limitations, and potential benefit and priority of optimizations.

Operational Intensity:

- Operations per byte of memory traffic.
- An operation could be floating point, integer... Traffic is measured at main memory (DRAM).

$$OI = (ops)/(bytes). \tag{5}$$

$$Attainable\ GFLOP/s = \min\begin{cases} Peak\ floating-point\ performance \\ Peak\ memory\ bandwidth * Operational\ intensity \end{cases} \tag{6}$$

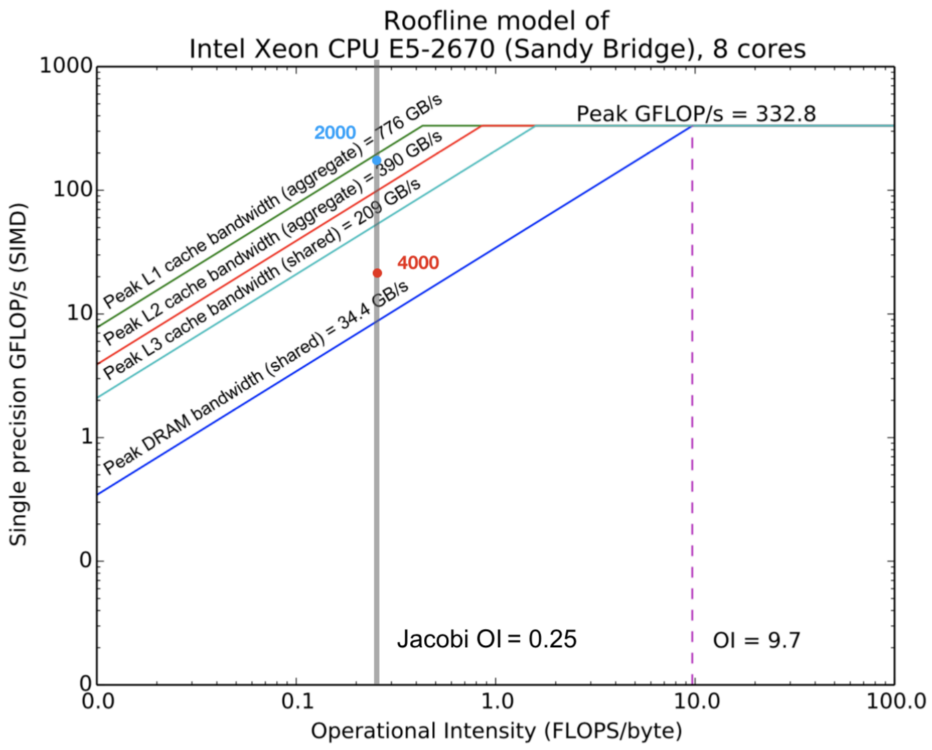


Figure 5: The Example of Roofline Model

6 OpenMPI

The Open MPI Project is an open source Message Passing Interface implementation that is developed and maintained by a consortium of academic, research, and industry partners. MPI has a well-designed library that is intended to be portable and it is available for C and Fortran.

Point-to-point communication

- Asynchronous:

```
void MPI_Send(buffer, data_length, data_type,
        dest, tag, MPI_COMM_WORLD);
void MPI_Recv(buffer, data_length, data_type,
        source, tag, MPI_COMM_WORLD);
```

- Safe, Portable: *MPI_Ssend()*
- Buffered, Blocking send: *MPI_Bsend()*
- Non-blocking: *MPI_Isend()* and *MPI_Irecv()*
- Packed data messages: *MPI_Pack* and *MPI_unpack*