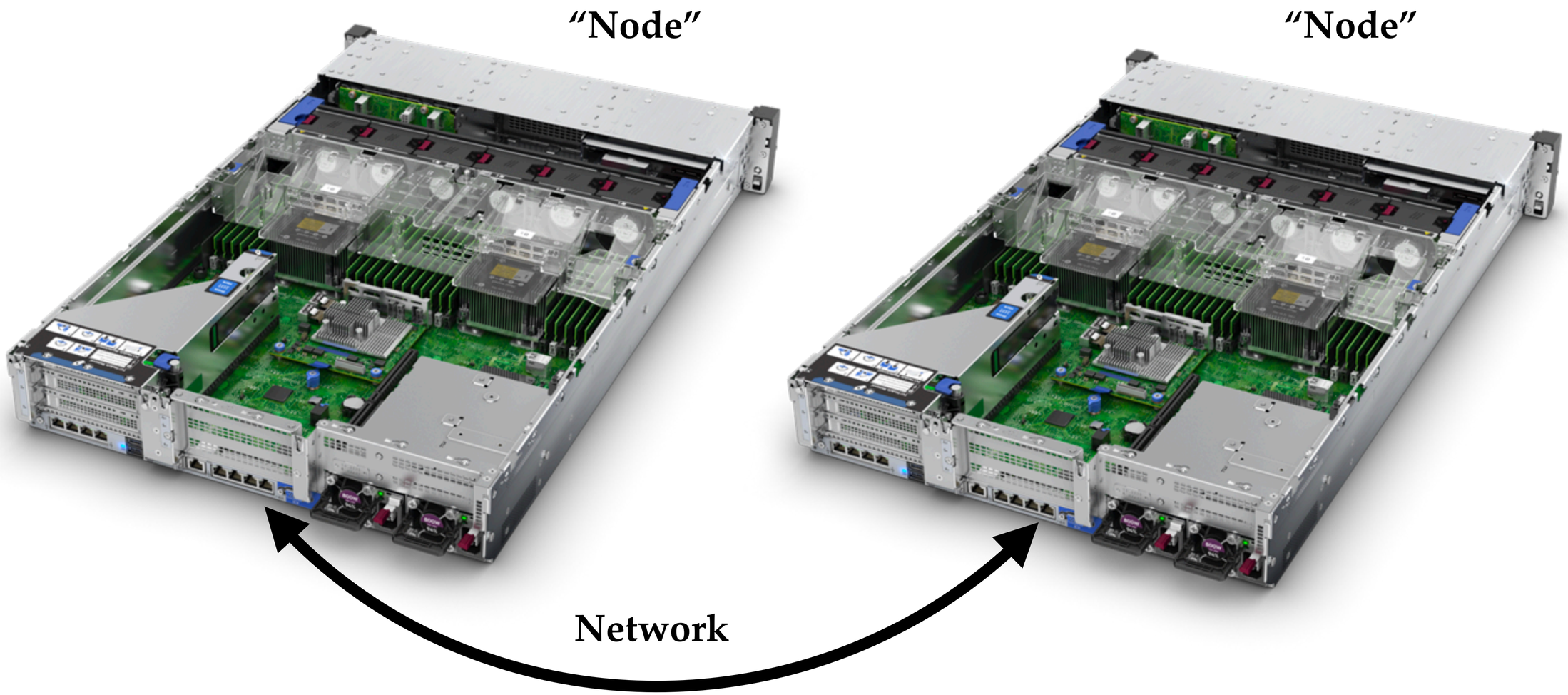
ASTR 660 / Class 14

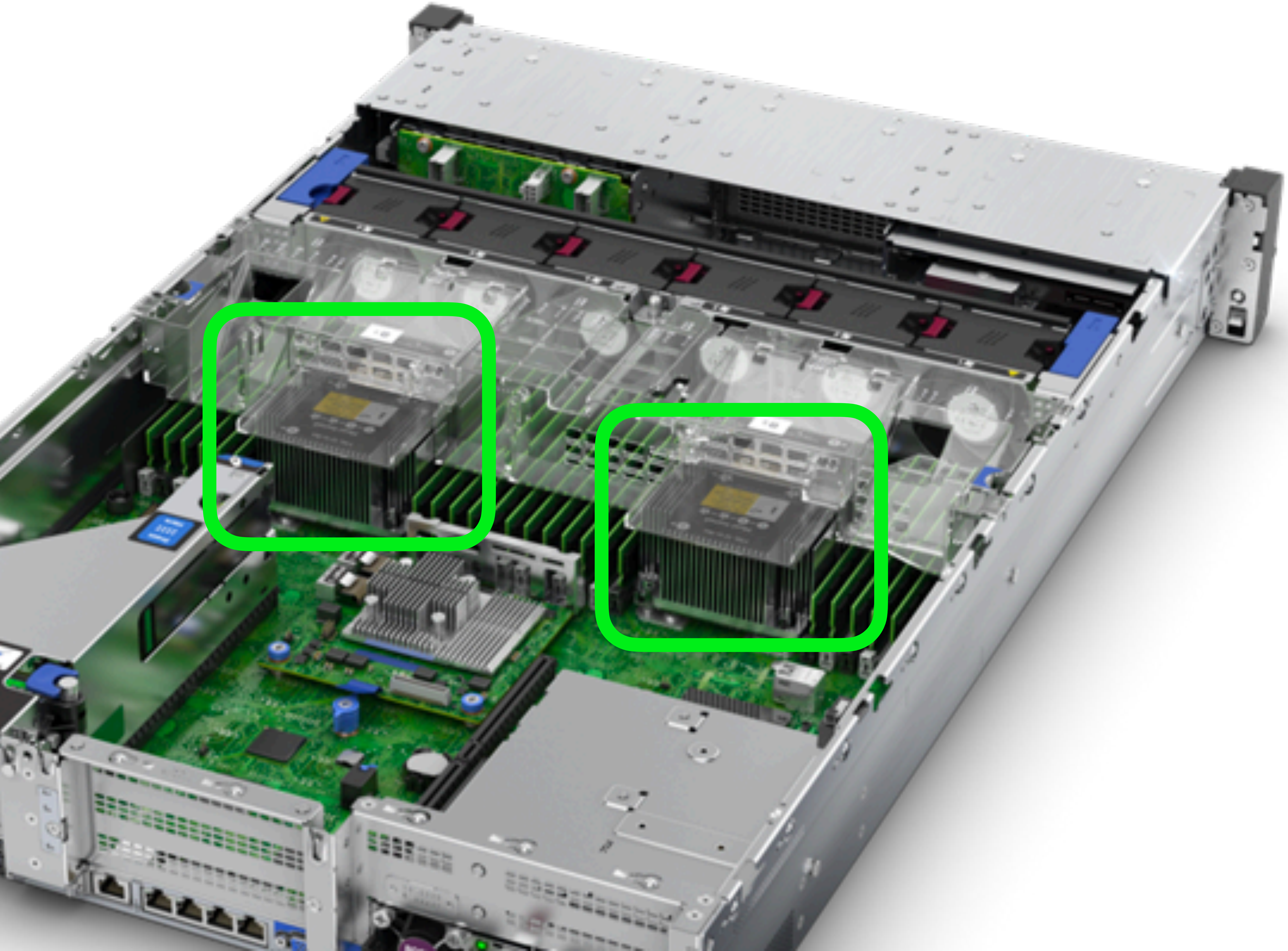
Parallel computing basics

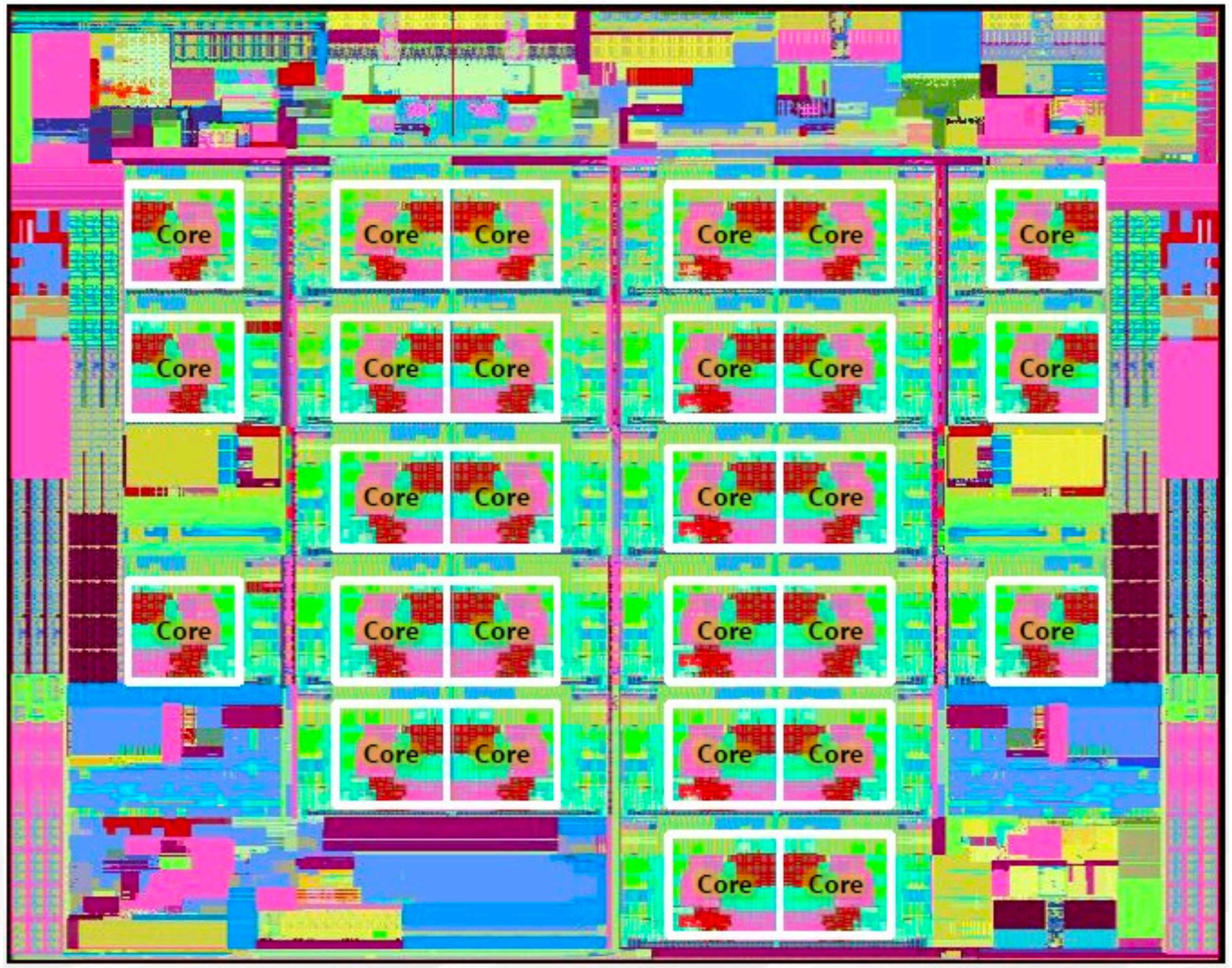
Photo:R. Foster / @TaiwanBirding

Hardware: nodes



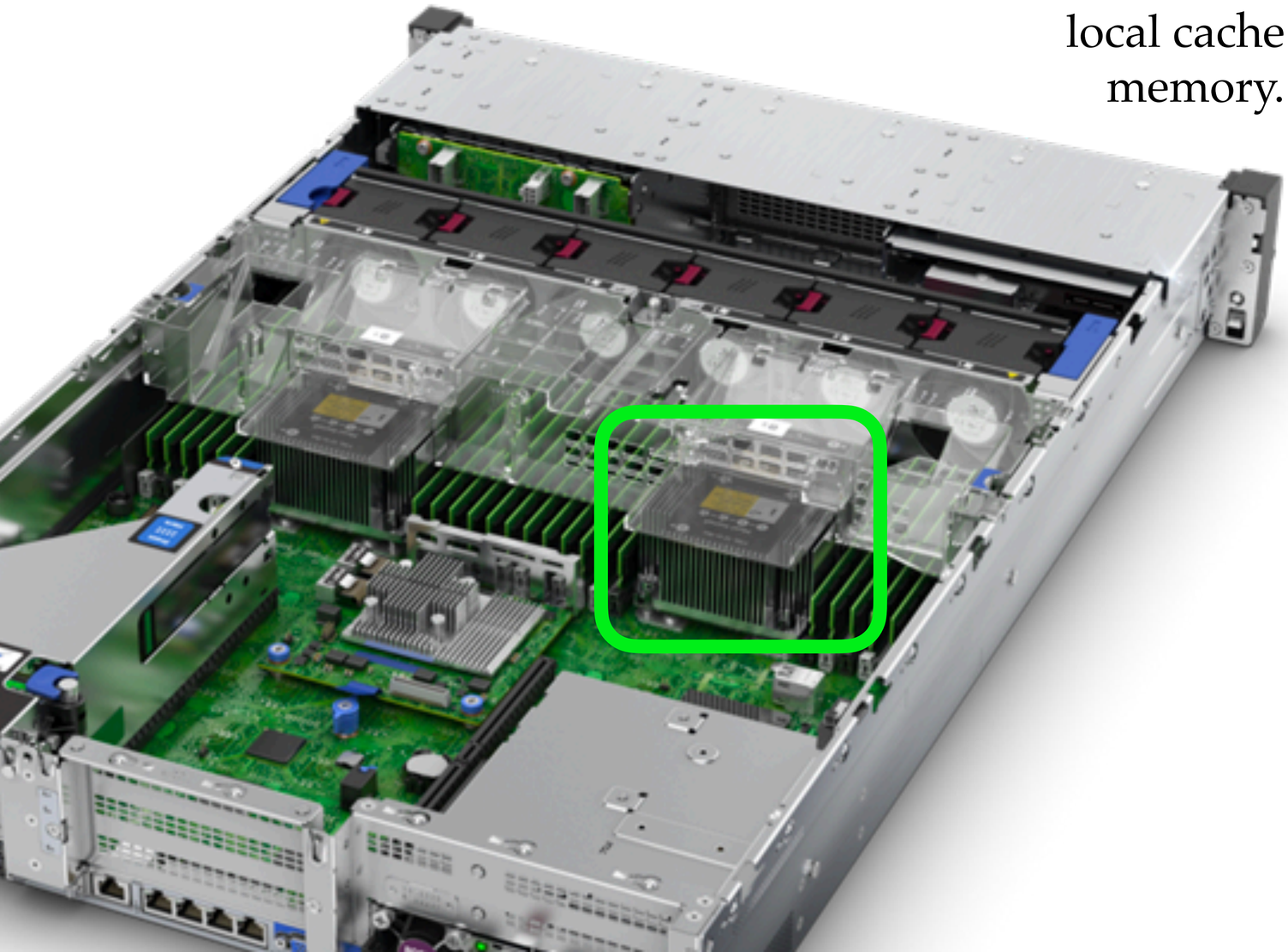
Sockets/cores



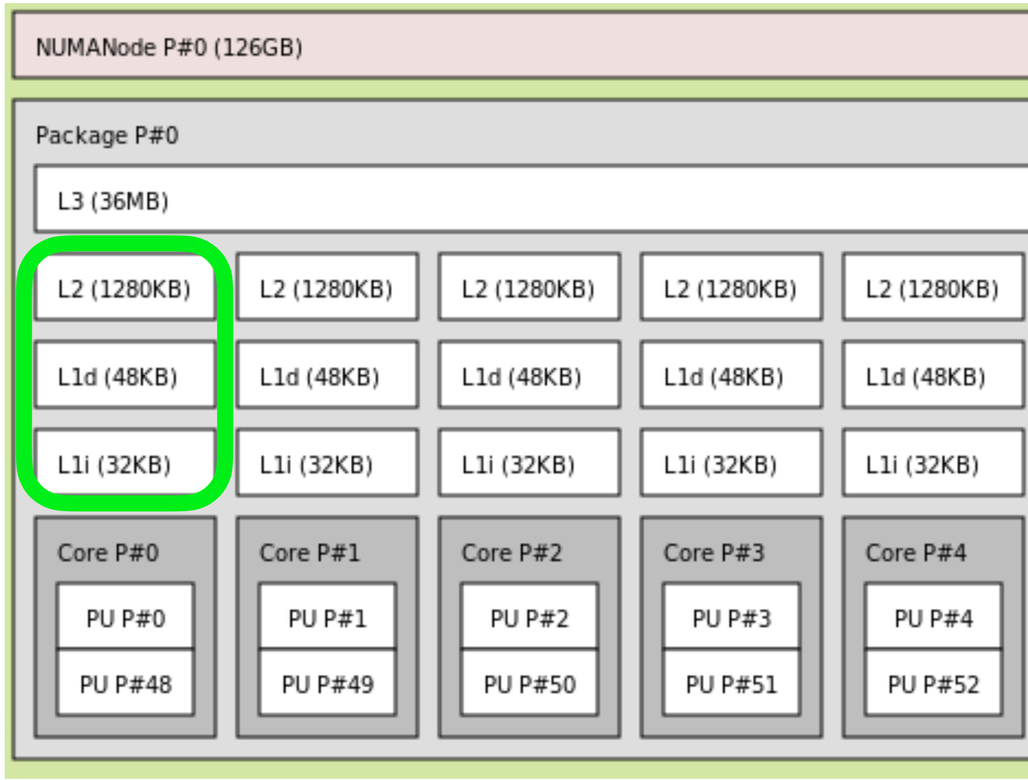


Example: 2 sockets, each with 28 cores.

Cores



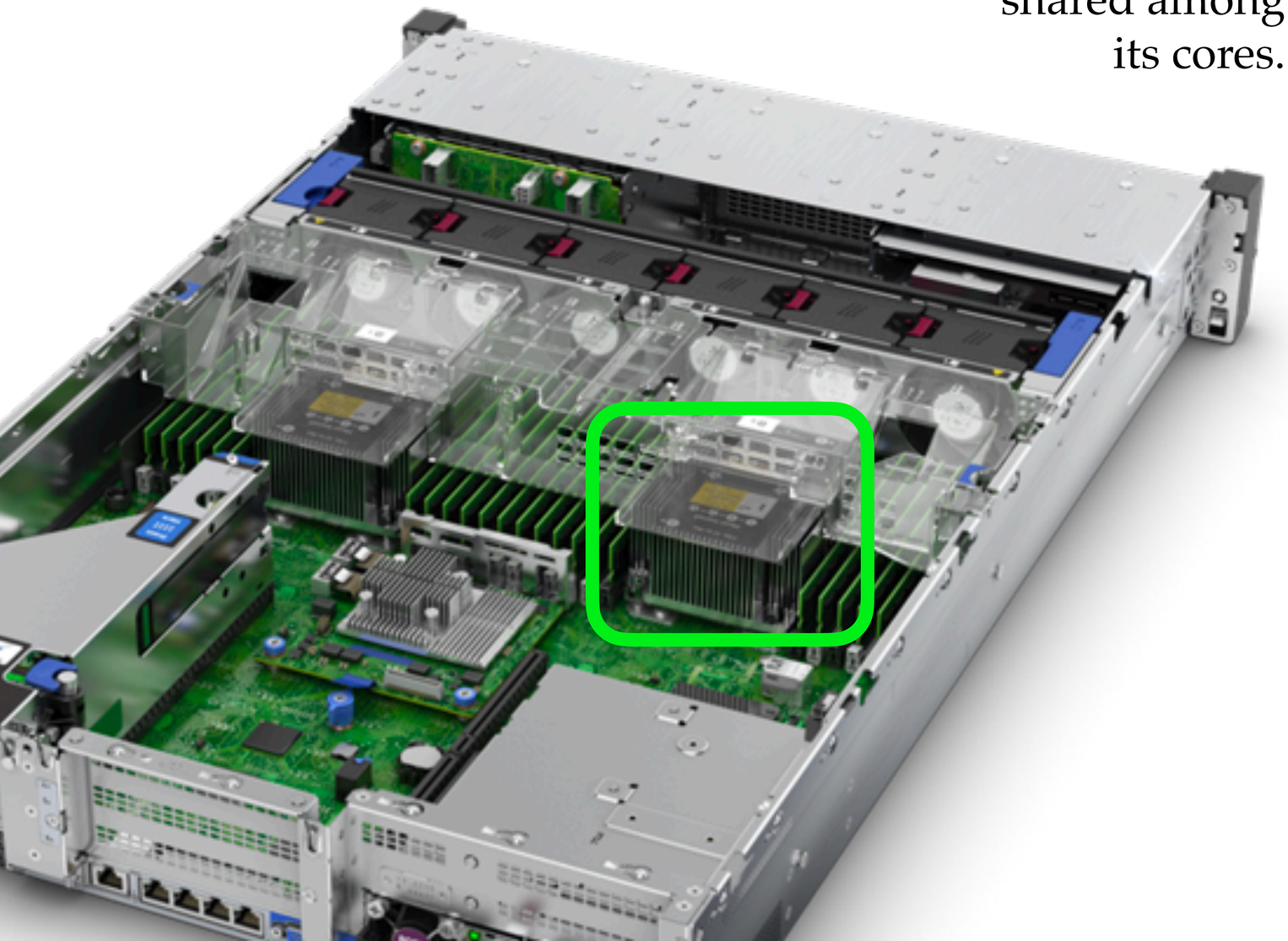
Each **core** has local cache memory.

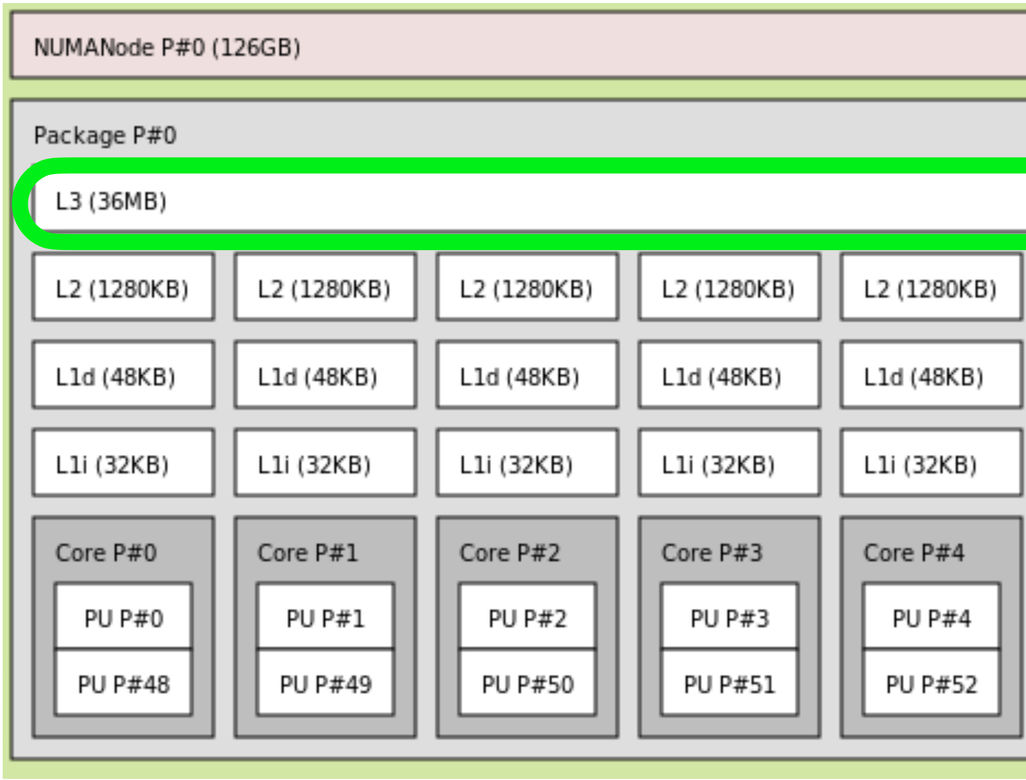


Cores operate on data **in memory**. The closer the data is to the core, the faster the operation.

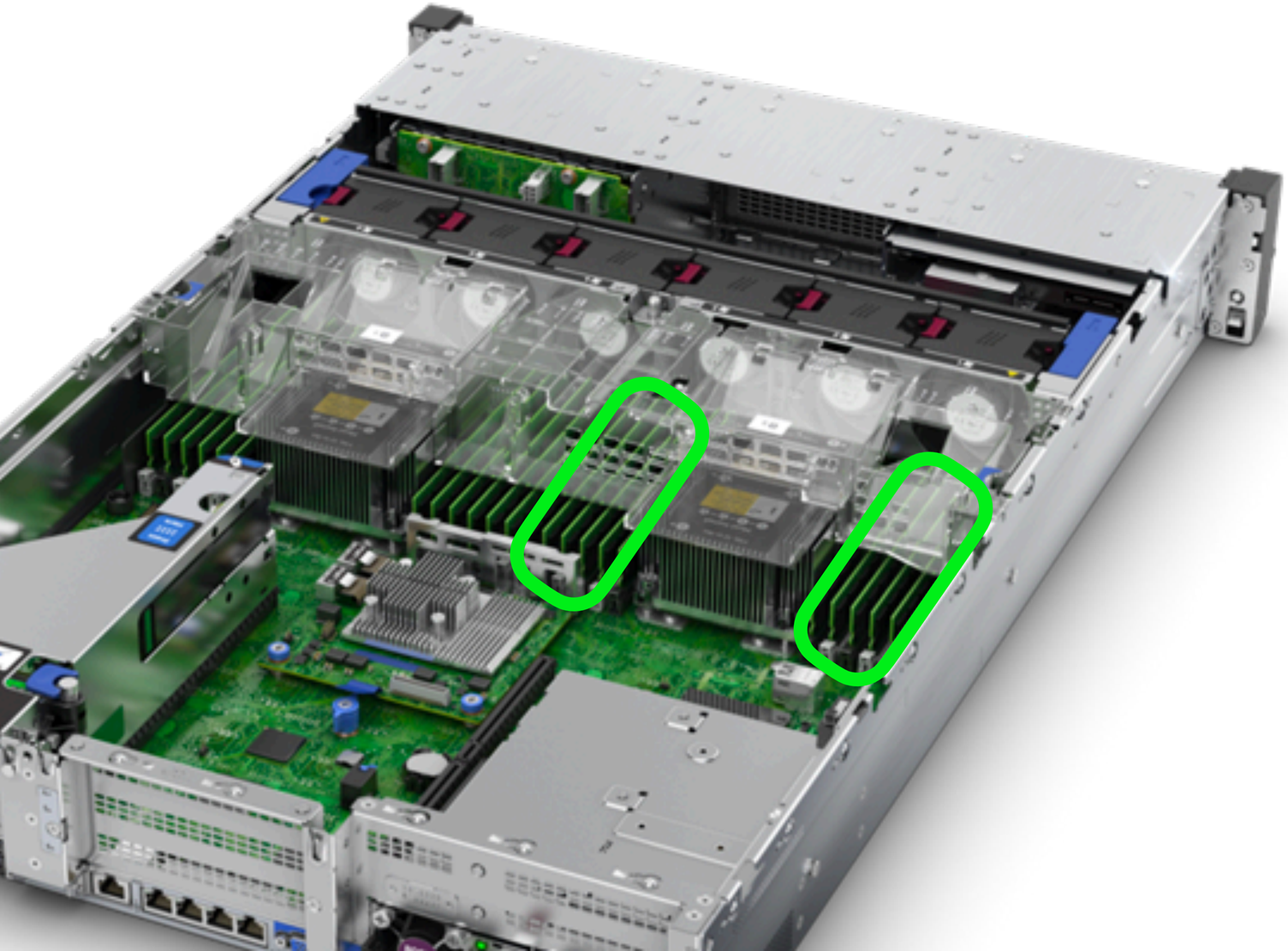
Socket

Each **socket** has L3 cache shared among its cores.





RAM



RAM is a lot larger and slower than cache memory.

256GB in each CICA node (except c01-c03); ~2GB per logical core.

Half the total RAM in a node is attached to **each socket**.

Access to RAM on the same socket is faster than access to RAM on the other socket.

Disks

/scratch

“Node”

“Fileserver” i.e. “disks”

/data/user

/lfs/data/user

Network

Disks

`/data/user`

- Slower, limited space.

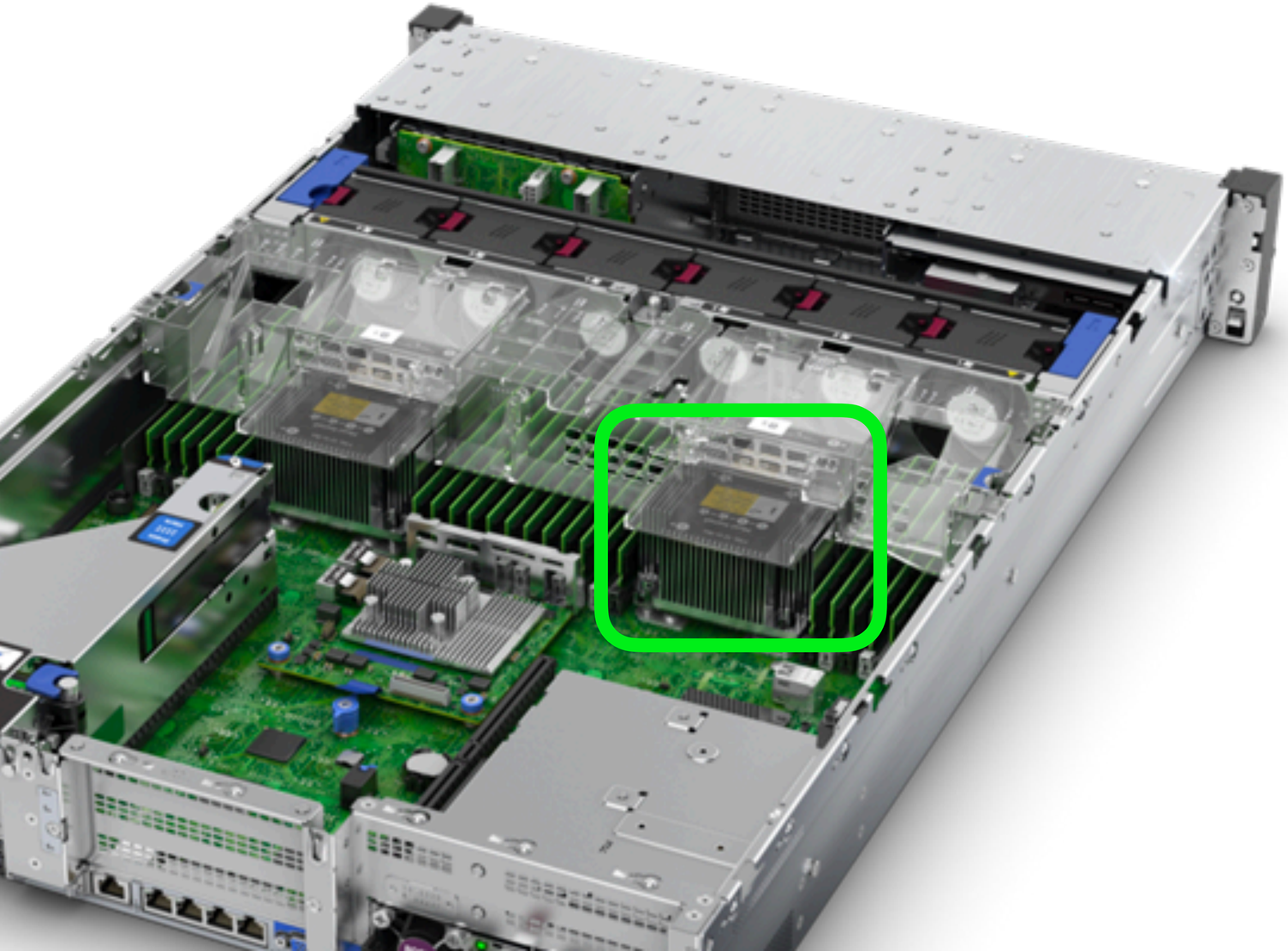
`/lfs/data/user`

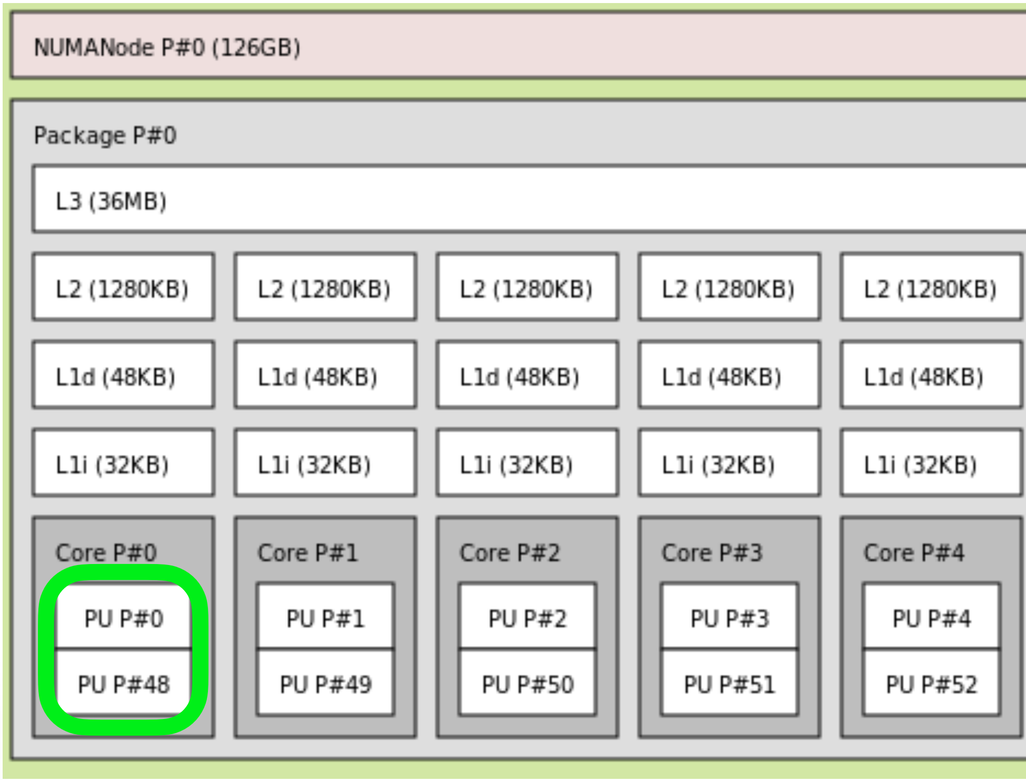
- Faster, larger — ask me if you need space here.

`/scratch`

- Very fast, very limited space, “private” to each node — need to copy data to/from other storage.

Hardware threads





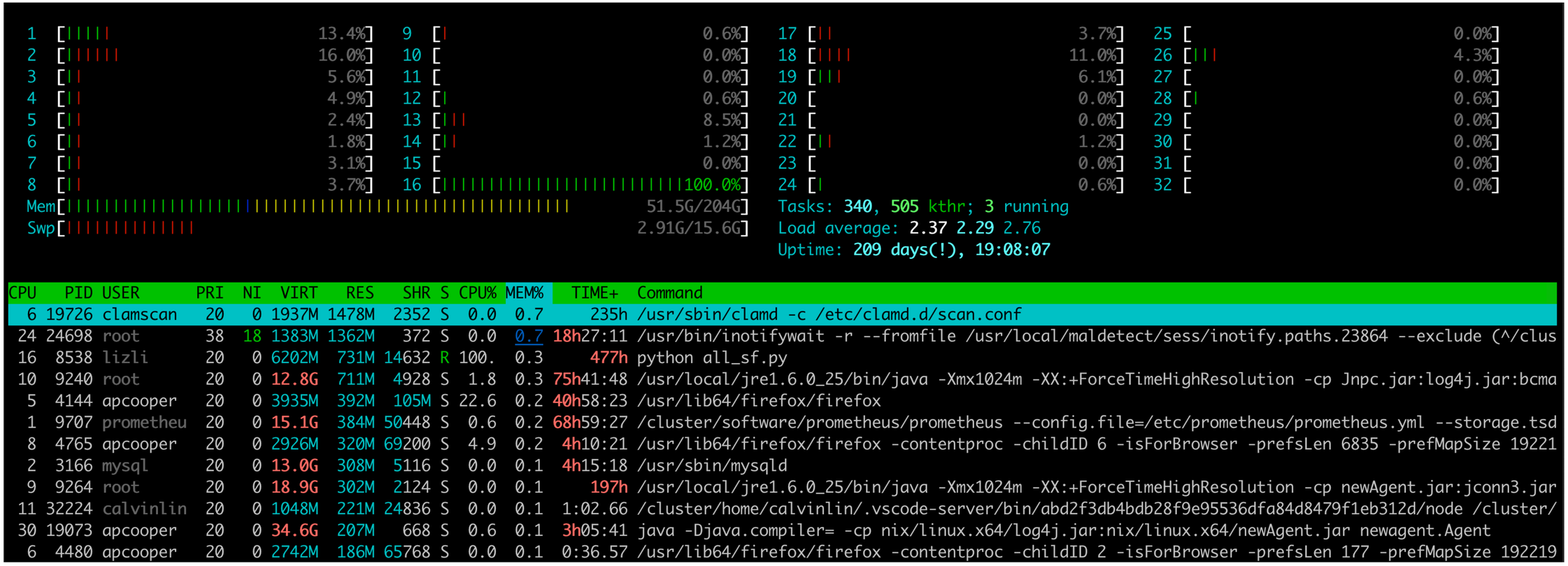
Each **physical** core can operate as two **logical** cores. This is called ***hyperthreading***.

Most CICA nodes have 96 logical cores.

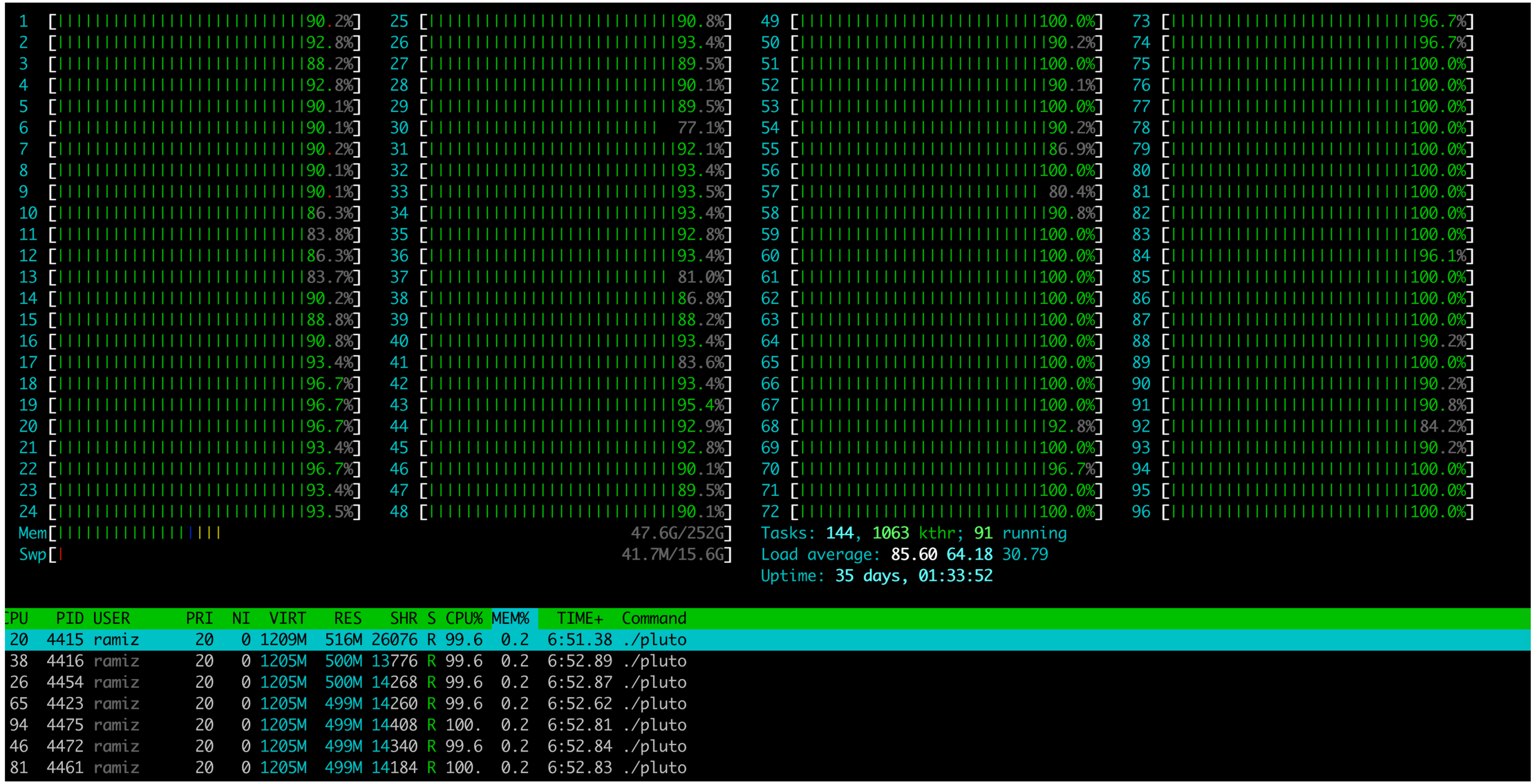
Cost: the core(s) run slower overall. The local cache is shared between the logical cores.

Software: processes

A process is the basic “worker” for computation. When you “run” a program, the operating system assigns a process (the series of instructions to be executed) to a core. Use `ssh` to access nodes and `top` or `htop` to see the processes running:



Software: processes



Software: processes

CPU	PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
4	4415	ramiz	20	0	1209M	516M	26076	R	99.7	0.2	7:46.27	./pluto
30	4416	ramiz	20	0	1205M	500M	13776	R	100.	0.2	7:47.79	./pluto
39	4454	ramiz	20	0	1205M	500M	14268	R	99.7	0.2	7:47.77	./pluto
65	4423	ramiz	20	0	1205M	499M	14260	R	99.7	0.2	7:47.51	./pluto
74	4475	ramiz	20	0	1205M	499M	14408	R	99.7	0.2	7:47.70	./pluto
31	4472	ramiz	20	0	1205M	499M	14340	R	99.7	0.2	7:47.74	./pluto
93	4461	ramiz	20	0	1205M	499M	14184	R	99.7	0.2	7:47.73	./pluto
46	4432	ramiz	20	0	1205M	498M	14172	R	99.7	0.2	7:47.76	./pluto
61	4450	ramiz	20	0	1205M	498M	13792	R	99.7	0.2	7:47.87	./pluto

Each of these processes has a process ID (PID) and is assigned to a CPU (a logical core). Each process has allocated some memory (RES).

A user can start as many processes as they want. If there are more processes waiting to run than there are CPUs available (*contention* / *oversubscription)*, the operating system will *schedule* the CPU time available to each process. This is not the same as batch queue scheduling (see later).

If the processes oversubscribe the CPU, everything will go slower, at least for some processes.

Parallel computation

Parallel algorithms also have important costs, including:

- Need to duplicate or move data between workers (memory/network overhead);
- Need to re-write algorithms in less natural forms (more things to go wrong).

Pipelines

It is often a good idea to split work into **pipelines** consisting of a series of logical stages, rather than trying to do everything in one code. As a very basic example:

- Make initial conditions;
- Run simulation;
- Post-process simulation (e.g. find all the groups of particles, compute statistics, do radiative transfer etc. etc.);
- Make plots.

In any serious work, always separate analysis from calculations.

Simple job array parallelism

Idea: take a serial code and run it in multiple simultaneous processes, without modification.

The code itself is purely serial in this case: only the execution is parallel. The speedup is a factor of the number of tasks that can be run simultaneously.

Different workers (in this case, processes) know **absolutely nothing** about each other and (ideally) **not sharing any resources**. There is no "monitor" to balance the work, only a "job manager" to assign it and check when it's done.

Sometimes called "embarrassingly parallel". The idea is simple, but there is nothing embarrassing about it.

If you can't use this technique effectively, you will never be able to write sophisticated parallel algorithms!

Slurm job arrays

Details: https://slurm.schedmd.com/job_array.html

In the Slurm batch queue system, pass the array argument to `sbatch`, either as a directive in the job script:

```
#SBATCH --array=0-9
```

or on the command line:

```
sbatch --array=0-9
```

These examples would create an array of 10 jobs, all running *exactly the same* code, **independently**. The **only** difference is in the environment variables that each process can see. Specifically, the environment variable `SLURM_ARRAY_TASK_ID` contains a different number (in this case, 0 to 10) for each process.

The rest is up to you!

Slurm job arrays

Each job in the array is scheduled and run separately by Slurm, using the resources you ask for.

For example, if your script asks for 1 hour of CPU time and 20G of RAM, each job in the array will be scheduled with those resources.

Generally speaking, job arrays suit large numbers of small jobs, each of which needs relatively few resources.

Slurm job arrays: useful tips

Limit the number of jobs in the array that can run at the same time by adding %N to the array specification, where N is the maximum number of jobs to run at once. The main point of this is to **be nice to other users** and **avoid overwhelming shared resources** (e.g. storage bandwidth). *It is not needed to avoid basic contention between processes, because that is naturally handled by the batch queue.*

```
--array=0-200%10
```

The indices do not have to be an unbroken sequence starting at zero. For example:

```
sbatch --array=88,99,888,999
```

Write separate slurm log files for each job in the array (almost certainly you want to do this) by using **wildcards** in the SBATCH directives. For example:

```
#SBATCH -o job.%A.%a.out
#SBATCH -e job.%A.%a.err
```

%A is the job array's master job allocation number (a number that is the same for all elements). %a is the job array index number (i.e. the number that is different for each element). See also https://slurm.schedmd.com/sbatch.html.

Slurm job arrays

Consider how you would use job arrays to do the following:

- Process 200 different initial conditions files, where:

  (a) The files have names “`ics0.dat`”, “`ics1.dat`”, “`ics2.dat`”, …, “`icsN.dat`”;

  (b) The files have names like “`ics_alpha7_viscous_a`”,”`ics_alpha7_viscous_b`”, “`ics_alpha7_inviscid`”, “`ics_alpha6.6_viscous`”, … etc.

- Run a code that generates a result based on a random number, using 200 different random numbers, with reproducible results.

Limitations of job arrays

The sequence in which the tasks are executed is semi-random and their lifetimes are determined only by their local work, so there is no useful way the tasks can communicate with each other.

Having multiple independent processes access the **same file** is usually problematic.

> Writing is particularly tricky (though possible). Care is needed to manage file "locks". Usually the best solution is to have each process write its own output.
>
> Reading is easier (the sequence in which each process gets data doesn't matters), but can be slow for the same reason. Large slowdowns can result from multiple processes "fighting" each other.

Storage ("disk") has very limited bandwidth. Too many processes accessing *different* files on the *same* storage volume will also be slow (a good reason to limit the number of simultaneous tasks).

In-process parallelism

Processes can create (“spawn”) other processes. This allows the programmer (rather than the execution environment) to deal with the subdivision of work into separate tasks.

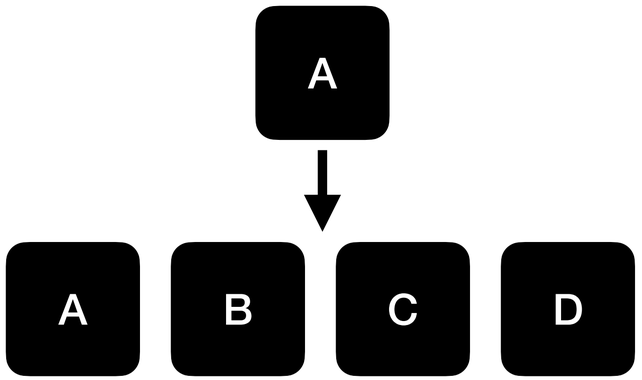
This is the basis for all parallel algorithms.

Obviously, it is only efficient if there are spare resources to execute the extra processes. If not, they will just fight each other, and their parent, for resources.

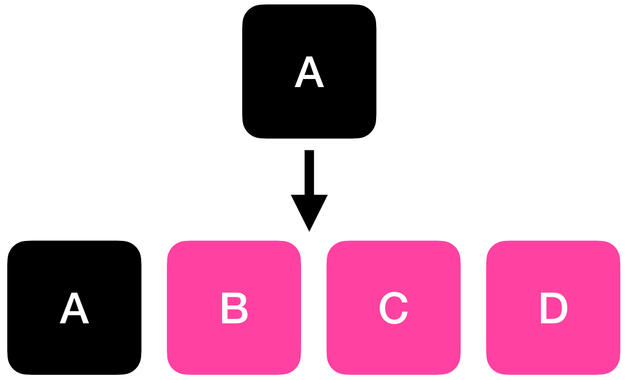
In-process parallelism

Imagine process *A* is doing some work that it wants to split into 4 chunks that can be processed at the same time. There are three options:

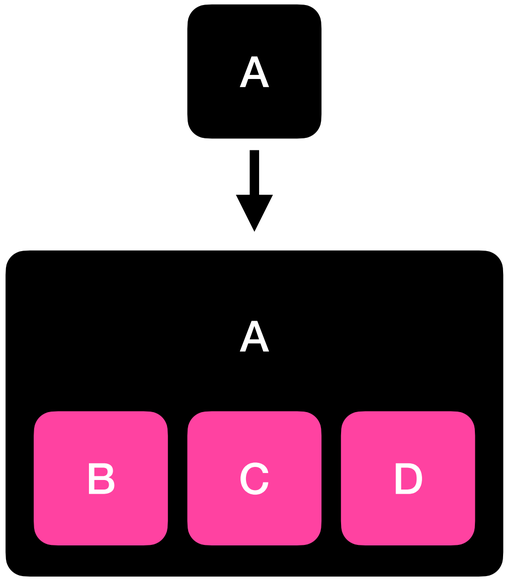
Full processes



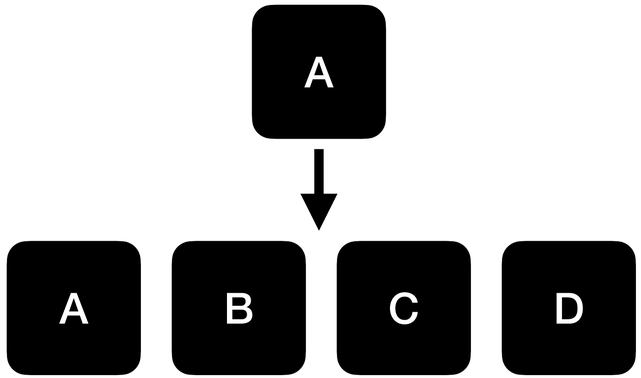
Sub-processes



Threads



Full processes

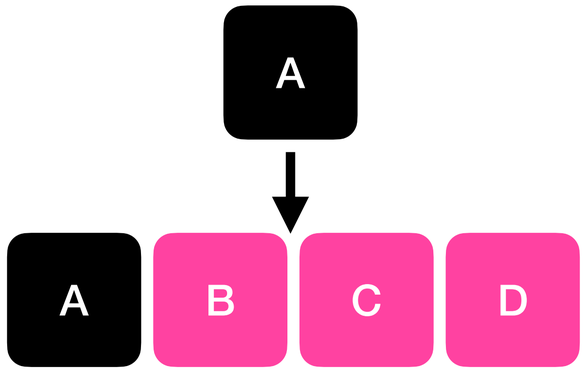


All processes are equivalent and independent. They know nothing, directly, about each other's memory. In principle they don't even have to be on the same node.

They can communicate using **interprocess communication**, i.e. via the operating system (because A knows the PIDs of B, C and D). This is hard to manage efficiently.

This is similar to a job array. It is what happens when you have your program run shell commands through "exec"-like calls; most languages can do this.

Sub-processes (in Python)



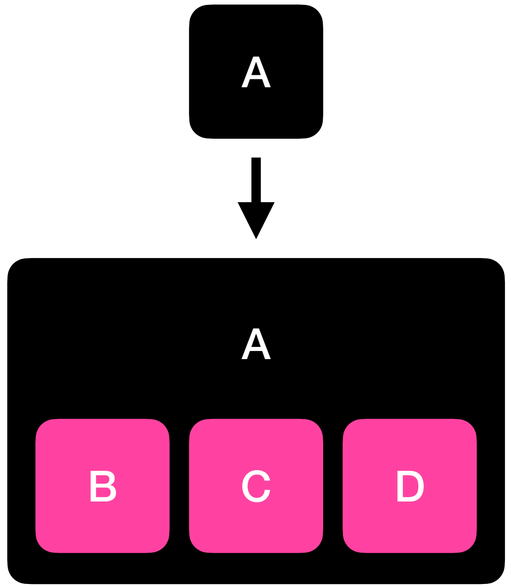
This is almost identical to the previous case, but with the extra concept of the new processes being “tied” to or managed by the parent in some way (using IPC behind the scenes).

The OS still sees them as full processes with their own resources.

In Python, this is implemented by the `os.subprocess` module. This lets you run external programs and store their console output inside your Python program.

Python’s `multiprocessing` module takes the concept even further.

Threads



Threads (sometimes called lightweight processes) are fundamentally different to full processes because they **share** the resources of their parent.

They communicated through shared memory, rather than IPC (although note that different processes can also communicate through **shared memory**).

In this case, **one** process is really using **many cores** simultaneously.

In (standard) Python there is a complication to this which means that subprocesses are used more often than threads.

Software: threads

We'll look more at practical uses of multiprocessing and threads in the next lecture.

The purpose here is to understand how these ideas map to the libraries you are likely to meet when running other codes (e.g. for your projects):

The Message Passing Interface (MPI)

Open Multi-Processing (OpenMP)

MPI

The MPI library enables data to be exchanged efficiently between independent processes (called ***tasks***), regardless of where they happen to be running.

In particular, it allows tasks to communicate with each other over a network or through shared memory (using the same interface). Even if using shared memory, the tasks communicated by "message passing".

To do this, it also provides a runtime environment that keeps track of all the processes that want to talk to each other. This means you need to start MPI programs with a special command, `mpirun`.

`module load openmpi/4.0.7` will give access to the MPI libraries, compiler and runtime commands.

`mpirun -np 16 ./myprogram` will start 16 (almost) identical processes that will be able to pass data to each other (*communicate*) using MPI functions.

MPI and slurm

```
#SBATCH --nodes=2
#SBATCH --ntasks_per_node=16
#SBATCH --mem_per_task=16G
# … etc.
module load openmpi/4.0.7

mpirun -np 32 ./myprogram
```

Example: You want to run 32 MPI tasks spread across 2 nodes, each with 16GB of RAM (so 16x16 = 256GB per node, 100% of the available RAM)

The `SBATCH` directives specify the total number of tasks. Note that you don't need "`--ntasks`", because the total is already specified by the other two options.

You tell mpirun about the **total** number of tasks you want. MPI and Slurm will sort out where those tasks will run (and how they should talk to each other, e.g. shared memory if they are on the same node)

It's easy to get "`ntasks_per_node`" this confused with a different option, "`cpus_per_task`" (see later slides). Remember, each MPI **task** is a single, independent **process**.

MPI and compilers

`module load openmpi/4.0.7` will give access to the MPI libraries, **compiler** and runtime commands.

The MPI compiler is really just a collection of wrappers around gcc. These have names like **mpicc** (for the C MPI compiler), **mpicxx** (for the C++ MPI coliler), and **mpif90** (for Fortran).

Codes that use MPI will probably assume these compilers exist, so you need the (same) openmpi module loaded when **compiling** (for the compiler) and when **running** (for the runtime commands, and perhaps the dynamic libraries).

OpenMP

OpenMP (not to be confused with OpenMPI) is a series of compiler extensions that enable **multithreaded** execution of code.

For example, if you find your code spends a lot of time running through a particular loop, you can “decorate” that part of the code with instructions that tell the compiler to use OpenMP to split the loop over multiple threads.

This is very different to MPI. In this case there is no “message passing”. Everything happens in shared memory under the control of a single process.

When that process enters a multi-threaded part of the code, it will spawn however many threads it needs, and destroy them afterwards.

OpenMP

OpenMP does not need a special runtime environment; everything is compiled into the program itself, linked against the openmp library.

```
#SBATCH --nodes=1
#SBATCH --ntasks_per_node=1
#SBATCH --cpus_per_task=32
#SBATCH --mem_per_task=256G
# … etc.

srun ./myprogram
```

Example: You want to run an OpenMP code with access to 32 CPUs and 256GB of RAM. This is only a single task. Notice we now set `cpus_per_task`!

We use `srun` here: this is the standard command to start non-MPI programs under Slurm, and is not related to OpenMP.

OpenMP Warning

There is nothing stopping an OpenMP program from creating as many threads as it wants. This can lead to massive slowdowns once the number of **active** threads exceeds the number of cores that can run them.

Sometimes the number of threads is a command line parameter to the program. Sometimes (quite often) the program will be written to detect the number of cores on the machine, and use the same number of threads.

This may not have the expected outcome when run under Slurm — for example, the program thinks it can use all 96 logical cores on the machine, so it starts 96 threads, but Slurm still restricts those threads to the 32 cores requested by the user.

The environment variable `OMP_NUM_THREADS` can be set to control (usually, but not strictly) the maximum number of threads used by OpenMP codes.

OpenMP thread limit

```
#SBATCH --nodes=1
#SBATCH --ntasks_per_node=1
#SBATCH --cpus_per_task=32
#SBATCH --mem_per_task=256G
# … etc.

export OMP_NUM_THREADS=32

srun ./myprogram
```

Example: You want to run an OpenMP code with access to 32 CPUs and 256GB of RAM. This is only a single task. Notice we now set `cpus_per_task`!

Limit the number of threads using the OMP environment variable.

We use `srun` here: this is the standard command to start non-MPI programs under Slurm, and is not related to OpenMP.

OpenMP thread limit

```
#SBATCH --nodes=1
#SBATCH --ntasks_per_node=1
#SBATCH --cpus_per_task=32
#SBATCH --mem_per_task=256G
# … etc.

module load slurm_limit_threads

srun ./myprogram
```

On CICA, for convenience, we provide a special module that sets several environment variables like `OMP_NUM_THREADS` to the number of cores visible to the job through slurm (in this example, 32).

See `module show slurm_limit_threads` for details.

Screen

When you log out of ssh, you lose your current session. The `screen` command allows you to “detach” from sessions on the remote server and “attach” to them again.

If you run non-interactive jobs through the slurm system, you should not need to use screen for running anything. However, it might be helpful for keeping editor or compiler sessions alive. If you are running interactive jobs in Slurm, it might help to start them from screen.

You can learn about screen for yourself. Some useful commands:

`screen -list` (list the names of all your screen sessions)

`screen -S name` (start a new screen with a name you give it)

`screen -dr name` (detach the named screen from whatever terminal is attached to it, then attach from the current terminal)

Photo: R. Foster @TaiwanBirding

Taiwan Bullfinch
台灣灰鷽

ASTR 660 / Class 14

Reports

Report-writing tips

Imagine someone else with similar knowledge to you is going to take over your project, starting from wherever you got up to.

Write with the idea of helping such a person, who would rely on the information in your report. What do they need to know?

What is the best way to communicate? For example, no-one wants to read ten pages of screenshots or plots without captions.

Report-writing tips

I'm much more interested in the **process** than the **result**.

Non-exhaustive list of things I'd like to see:

- An explanation of the background to the problem;
- An explanation of the relevant computational challenges;
- Your intended plan (even if things didn't work out that way);
- An explanation of the tools you used and why;
- What you actually did;
- Where you got stuck, why, what you tried to make progress;
- What you would try to do if you had more time.
- In your code (via Github): logical organisation, readability, comments.

Report-writing tips

Some things I don't want to see:

- ChatGPT waffle (i.e. lots of text that shows no evidence of **your own** critical thought);
- "I got stuck, this is the error message … " followed by a screenshot of some cryptic compiler error.

I will give help with technical trouble up to one week before the deadline (by which stage you should be writing up, not computing).

After than, out of fairness, I will not give any technical help.

Grade philosophy

For a basic pass (B+) I'm looking for evidence that you invested some reasonable time and **effort**, and tried your best.

For the B+/A- boundary, I'm looking for evidence of **independence —** having a plan, working step by step, being critical, understanding what you're trying to do and why.

For the A-/A boundary, I'm looking for evidence of **competence** — getting something done and presenting it in a clear and readable way.

For the A/A+ boundary, I'm looking for evidence of **insight** — not only did you do the work, but you spent some time to explore the result and the process, learn something interesting (to you) and draw conclusions (for example, how it could be done better).

Report-writing tips

Independence (the B+/A- difference) does not mean “not asking for any help”.

It does mean asking for appropriate help at an appropriate time.

For example: “The code won’t compile, please help” two days (or even one week) before the deadline is neither an appropriate question (no detail of what you think the problem is or what you have tried) nor appropriate timing.