Supercomputing Systems 101

Edgar A. León and **Jane E. Herriman**
Lawrence Livermore National Laboratory

LLNL-CODE-812647

Tech Workshop @Tapia Conference
Friday 9 September 2022
Washington, DC

Table of contents

1. Learning objectives

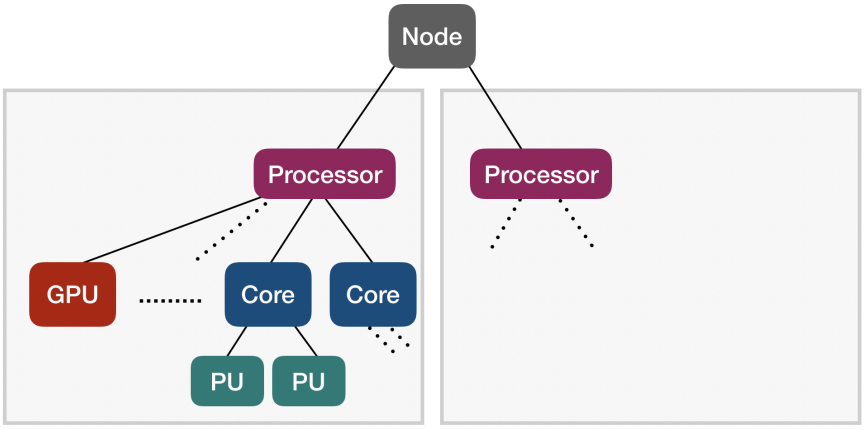
- Learn basics of computer architecture, as needed for this workshop
- Learn how to use `hwloc` to explore node topology
- Understand the concepts of *affinity*, *binding*, and *mapping*
- Run a parallel program using `Slurm`
- Identify the resources available to the tasks of MPI programs using `mpibind`
- Learn how to set affinity implicitly and explicitly with `Slurm`

2. Computer architecture

Before introducing you to the `hwloc` library, we want to cover some of the computer architecture terminology we'll be using throughout this module and tutorial.

Compute only

Let's start by considering compute resources only (and ignoring memory).



At the bottom of the tree shown above, we see "Processing Units" denoted as "PU"s. A **PU** is the smallest processing unit that can handle a logical thread of execution, i.e. execute its own set of instructions. Multiple PUs are sometimes packaged together into a **core**. PUs have both dedicated and shared hardware resources on a core; for example, floating point units are shared by multiple PUs.

In our tree, we see **GPU**s, Graphical Processing Units, shown at the same level as cores. In contrast to cores and their PUs, GPUs allow for greater data parallelization by working on vectors of data at once.

Multiple cores and possibly one or more GPUs are included on a single processor. Each of these processors is a set of compute resources written onto a single piece of Silicon (a die).

Finally, at the top of our tree is a node, which you can think of as a stand-alone computer. Modern nodes are often built from multiple processors, and the example architectures we'll consider each have two processors.

Comprehension question 1

The number of hardware threads on a processor is equal to

A) The number of cores per processor

B) The number of PUs per core

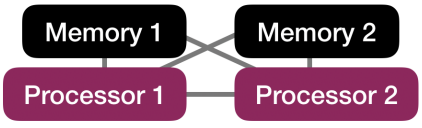
C) The number of cores per processor x the number of PUs per core

▶ Answer

Adding in memory

Once we throw memory into the picture, we need to consider not only what resources are available, but how they're physically arranged and, therefore, how easily they can talk to one another.

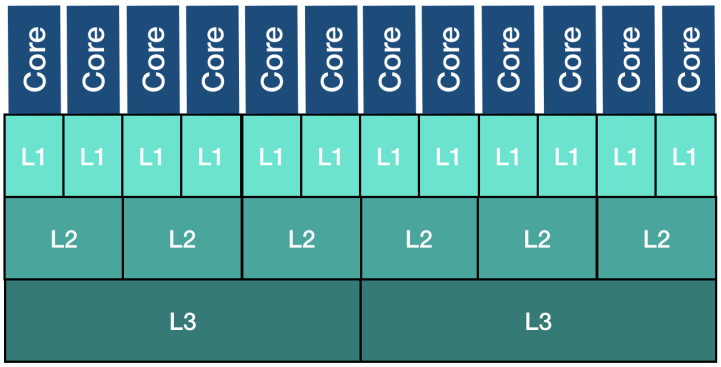
In the image below, consider a scenario where we have two processors and two stores of memory. In the layout shown, processor 1 is closer to memory 1 than to memory 2; similarly, processor 2 is closer to memory 2 than memory 1. This means that processor 1 can more easily and *more quickly* access data stored in memory 1 than data stored in memory 2 and vice versus. In this case, the processors have Non-Uniform Memory Access (NUMA) and we say that memory 1 and processor 1 are in the same NUMA domain. Processor 2 shares the second NUMA domain with memory 2 and will access data in memory 1 with higher latency.



We can imagine scenarios where memory is laid out to be equidistant from multiple processors and where multiple processors are in the same NUMA domain. In the architectures we'll consider, however, there will be a one-to-one mapping between NUMA domains and processors; all computing resources on a Silicon die will be in the same NUMA domain and different processors will have different "local" memory.

Our references to "memory" above refer to memory that's transmitted over a frontside bus. In contrast, **cache memory** serves as a faster and closer source of memory, and different cores on the same processor and within the same NUMA domain may have access to different cache.

In general, cache levels are denoted as `L<N>` where `<N>` denotes the cache level. Lower values of `N` denote smaller and faster levels of cache. In the figure below, we see an example of what the cache hierarchy and layout might look like on a single processor.



In this example cache layout, there are three levels of cache -- `L1`, `L2`, and `L3`. Each core has its own `L1` cache, every two cores share a `L2` cache, and sets of six cores each have a `L3` cache.

3. Example architectures

The topologies of a few example architectures are summarized and diagramed here.

In examples throughout the coming modules, we'll be focusing on Pascal and sometimes Corona!

Comprehension question 2

How many PUs per core does Corona have?

A) 4

B) 2

C) 8

▶ Answer

4. Discovering node topology

`hwloc` is your friend

You can discover the topology of a node with `lstopo` and `lstopo-no-graphics`. These tools are provided by a library called `hwloc` and respectively give graphical and textual descriptions of node topology.

The images used to show the topologies for `Corona` and `Pascal` in the Example Architectures section above were produced by `lstopo`. On the other hand, text output on Pascal, for example, looks like

▶

```
janeh@pascal83:~$ lstopo-no-graphics
```

Hands-on exercise A: Experimenting with `lstopo`

By default, `lstopo` and `lstopo-no-graphics` show the topology of the machine you're logged into. Alternatively, you can pass an `.xml` file describing the topology of a *different* machine to see the topology of that machine

you can pass an `.xml` file describing the topology of a *different* machine to see the topology of that machine.

From your AWS desktop, try the following:

```
lstopo --input ~/pascal.xml
```

and then try:

```
lstopo-no-graphics --input ~/pascal.xml
```

Hands-on exercise B: Investigating AWS nodes with `lstopo`

Let's use `lstopo` to show the topology of the nodes we can see through AWS. The node you see immediately after logging in doesn't have the most interesting topology, but the nodes waiting for you in the "queue" have more features. To see the topology of one of these nodes, use the following command:

```
srun -p<QUEUE> -t1 lstopo
```

How are the features of the node described by `lstopo`'s output different than those of Pascal?

Customizing `lstopo` output

A few other basic parameters can help to customize outputs, such as `--only <type>`.

`--only <type>` causes the topology of only `type` devices to be described:

```
janeh@pascal83:~$ lstopo-no-graphics --only NUMANode
NUMANode L#0 (P#0 63GB)
NUMANode L#1 (P#1 63GB)
```

Possible options to pass to `only` include `NUMANode`, `core`, `PU`, and `OSDevice`.

Hands on Exercise C: Experimenting with `--only`

Run

```
lstopo-no-graphics  --input ~/pascal.xml --only core
```

and

```
lstopo-no-graphics  --input ~/pascal.xml --only core | wc -l
```

to see the first the full list of cores on Pascal and then to tally them.

Now try

```
lstopo-no-graphics  --input ~/pascal.xml --only PU | wc -l
```

to see how many hardware threads there are on Pascal. How many hardware threads are there per core on Pascal?

A) 2

B) 4

C) 1

▶ Answer

Hands-on exercise C: Investigating AWS nodes with `--only`

Use the commands

```
srun -p<QUEUE> -t1 lstopo-no-graphics --only core | wc -l
```

and

```
srun -p<QUEUE> -t1 lstopo-no-graphics --only PU | wc -l
```

to identify the number of cores and PUs on our AWS nodes. Do these nodes support Simultaneous Multi-Threading (SMT)? How many hardware threads exist per core?

5. Mapping, affinity, and binding

Mapping: A function from the worker set (our set of tasks and/or threads) to the hardware resource set.

Affinity: Policies for how computational tasks map to hardware.

Binding: Mechanism for implementing and changing the mappings of a given affinity policy.

Mappings, *affinity*, and *bindings* are key and interrelated concepts in this tutorial, and so we want to define and distinguish between them. When we know a program's hardware mapping, we know which hardware resources (for example, which cores and which caches) are available to each of the workers executing the program. The affinity is a policy that informs the mapping used for a particular combination of hardware and workers, and then workers are bound to hardware resources as defined by the mapping.

For example, an affinity policy asking processes to spread out may result in a mapping where Process 1 is assigned to Core 1 and Process 2 maps to Core 16. After binding to Core 16, Process 2 can start performing computational work there.

Since we're just getting started, these concepts may still feel abstract, but our hope is that after seeing some concrete examples in this module, these definitions will start to feel more straightforward.

MPI mapping 1

Suppose we have an affinity policy that specifies **one MPI task per `Core` on `Pascal`**. We might get the following mapping:

- *Task 0: PU 0,36*
- *Task 1: PU 1,37*
- ...
- *Task 18: PU 18,54*
- ...
- *Task 35: PU 35,71*

Note that each core on `Pascal` has two processing units (PUs), so every task assigned to a single core is assigned to two PUs. Each `Pascal` node has 2 processors each with 18 cores for a total of 36 cores per node, so we can have a max of 36 tasks on a single node given the "one task per core" affinity policy.

MPI mapping 2

Next, suppose we had an affinity policy with **two MPI tasks per `NUMANode` on `Corona`**. In the image depicting `Pascal`'s topology above, we see that each of two NUMA domains on `Pascal` has 18 cores and 36 PUs, so we get 9 cores and 18 PUs per task:

- *Task 0: PU 0-8,36-44*

- *Task 1: PU 9-17,45-53*
- *Task 2: PU 18-26,54-62*
- *Task 3: PU 27-35,63-72*

Comprehension question 3

Match the following:

"Task 8 —> PUs 24-26 and 72-74"

"Policy for one openMP thread per L3 on Corona"

A) Affinity

B) Mapping

▶ Answer

6. Running parallel jobs and reporting affinity

Using Slurm to run a parallel job

In Hands-On Exercises B & D, we used commands that began with `srun -p<QUEUE> -t1`. `srun` is a command that comes from a piece of software called Slurm and allows us to run parallel programs. In order to run parallel programs, we often need to request special resources. For example, if we want to run a program across several nodes, we first need to request access to use several nodes at once. Slurm helps manage requests like this, and figures out how to "schedule" jobs in environments where many users can make requests for the same resources.

Hands-on exercise D: Serial *Hello World*

First, let's try running a serial version of "Hello world". You should see the binary `hello` at `~/hello`. Run this binary by simply typing `hello` and **Enter** at the command prompt. You should receive a single greeting from a single node.

Hands-on exercise E: Parallel *Hello World*

Now, try running

```
srun -ppdebug -t1 -N1 -n2 hello
```

and then

```
srun -ppdebug -t1 -N1 -n4 hello
```

How many greetings did you receive for each of these runs?

Note that `-n` specifies the number of tasks that will run your program.

`-N` specifies the number of nodes you want Slurm to use to run your program, `hello`.

If you leave the value passed to `-n` fixed, does changing the value passed to `-N` change the number of greetings?

The flags `-ppdebug` and `-t1` tell Slurm which set of nodes to draw from and for how many minutes you expect your job to run.

Reporting affinity

Next, we'll run a few parallel jobs that use multiple tasks to execute a binary named `mpi` instead of `hello`. The program `mpi` reports affinity. In other words, the output of `mpi` will tell us where & with which resources the tasks executing `mpi` were run. For example, if we run `mpi` on a single node and with 2 tasks:

```
$ srun -ppvis -t1 -N1 -n2 ./mpi
pascal5    Task   0/  2 running on 18 CPUs: 0-17
pascal5    Task   1/  2 running on 18 CPUs: 18-35
```

Note that the CPUs on the node `pascal5` were split evenly between the two tasks created.

Similarly, if we increase the number of tasks to 4 and other keep the command, we see

```
$ srun -ppvis -t1 -N1 -n4 ./mpi
pascal6    Task   0/  4 running on 9 CPUs: 0-8
pascal6    Task   3/  4 running on 9 CPUs: 27-35
pascal6    Task   1/  4 running on 9 CPUs: 9-17
pascal6    Task   2/  4 running on 9 CPUs: 18-26
```

A new node, `pascal6` was assigned to this job, and the cores on Pascal were split evenly between the tasks.

Comprehension question 4

Now let's throw a second node into the mix. We run the same two commands above but change `-N1` to `N2`:

```
$ srun -ppvis -t1 -N2 -n2 ./mpi
pascal10   Task   1/  2 running on 72 CPUs: 0-71
pascal9    Task   0/  2 running on 72 CPUs: 0-71
$ srun -ppvis -t1 -N2 -n4 ./mpi
pascal8    Task   2/  4 running on 18 CPUs: 0-17
pascal8    Task   3/  4 running on 18 CPUs: 18-35
pascal7    Task   0/  4 running on 18 CPUs: 0-17
pascal7    Task   1/  4 running on 18 CPUs: 18-35
```

The total number of tasks doesn't change, but the assignment of tasks to cores does. When we have two tasks and two nodes, each task gets its own node.

If we ran `srun -ppvis -t1 -N2 -n8 ./mpi` on Pascal, how many tasks would run `mpi`? How many cores would each task have access to?

Hands-on exercise F: Reporting affinity

From your AWS instance, run

```
srun -ppdebug -t1 -N1 -n2 ./mpi
```

and then

```
srun -ppdebug -t1 -N2 -n8 ./mpi
```

How many cores are assigned to each task in each case?

7. Setting Affinity

In HPC environments, the resource manager controls, manages, and assigns computational resources to user jobs. The resource manager both assigns compute nodes to jobs and on-node resources to the job's tasks. *Resource managers are also a way to provide affinity.*

A key benefit of applying affinity through the resource manager is that this approach works across MPI libraries. We could instead handle affinity at the MPI-library level, but because each MPI library has its own interface and policies that provide affinity, this is not a portable solution when we switch from one MPI library to another. For this reason, we prefer to handle affinity at the resource manager level rather than at the MPI-library level.

In this section, we focus on the **Slurm** resource manager and delve into two aspects:

1. User-friendly bindings using high-level abstractions.
2. Binding using low-level abstractions, which provide the most flexibility.

Before diving into the technical details, let's agree on some terminology:

- When a user submits a job, the resource manager will allocate a set of compute nodes to run the job; we call these nodes the `node allocation`.
- Within this allocation a user may run a single job or multiple jobs; Slurm refers to these as `job steps`, but in this module, unless specified otherwise, we will simply call them `jobs`.
- In addition, to be consistent with Linux, we use the term `CPU` to refer to a hardware thread.

Simple (implicit) binding

Once the tasks have been distributed and ordered among compute nodes, the next natural step is to assign on-node resources to each task. For this step we will leverage the following options:

```
-c, --cpus-per-task=<ncpus>
--cpu-bind=none|threads|cores|sockets|rank
```

Let's review a few examples.

Binding to threads

When `-c` is used with `--cpu-bind`, `-c` specifies the number of objects to be bound to each task and `-cpu-bind` specifies the type of object.

We see that below, where `--cpu-bind=thread` specifies thread as the object considered by `-c`. `-c6` therefore specifies that each task should be assigned to 6 threads. Contrast this with the behavior in the next example.

```
# Bind each task to 6 hardware threads
$ srun  -N1 -n4 -c6 --cpu-bind=thread ./mpi
corona294  Task   0/  4 running on 6 CPUs: 0-5
corona294  Task   1/  4 running on 6 CPUs: 6-11
corona294  Task   2/  4 running on 6 CPUs: 12-17
corona294  Task   3/  4 running on 6 CPUs: 18-23
```

Binding to cores

As in the example above, `-c6` means that each task will be assigned to 6 objects. Here, `--cpu-bind=core` specifies `core` as the object type, so each task will be assigned to 6 cores.

On `Corona`, each core has two hardware threads, so each task will be assigned to 12 threads. For example, Task 3 is assigned to the threads `18-23` and `66-71`.

```
# Bind each task to 6 cores
# Remember that each core is comprised of 2 hardware threads
# For example, the first core has CPUs 0,48
$ srun  -N1 -n4 -c6 --cpu-bind=core ./mpi
corona294  Task   0/  4 running on 12 CPUs: 0-5,48-53
corona294  Task   1/  4 running on 12 CPUs: 6-11,54-59
corona294  Task   2/  4 running on 12 CPUs: 12-17,60-65
corona294  Task   3/  4 running on 12 CPUs: 18-23,66-71
```

No binding

With `--cpu-bind=none`, no binding occurs and tasks are not bound to specific resources. Instead, all tasks are distributed across all threads. In this case, 4 threads are distributed across 96 threads.

```
# No binding, everybody runs wild!
$ srun  -N1 -n4 --cpu-bind=none ./mpi
corona294  Task   0/  4 running on 96 CPUs: 0-95
corona294  Task   1/  4 running on 96 CPUs: 0-95
corona294  Task   2/  4 running on 96 CPUs: 0-95
corona294  Task   3/  4 running on 96 CPUs: 0-95
```

Hands-on exercise G: Setting affinity implicitly

From your AWS instance, run

```
srun -ppdebug -t1 -N1 -n4 -c2 --cpu-bind=thread ./mpi
```

and then

```
srun -ppdebug -t1 -N1 -n4 -c2 --cpu-bind=cores ./mpi
```

Are the CPUs assigned to each task the same in each case? Why?

More elaborate (explicit) binding

More advanced use cases may require placing tasks on specific CPUs. For example, one may want a monitoring daemon running on the *system cores* or, say, the last core of the second socket. To do this explicit binding, Slurm provides the following parameters to an already familiar option:

```
--cpu-bind=map_cpu:<list>
--cpu-bind=mask_cpu:<list>
```

With `map_cpu` one specifies a 1:1 mapping of tasks to CPUs, one CPU per task. The first task is placed on the first CPU in the list and so on. With `mask_cpu` one specifies a 1:1 mapping of tasks to sets of CPUs, one CPU set per task. The first task is placed on the first CPU set in the list and so on. A set of CPUs is specified with a CPU mask.

While these options provide the flexibility to implement custom mappings, the downside is the user must know the topology to determine the CPU ids of the resources of interest. Thus, this is not a portable solution across systems and architectures.

We will resort to more examples for ease of exposition.

Custom bindings with `map_cpu`

In this first example with `map_cpu`, we want to use the first three cores of both the first and second socket. Cores `0,1,2` on socket 1 will be assigned to tasks `0-2` and cores `24,25,26` from socket 2 will be assigned to tasks `3-5`:

```
# Place tasks 0-2 to the first three cores of the first socket
# Place tasks 3-5 to the first three cores of the second socket
$ srun  -N1 -n6 --cpu-bind=map_cpu:0,1,2,24,25,26 ./mpi
corona191  Task   0/  6 running on 1 CPUs: 0
corona191  Task   1/  6 running on 1 CPUs: 1
corona191  Task   2/  6 running on 1 CPUs: 2
corona191  Task   3/  6 running on 1 CPUs: 24
```

```
corona191  Task   4/  6 running on 1 CPUs: 25
corona191  Task   5/  6 running on 1 CPUs: 26
```

Notice that in the above example we merely specified the set of CPUs to use and the number of tasks; the ordering of the cores in the list passed to `--cpu-bind=map_cpu:` determined the exact assignment of tasks to cores.

Round robin tasks with `map_cpu`

As in the last example, we'll assign 6 tasks to the same 6 cores. This time, we'll manually assign these tasks in a round robin fashion between the two sockets by changing the order of the list of cores passed to `--cpu-bind=map_cpu`. This means that alternating tasks are assigned to sockets 0 and 1:

```
# Round robin the tasks between the two sockets
$ srun  -N1 -n6 --cpu-bind=map_cpu:0,24,1,25,2,26 ./mpi
corona191  Task   0/  6 running on 1 CPUs: 0
corona191  Task   1/  6 running on 1 CPUs: 24
corona191  Task   2/  6 running on 1 CPUs: 1
corona191  Task   3/  6 running on 1 CPUs: 25
corona191  Task   4/  6 running on 1 CPUs: 2
corona191  Task   5/  6 running on 1 CPUs: 26
```

Hands-on exercise H: Setting affinity explicitly

On your AWS instance, run the `mpi` program with 6 tasks so that the tasks round robin places between the first three cores and the last three cores. In other words, the first, third, and fifth task are assigned to the first three cores, and the second, fourth, and sixth tasks are assigned to the last three cores. Make sure that tasks do not overlap places.

8. Bibliography and references

- The `hwloc` library
  - `hwloc-calc -h`
  - `man 7 hwloc`
  - hwloc project
  - hwloc tutorials
- The `Slurm` resource manager
  - `srun --cpu-bind=help`
  - `man srun`
  - Slurm srun
  - Slurm support for multi-core/multi-thread architectures
- The `mpibind` affinity library
  - Github repo
  - Reporting affinity