Intro to Parallel Programming

Scientific Computing II

Fundación Universitaria Konrad Lorenz

2023-II

Concurrency

When programming in parallel, we require to distribute work across n cores.

- All these cores need something to do.
- We need to find parts that can done independently, and therefore on different cores concurrently.
- Ideally, the order of execution should not matter either.
- However, data dependencies limit concurrency.

Supercomputer Architectures

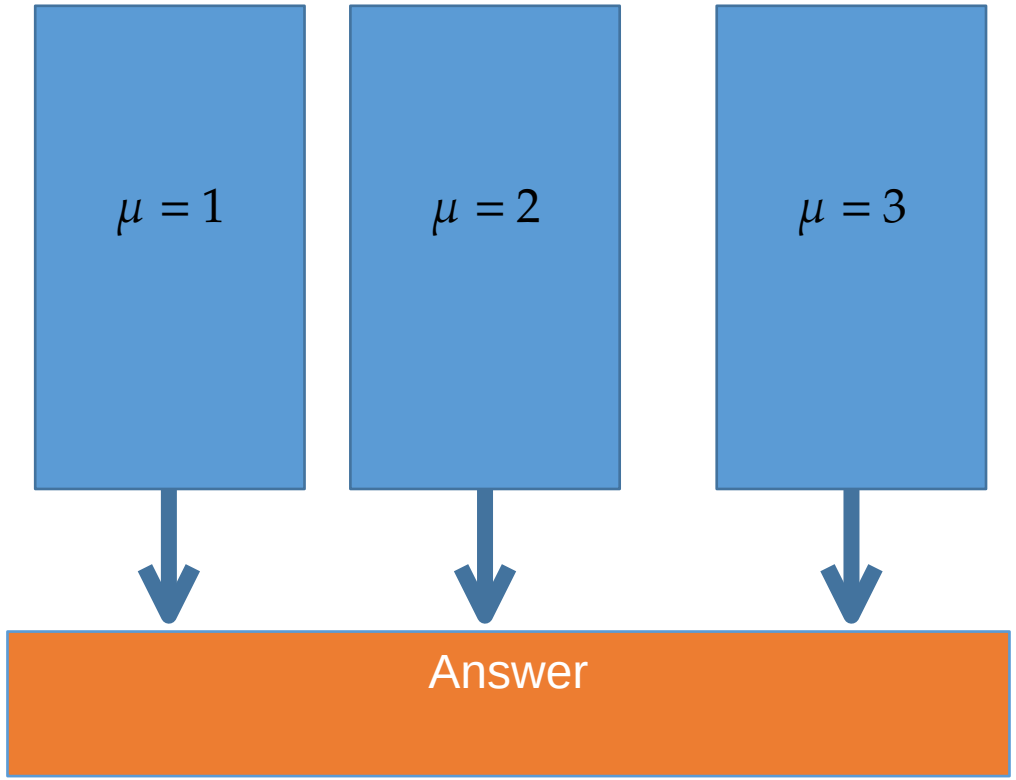
Supercomputer architectures come in a number of different types:

- **Clusters**, or distributed-memory machines, are in essence a bunch of desktops linked together by a network (”interconnect”). Easy and “cheap”.
- **Multi-core machines**, or shared-memory machines, are a collection of processors that can see and use the same memory. Limited number of cores, and much more expensive when the machine is large.
- **Accelerator machines**, are machines which contain an “off-host” accelerator, such as a GPGPU or Xeon Phi, that is used for computation. Quite fast, but complicated to program.
- **Vector machines** were the early supercomputers. Very expensive, especially at scale. These days most chips have some low-level vectorization, but you rarely need to worry about it.

Most supercomputers are a hybrid combo of these different architectures.

Best case scenario

- Suppose the aim is to get results from a model as the parameter varies.
- We can run the serial program on each processor at the same time.
- Thus, we get “more” done.



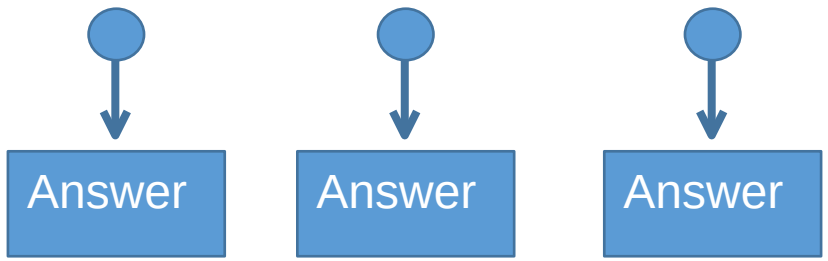
Throughput/Rendimiento

- Tasks per unit of time:

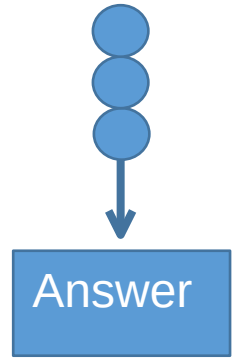
$$throughput = H = \frac{N}{T}$$

N is the number of tasks, T is the total time.

- Maximizing H means that you can do as much as possible.
- Independent tasks: using P processors increases H by a factor of P.



$$T = NT_1/P \qquad H = P/T_1$$



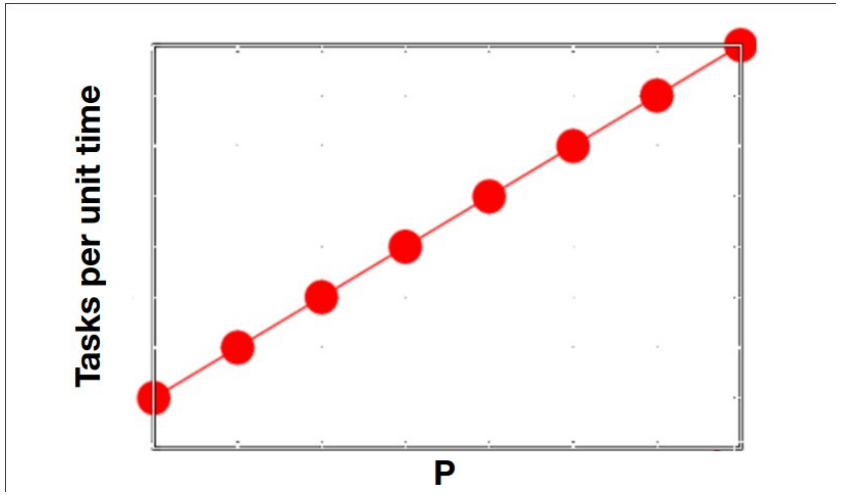
$$T = NT_1$$
$$H = 1/T_1$$

Scaling - Throughput

- How a given problem's throughput scales as processor number increases is called "strong scaling".
- In the previous case, we had linear scaling:

$$H \propto P$$

- This is called perfect scaling. These are called "embarrassingly parallel" jobs.

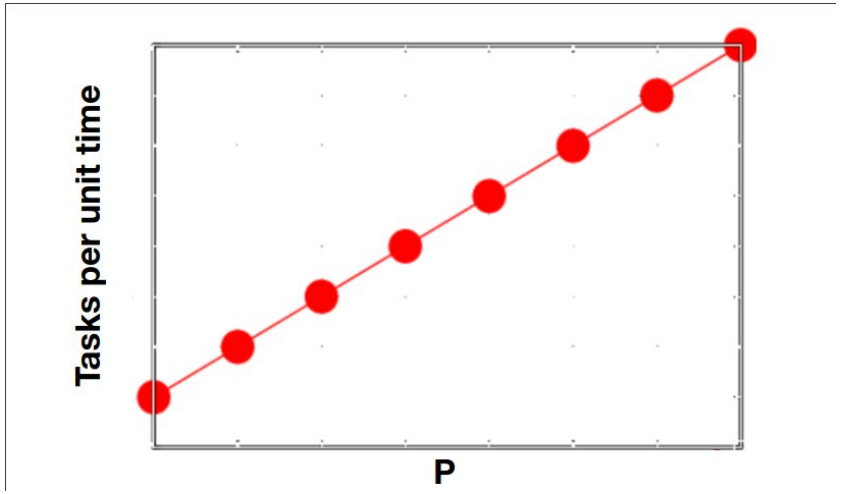


Scaling - Speedup

- Speedup: how much faster the problem is solved as processor number increases.
- This is measured by the serial time divided by the parallel time.

$$S = \frac{T_{serial}}{T(P)}.$$

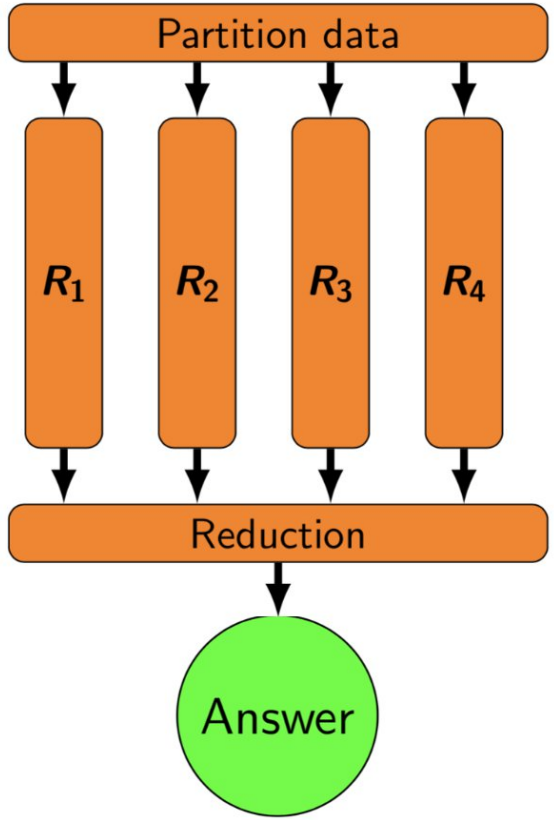
- For embarrassingly parallel applications, $S \propto P$: linear speedup.



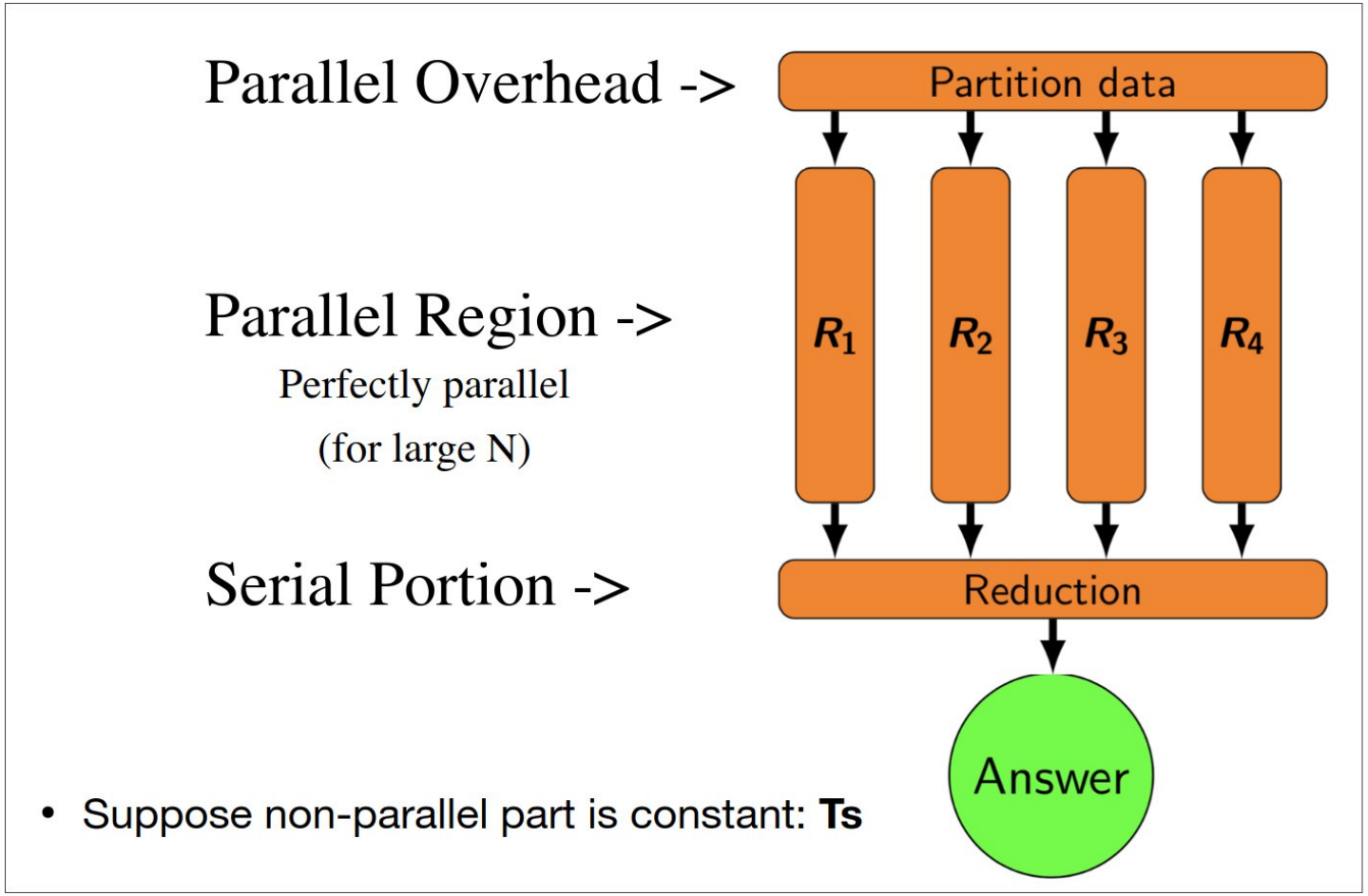
Non-ideal cases

- Suppose we want to integrate some tabulated experimental data.
- Integration can be split up, so different regions are summed by each processor.

- Non-ideal:
  - We first need to get data to each processor.
  - At the end, we need to bring together all the sums: “**reduction**”.



Non-ideal cases



- Suppose non-parallel part is constant: **Ts**

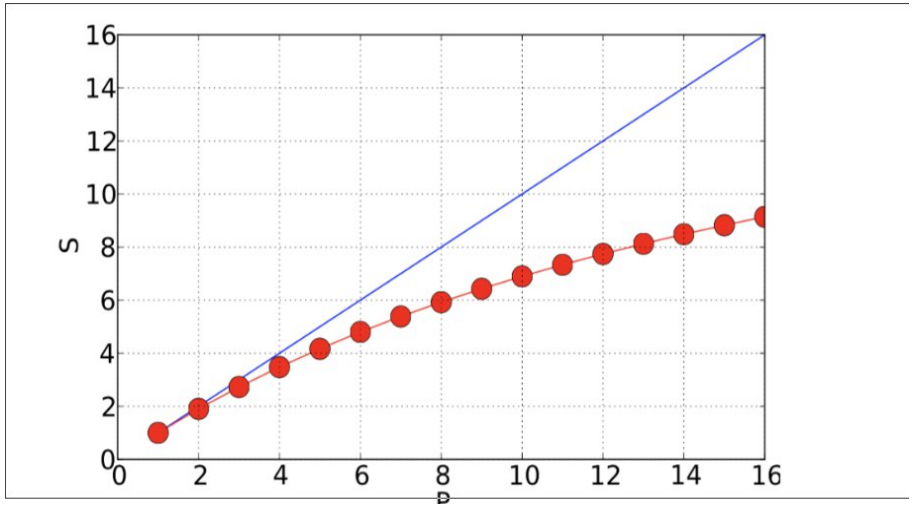
Amdahl's law

- **Speedup** (without parallel overhead):

$$S = \frac{T_{serial}}{T(P)} = \frac{NT_1 + T_s}{NT_1/P + T_s}.$$

or, calling $f = T_s/(T_s + NT_1)$ the serial fraction of the program,

$$S = \frac{T_{serial}}{T(P)} = \frac{1}{f + (1-f)/P}.$$



The serial part dominates asymptotically. The speedup is limited, no matter the size of P.

Example of Amdahl's law

- Suppose your code consists of a portion which is serial, and a portion that can be parallelized. Suppose further that, when run on a single processor,
  - The serial portion takes one hour to run.
  - The parallel portion takes nineteen hours to run.
- Even if you throw an infinite number of processors at the parallel part of the problem, the code will never urn faster than 1 hour, since that is the amount of time the serial part needs to complete

The goal is to structure your program to minimize the serial portions of the code.

Scaling efficiency

Speedup compared to ideal factor:

$$\boldsymbol{Efficiency} = \frac{S}{P}.$$

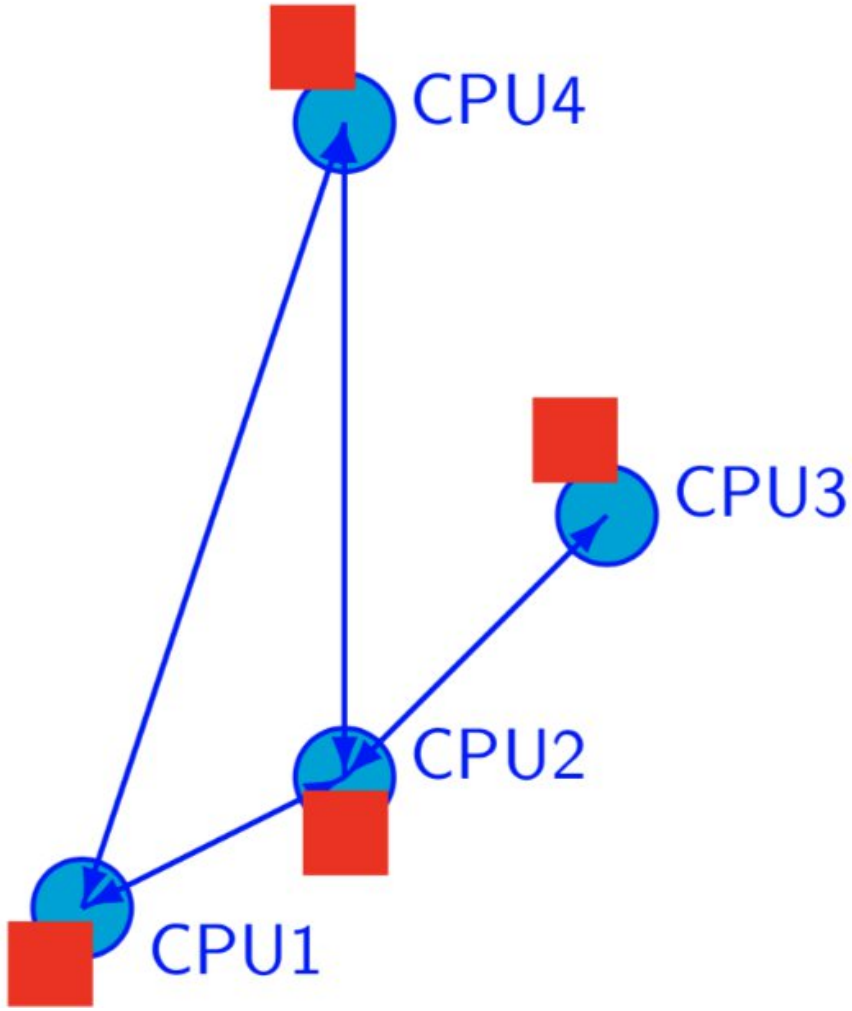
This will decrease for larger P, except for embarrassingly parallel problems.

$$Efficiency \sim \frac{1}{fP} \to 0 \text{ when } P \to \infty.$$

- You cannot get 100% efficiency in any non-trivial problem
- All you can aim is to make the efficiency as high as possible.

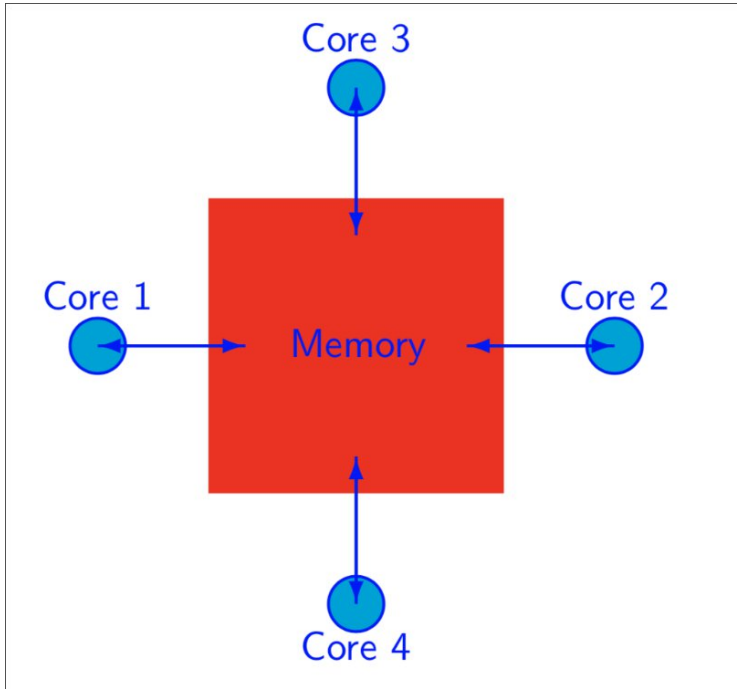
Distributed Memory: Clusters

- Each processor is independent! Programs run on separate processors, communication with each other when necessary.
- Each processor has its own memory. Whenever it needs data from another processor, that processor needs to send it.
- All communication must be **hand-coded**: harder to program.
- **MPI** programming is commonly used in this scenario.



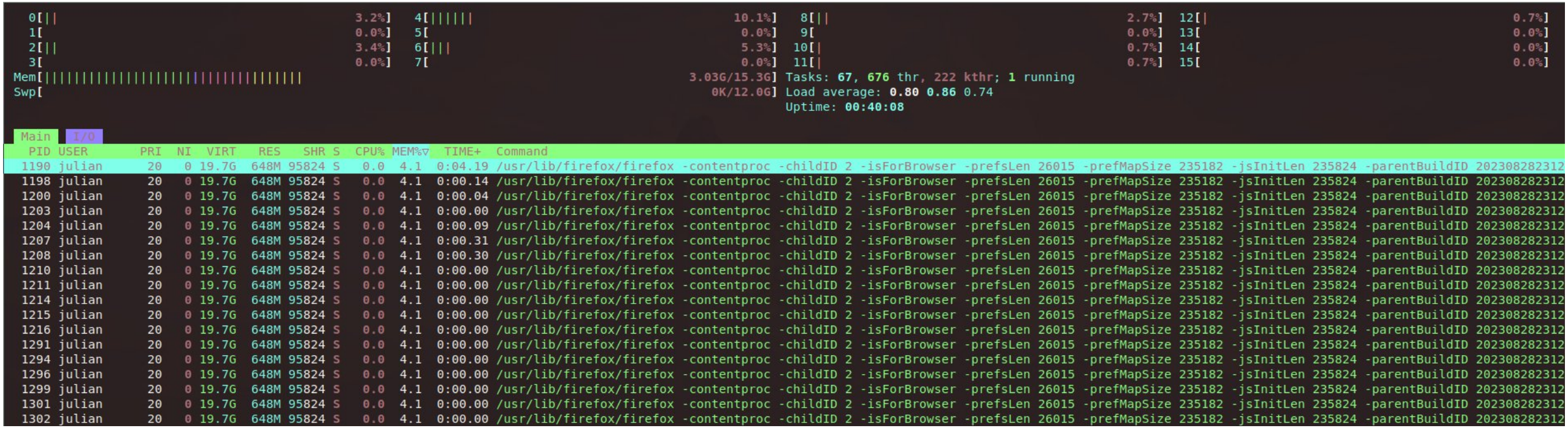
Shared Memory: Clusters

- Different processors acting on one large bank of memory. All processors “see” tje sa,e data.
- All coordination/communication is done through memory.
- Each core is assigned a thread of execution of a single program that acts on the data.
- Can also use hyper-threading: assigning more than one thread to a given core.
- **OpenMP** is commonly used in this scenario.



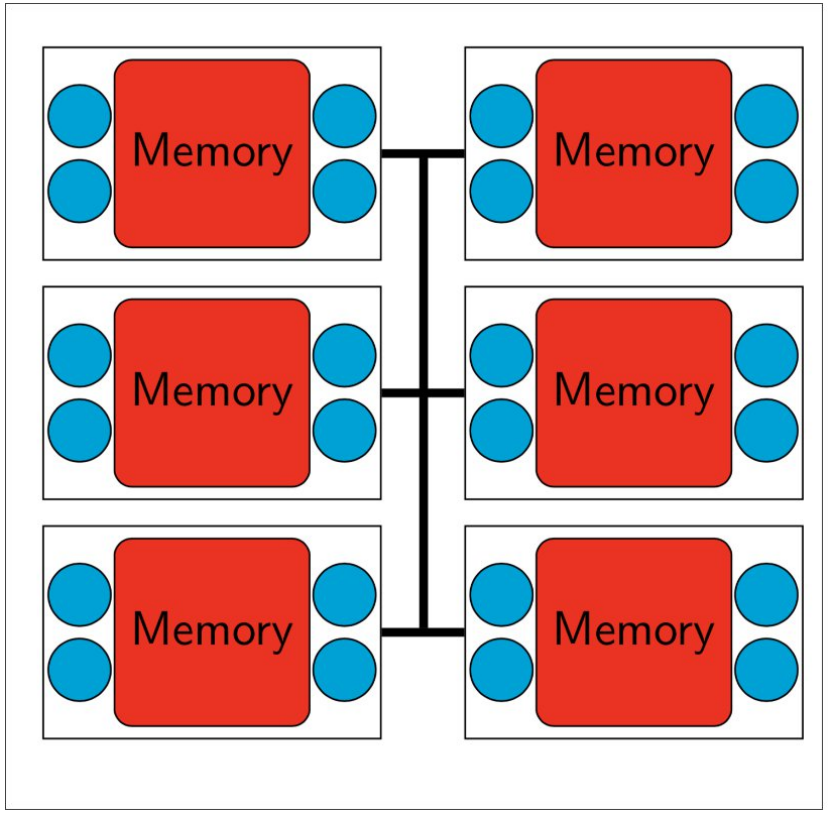
Threads vs processes

- **Threads**: Threads of execution within one process, with access to the same memory.
- **Processes:** Independent tasks with their own memory and resources.



Hybrid architectures

- Multicore nodes linked together with an (high-speed) interconnect.
- Many cores have modest vector(ization) capabilities.
- **OpenMP + MPI** can be used in this scenario.



Choosing your programming approach

The programming approach you use depends on the type of problem you have, and the type of machine that you will be using:

- Embarrassingly parallel applications: scripting, GNU Parallel.
- Shared memory machine: OpenMP, p-threads.
- Distributed memory machine: MPI, PGAS (UPC, Coarray Fortran).
- Graphics Computing: CUDA, OpenACC, OpenCL.
- Hybrid combinations.

We will focus on OpenMP, MPI programming.

Data or computation bound?

The programming approach you should use also depends upon the type of problem that needs to be solved.

- Computation bound, requires task parallelism
  - Need to focus on parallel processes/threads.
  - These processes may have very different computations to do.
  - Bring the data to the computation.
- Data bound, requires data parallelism.
  - The focus here is the operations on a large dataset.
  - The dataset is often an array, partitioned and tasks act on separate partitions.
  - Bring the computation to the data.

Granularity

The degree to which parallelizing your algorithm makes sense affects the approach used:

- **Fine**-grained (loop) parallelism.
  - Smaller individual tasks.
  - The data is transferred among processors frequently.
  - Shared memory model, **OpenMP**.
  - Scale limitations.
- **Coarse**-grained (task) parallelism
  - Divide and conquer.
  - Data communicated infrequently, after large amounts of computation.
  - Distributed memory, **MPI**.

Too fine-grained -> overhead issues.

Too coarse-grained -> load imbalance issues.

Summary

- You need to learn parallel programming to truly use the hardware that you have at your disposal.
- The serial only portions of your code will truly reduce the effectiveness of the parallelism of your algorithm. Minimize them.
- There are many different hardware types available: distributed-memory cluster, shared-memory, hybrid.
- The programming approach you need to use depends on the **nature of your problem**.

Serial programming

Sometimes your code is serial, meaning it only runs on a single processor, and that's far as it's going to go. There are several reasons why we might not push the code farther:

- The problem involves a parameter study; each iteration of the parameter study is fast; each iteration is independent, but many need to be done.
- The algorithm is inherently serial, and there's not much that can be done about it.
- You are running a commercial code, and do not have the source code to modify it.
- You do not have the time to parallelize your code.

Sometimes you just have to let your code be serial, and run the processes in parallel. That is our topic for today's lecture.

What are our assumptions?

Let us assume you have a serial code.

- Your code takes a set of parameters, either from a file or (preferably) from the command line.
- The code runs in a reasonably short amount of time (minutes to hours).
- You have a large parameter space you want to search, which means hundreds or thousands of combinations of values of parameters.
- You'd probably like some feedback on your jobs, things like error checking, fault tolerance, etc.
- You want to run your code on a cluster or a WorkStation.

How do we go about performing this set of calculations efficiently?

What are your options?

So how might we go about running multiple instances of the same code (as subjobs) simultaneously?

- Write a script from scratch which launches and manages the subjobs.
- If the code is written in Python, you could use Jupyter Notebook to manage the subjobs.
- If the code is written in R, you could use the parallel R utilities to manage the subjobs.
- Use an existing script, such as GNU parallel, to manage the subjobs.

Let us discuss the first and last options.

Working example, GNU Parallel.

My greet.py file

```
import sys

print(f"Hello World from task: {sys.argv[1]}")
```

The execution of my .py using multiple jobs in bash

```
#!/bin/bash

parallel -j 8 --progress python greet.py ::: {0..20}
```

Your working example: The (Conway's) game of life

You are required to graph the initial density vs the long-term density of alive cells in a 200x200 grid after $T = 200$ steps. You have to:

- Run the simulation for initial densities between 0 and 1.
  - Take a sample of 50 evenly spaced densities between 0 and 1.
  - Uniformly choose the initial alive cells in terms of the initial density.
  - Evolve the system.
  - Save the final density.
- Plot the final density in terms of the initial density.
- Re-execute your code multiple times. What happens with the graph?