Tutorial 1
Parallel Computer Architecture and Slurm

CS3210 – 2025/26 Semester 1

Learning Outcomes

Part 1: Tutorial

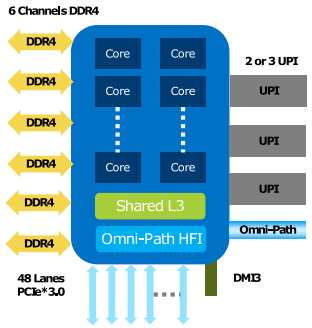
1. Recall the various levels of parallelism
2. Understand and apply the Flynn's taxonomy of computer architectures
3. Understand the various ways memory can be organized

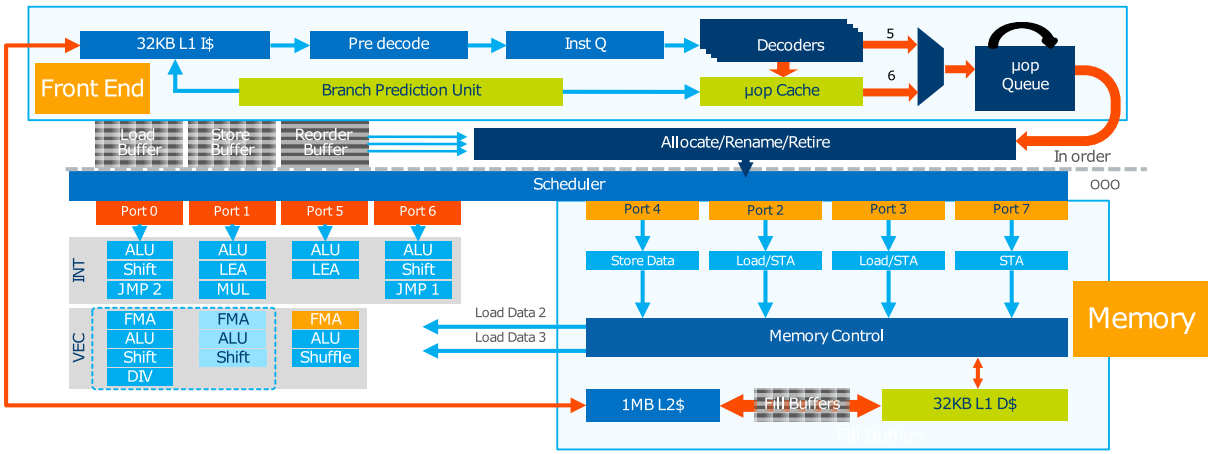
Part 2 and 3: Lab

4. Understand the hardware configuration of our lab cluster
5. Set-up and run applications on our lab cluster using Slurm

Part 1: Tutorial on Parallel Computing Architectures

1. **(Levels of Parallelism)** A simplified diagram of the Intel Xeon Scalable processor family and its micro-architecture diagram is reproduced below:





Briefly discuss and identify the various forms of parallelism that are supported by this processor chip. Feel free to search online for any unfamiliar terminology.

Useful information on Intel Xeon Scalable

- https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview
- https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-scalable-platform-brief.pdf (page 9)

2. **(Flynn's Taxonomy)** For each of the following scenarios, classify them according to Flynn's taxonomy (i.e. SISD, SIMD, MISD, MIMD) and provide a brief (say, one line) justification
   (a) A personal computer from the 1980s
   (b) A laptop with a multi-core processor
   (c) Intel's AVX instruction set
   (d) A uniprocessor (i.e. single core) with pipelining (ILP)
   (e) Students attempting the same exam in an exam hall

3. **(Memory + Multi-core Architecture)** For each of the following questions, indicate whether the statement is true or false. Briefly justify your answer.
   (a) Shared-memory system implies a UMA architecture.
   (b) On a NUMA architecture, the actual location of data (i.e. on which node) has no impact for a distributed shared-memory program.
   (c) In the hierarchical design of multicore architecture, the memory organization is hybrid (distributed-shared memory)

4. **(Processes and Threads)** For each of the following questions, indicate whether the statement is true or false. Briefly justify your answer.
   (a) A semaphore can be used to replace a mutex without affecting the correctness of a program.
   (b) Implementing an algorithm using multiple threads always runs faster than implementing the same algorithm using multiple processes.

Part 2: Foundation: Parallel and Distributed Computing Lab Cluster

To start using our lab cluster to its full potential, we need to **understand the available hardware.**

Lab Machines: Physical Layout

The diagram below shows an example of the arrangement of machines in each workbench at the Parallel Computing Lab at COM1-B1-02, where machines are connected by switches:

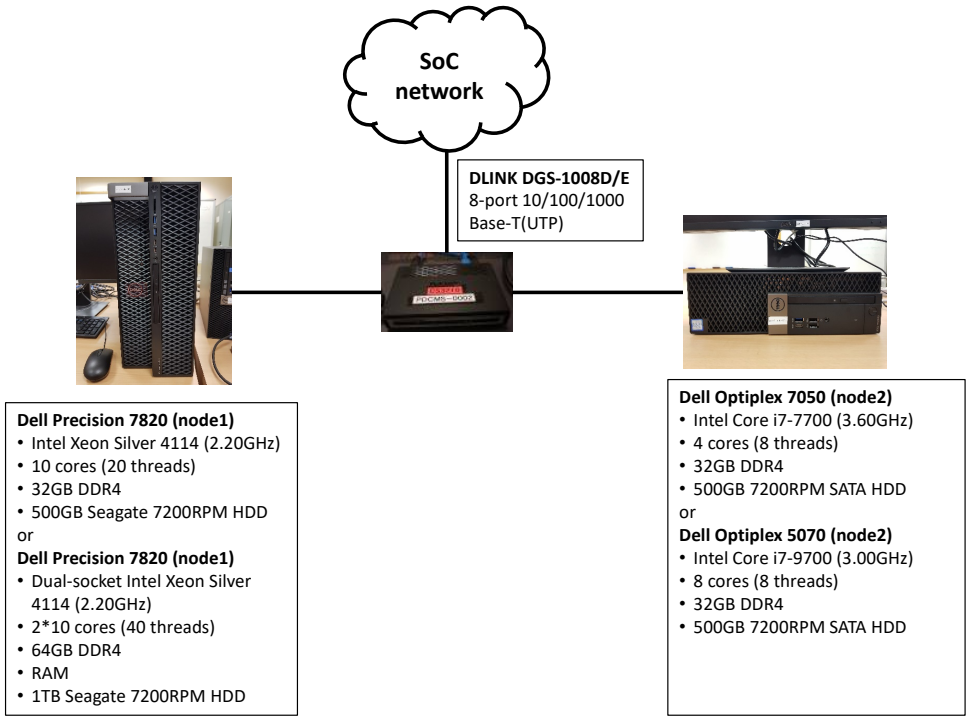


Figure 1: Example layout of machines on each workbench (not all machines shown)

The hostnames and the types of nodes are shown below.

- `soctf-pdc-001` - `soctf-pdc-008`: Intel Xeon Silver 4114
- `soctf-pdc-009` - `soctf-pdc-016`: Intel Core i7-7700
- `soctf-pdc-018` - `soctf-pdc-019`: Dual-socket Intel Xeon Silver 4114
- `soctf-pdc-020` - `soctf-pdc-021`: Intel Core i7-9700
- `soctf-pdc-022` - `soctf-pdc-024`: Intel Xeon W-2245 (not shown above)
- `soctf-pdc-025` - `soctf-pdc-032`: Intel Xeon W5-3425 (not shown above)
- `soctf-pdc-033` - `soctf-pdc-040`: Intel Core i7-13700 (not shown above)

Exercise 1

Please `ssh` into one of our lab machines as in Lab 1. What is the hardware configuration of the lab machine you are currently connected to? Run:

`$ lscpu`

`$ lstopo` (or `lstopo --of ascii` for a more graphical view)

Some questions to think about (non-exhaustive):

1. What is a socket and how many do you have?
2. What are, and what are the relationships between CPUs, cores, and threads?
3. What are the different levels of cache present and how large are they?

Part 3: Using Slurm Workload Manager (Practical)

Please read this section carefully and try all the exercises.

Please also open our **Student Guide** (https://comp.nus.edu.sg/~cs3210/student-guide).

In case of any conflicting information after this tutorial, the Student Guide takes precedence, as it is updated regularly throughout the semester.

What is Slurm? Why does it exist?

So far, you have been running your programs *interactively*. That is, you have been connecting *directly* to a specific machine over `ssh`, and executing programs on it through an interactive shell (e.g., `bash`).
However, there are a few reasons why running programs interactively **not ideal**:

- **"Hogging" the machines**: Running massive CPU-intensive jobs on a login node causes the node to become unresponsibe for other students. Some students may not be able to run their code in a timely manner. This problem is exacerbated during assignments that may need to be executed across multiple machines simultaneously. **Do not run CPU-intensive jobs on login nodes! You can run them with Slurm.**
- **Cannot get accurate performance readings**: Students may not be able to get an accurate measurement of how fast their solutions are. If many students are logged into a single node and running their programs simultaneously, all of their programs will seem slower (take more wall-clock time) than if each student had exclusive access to the node for a short period.

To address these problems, a significant number of machines in the Parallel and Distributed Computing Lab are controlled by the **Slurm Workload Manager**. Slurm performs three key functions:

1. **Fair Job Queue:** Students will submit **jobs** to Slurm through commands such as `sbatch` and `srun`. These jobs will be placed in a queue and then scheduled to run on the lab machines once enough resources are available for the specific job. Every student will get their fair share of time to run their programs - nobody can hog the machines.
2. **Exclusive Access for Performance Measurements:** When a student's job is running on Slurm nodes, no other jobs will run on those nodes. This ensures that any performance measurements can be taken accurately and free from the effects of other users' jobs.
3. **Running Jobs Across Multiple Machines:** Slurm provides a framework to distribute computation across many physical machines, and integrates with existing parallel and distributed computing systems such as OpenMP and MPI.

In fact, NUS SoC uses Slurm across our entire Compute Cluster, as do other major institutions (Harvard, MIT), industry users (NASA, AWS, IBM), and even over half of the top ten supercomputers.

Logging in & Learning About Slurm

As in Lab 1, please `ssh` into one of our lab machines.
(If you have forgotten how to do so, please refer to the Lab 1 pdf).

Please continue to the next page once you have connected to a lab machine. The following commands will not work unless you are running them on a lab machine.

Step 1: Understanding our Slurm Cluster

Now that we understand the individual lab machines better, let's look at how our Slurm manages our cluster of machines.

Exercise 2

On your lab machine, run the following command and observe the output:

```
$ sinfo
```

Your `sinfo` output will look similar to this (may not be exact, depending on the state of the cluster)

```
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
all*         up    3:00:00     16   idle soctf-pdc-[004-008,012-016,018-021,023-024]
xs-4114      up    3:00:00      5   idle soctf-pdc-[004-008]
i7-7700      up    3:00:00      5   idle soctf-pdc-[012-016]
dxs-4114     up    3:00:00      2   idle soctf-pdc-[018-019]
i7-9700      up    3:00:00      2   idle soctf-pdc-[020-021]
xw-2245      up    3:00:00      2   idle soctf-pdc-[023-024]
```

Figure 2: Possible `sinfo` output

`sinfo` lists information about Slurm nodes and **partitions**. A partition is a logical grouping of compute nodes which can each be thought of as a job queue. For example, if a user submits a job to the `xs-4114` partition above and asks for a single node, the job can run on any of `soctf-pdc-004` to 008. The partition with a "*" symbol is the *default partition* that will be used to run a job if a partition is not explicitly specified.

Notice that we have created **separate partitions for each hardware type**. For example, the `xs-4114` partition only contains Intel Xeon Silver 4114 nodes. This allows students to queue jobs to run only on a specific hardware type, which may be useful for performance measurements later on.

Lastly, take note that you *cannot ssh directly to these machines*. This allows for accurate performance measurement as students can be guaranteed that only Slurm-allocated jobs are running on those nodes.

Step 2: Running your first jobs!

We finally have enough information to run our first Slurm job. We can use the `srun` command to send very simple commands as jobs to the Slurm system.

Exercise 3

Run the following commands and observe the output:

```
$ hostname
$ srun hostname
```

The first command is a normal shell command that executes on your local host machine. You should see it print the name of the machine you are currently logged into. The second command uses `srun` (think of it as short for "Hey Slurm, please run this") to run the `hostname` command as a job on the Slurm system.

You should see that the second command prints the hostname of a node *other* than the one you are currently logged into. `srun` caused Slurm to allocate a node from the cluster to you, and it automatically ran the `hostname` executable on it, printing the hostname of the allocated node.

Congratulations - you just ran a job on a totally different machine by using Slurm!

Let's actually see what Slurm allocated to us during our previous command.

Exercise 4

Run the following command and observe the output (you may need to wait some time for it to update after running a previous job):

```
$ sacct
```

You might see something like this:

```
JobID           JobName  Partition    Account  AllocCPUS      State ExitCode
------------ ---------- ---------- ---------- ---------- ---------- --------
258            hostname        all      staff         20  COMPLETED      0:0
```

Figure 3: Possible `sacct` output - your Account should be `students`, however

`sacct` shows us our past jobs and the Slurm "accounting" data for them, which tracks how many resources we used. Notice above that we allocated **20 CPUs** (your number may be different: equal to every CPU available in the node we ran on) for our simple `hostname` job (even though we didn't actually need 20 CPUs worth of computing power). As we had *exclusive access* to the node during our job, no other jobs could have run on it, so we are still "charged" by the Slurm fair-use accounting system for allocating 20 CPUs.

Let's get Slurm to stretch its legs by running our `hostname` command across *multiple nodes*.

Exercise 5

Run the following commands and observe the output:

```
$ srun -N3 -n3 hostname
(equivalent to $ srun -N3 --ntasks-per-node=1 hostname)
$ sacct
```

Notice that this job printed out three *different* hostnames. The fact that we see the output of the `hostname` commands on our current node implies that Slurm collates all the standard output (and error) streams from the allocated nodes. By using the `-N3` option, we have asked Slurm to run our job on 3 separate nodes, and `-n3` indicates we want to only run 3 tasks in total (1 per node).

We can also get Slurm to run our jobs on specific partitions, or even specific nodes.

Exercise 6

Run the following command and observe the output:

```
$ srun --partition i7-7700 bash -c "lscpu; hostname"
```

Verify that your job was executed on a node within the i7-7700 partition. We use `bash -c` to execute multiple commands within one Slurm job.

Choose your favorite hostname from our lab machines (make sure it's managed with Slurm by checking `sinfo`). Run this command (assuming you chose a hostname of `soctf-pdc-0xx`):

```
$ srun -w soctf-pdc-0xx hostname
```

Verify that your job was executed on the node you specified.

Step 3: Running real batch jobs with `sbatch`

While `srun` is useful for testing and as a component of larger jobs, we rarely submit jobs directly this way. We usually submit jobs using a command like `sbatch job.sh`, where `job.sh` is a Slurm *batch script* written by you. Here are some reasons why:

- `srun` runs in an **interactive** and **blocking** mode - your terminal blocks until the command completes, and you see the output of stdout and stderr in your terminal. `sbatch` runs in a **batch, non-interactive** mode, so you can submit your job and disconnect safely.
- `srun` requires you to list all your options as command line parameters (e.g., `-w soctf-pdc-0xx`, `--ntasks-per-node=1`). `sbatch` scripts can include all these options as part of the script itself, as we will see soon.

Let's write our first Slurm batch script!

Exercise 7

Create a file in your directory called `basic.sh`. Observe the code listing below and duplicate it **exactly** into `basic.sh`. Save the file.

basic.sh

```
#!/bin/bash
#SBATCH --job-name=basic
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --mem=1gb
#SBATCH --time=00:00:30
#SBATCH --output=basic_%j.log
#SBATCH --error=basic_%j.log

echo "Running basic job!"
echo "We are running on $(hostname)"
echo "Job started at $(date)"
# Sleep "job", we use "srun" for better accounting
srun sleep 5;
# This is useful to know (in the logs) when the job ends
echo "Job ended at $(date)"
```

Notice that our job script has several lines starting with `#SBATCH`. These are *Slurm directives* which affect the configuration of our job: these should be mostly self-explanatory given your experience with `srun`. There are a few new options: `--time` which is limited to 30 seconds for this job, `--job-name` which gives your job a human readable name in the queue, and `--output` which tells Slurm what to call your logfile which has all the printouts from this run.

There are a few crucial points to notice before we finally run our script:

stdout/err file name

We are asking Slurm to collate the script output into a file called `basic_%j.log`, where Slurm will replace %j with your job ID. If your job ID was 231, your stdout and error will be in `basic_231.log`.

As the %j variable only applies inside the #SBATCH directives, in the rest of the shell script, the same value is available in the environment variable `SLURM_JOB_ID`.

Now we can finally run our job script.

Exercise 8

Run our job script as follows, and be prepared to run a command immediately after:

```
$ sbatch basic.sh
```

Now immediately run the following command a few times until your job completes and observe the output:

```
$ squeue
```

(alternatively, try `watch -n 0.5 squeue`) to auto-run the command every 0.5 seconds.

Finally, locate your logfile inside your directory (use `ls`) and look at its contents. If the job ID was 231, we could print the logfile with:

```
$ cat basic_231.log
```

You should observe a few things:

- **No output from sbatch:** Note that sbatch itself did not write any output to your screen (besides the job ID), which is unlike srun's behavior. All output is redirected to the logfile and ultimately copied to your directory.
- **squeue output while waiting:** Your job may have to wait for other jobs to complete. If it is waiting only because of a lack of resources (nodes), the REASON column will state "(Resources)". If it is waiting due to other jobs having priority over yours, it will state "(Priority)"
- **squeue output while running:** You should see your job in the running state ("R" under the "ST" column), and the time it has been executing. You can also see the node assigned to your job under the NODELIST(REASON) columns.

Step 4: Running custom executables with `sbatch`

In the past few examples, we have only been running executables that are *already pre-installed* on each of the cluster's machines (e.g., `hostname`, `sleep`, `date`, etc). What if we want to run our own programs?

Exercise 9

1. Download last lab's code to your folder via `wget` https://www.comp.nus.edu.sg/~cs3210/L0_code.zip, then run `unzip` on it.
2. Compile the `ex6-cond-example.cpp` file into an executable called `cond`.
3. Run the `cond` executable normally (i.e., `./cond`) and ensure that it is working normally as expected.

Now, we will set up a Slurm job to run `cond`. We will make only very minor changes to the original `basic.sh`,

Exercise 10

Create a file in your directory called `cond.sh`. Observe the code listing below and duplicate it into `cond.sh`. Save the file.

cond.sh

```
#!/bin/bash
#SBATCH --job-name=cond
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --mem=1gb
#SBATCH --time=00:00:30
#SBATCH --output=cond_%j.log
#SBATCH --error=cond_%j.log

echo "Running cond job!"
echo "We are running on $(hostname)"
echo "Job started at $(date)"
# NOTE: LINE BELOW CHANGED TO RUN COND
srun ./cond
echo "Job ended at $(date)"
```

Ensure that the `cond` executable is in the same folder. Now, run this job with `sbatch cond.sh` and look at the output of the logfile (in your directory).

The job should end successfully in approximately 10 seconds or so. **Congratulations - you have just run your first custom executable on Slurm!**

Step 5: Bringing it all together: Preliminary Performance Evaluation and Optimization

So far, you have experienced Slurm in a guided and structured manner. To conclude this tutorial, you will self-explore some the strengths that Slurm has (exclusive node access for performance evaluation, ease of code execution on different machines, etc) to perform a real task.

Exercise 11

This tutorial requires a file called `pthread_addsub.cpp`, which should be downloable from https://www.comp.nus.edu.sg/~cs3210/T1_code.zip. Download that file to your home folder of your current machine (remember: use `wget` and `unzip`).

Compile that file with the command:

```
$ g++ -o pthread_addsub pthread_addsub.cpp -pthread
```

Your goal: Use Slurm to investigate the time taken for the `pthread_addsub` program to run across each of our lab machine hardware types, and the reasons for what you observe.

Some guiding questions:

- Which hardware type can run our program the fastest? Why? Note that we have 7 different hardware types in our lab (Intel Xeon Silver 4114, Intel Core i7-7700, Dual-socket Intel Xeon Silver 4114, Intel Core i7-9700, Intel Xeon W-2245, Intel Xeon W5-3425, Intel Core i7-13700), and each is on a different Slurm partition.
- What happens if you change the value of `DELAY`? Does this trend change or scale differently? Why?
- What happens if you change the value of `ADD_THREADS` and `SUB_THREADS`? Does this trend change or scale differently? Why?
- Is there an optimum value of `ADD_THREADS` and `SUB_THREADS` for a given type of machine? Why?

Use what you have learnt so far to write the necessary Slurm batch scripts to explore these questions. Try to focus on why you see the results you do, and not just the results themselves.

In case of technical issues

If there is something **operationally wrong** (not debugging related!) with the cluster (e.g., nodes are unexpectedly down, no internet access, your user account doesn't work with Slurm, etc), please email us.

Conclusion and Summary

In this section, you have explored most of Slurm's basic features, such as:

1. Using `sinfo` to view partition information.
2. Using `srun` to run simple commands.
3. Using `sbatch` to run more complex jobs.
4. Using `sacct` and `squeue` to monitor job information.

There is much more to Slurm, and more that we will explore this semester. We will soon we how to use Slurm to run multithreaded/multiprocess code, OpenMP programs, and even distributed MPI programs!

Further reading

1. The **Slurm Student User Guide** (https://comp.nus.edu.sg/~cs3210/student-guide/) will contain the most updated information about our node allocations, policies, and advanced Slurm topics.
2. The Slurm official quickstart guide (https://slurm.schedmd.com/quickstart.html).
3. The `man` pages for common Slurm commands such as (sacct, sbatch, scancel, ...) .