PSC NCSA NSF

Award
2417789

Carnegie Mellon University

University of Pittsburgh

CI Pathways

Leading the Way to Effective Cyberinfrastructure Use

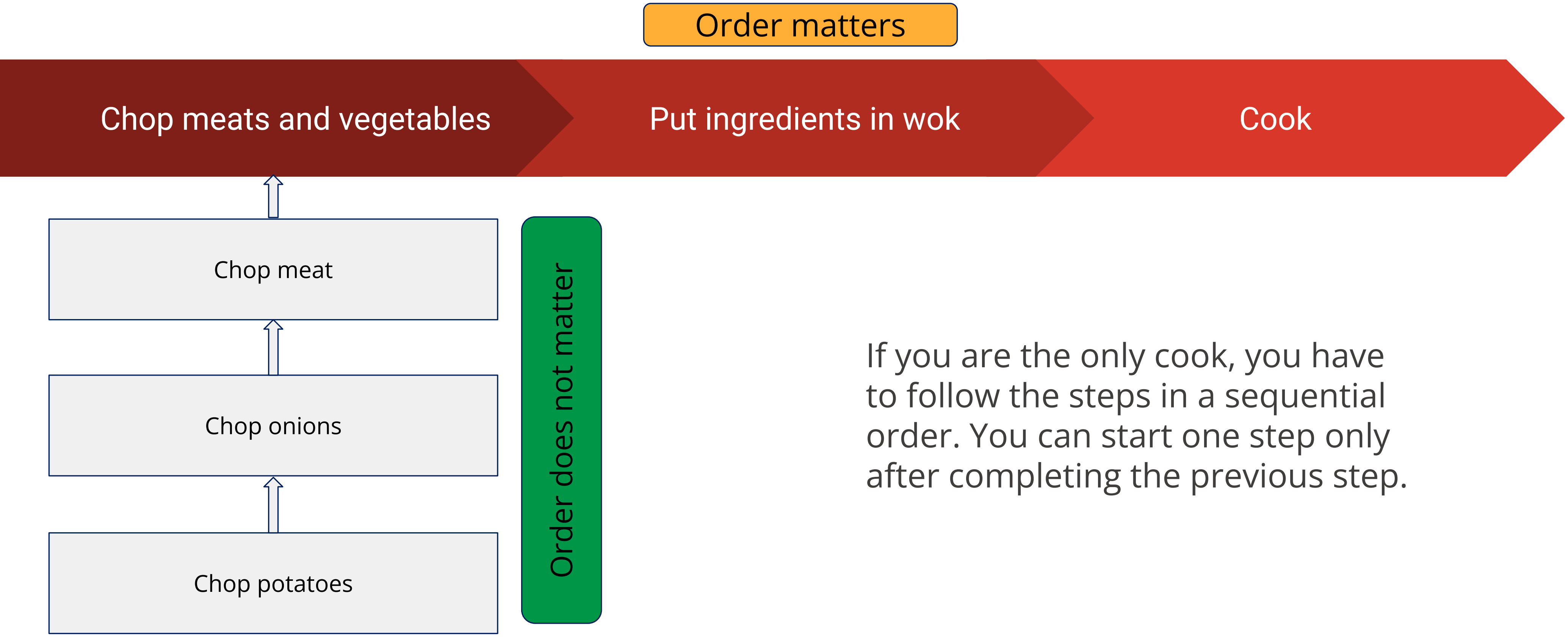
CI Awareness: Introduction to Parallel Computing

Soham Pal - Research Computing & Data Facilitator, NCSA

Overview

- What is parallel computing?
- Why is it needed?
- How to do it?

Serial computing



Order matters

Chop meats and vegetables

Put ingredients in wok

Cook

Chop meat

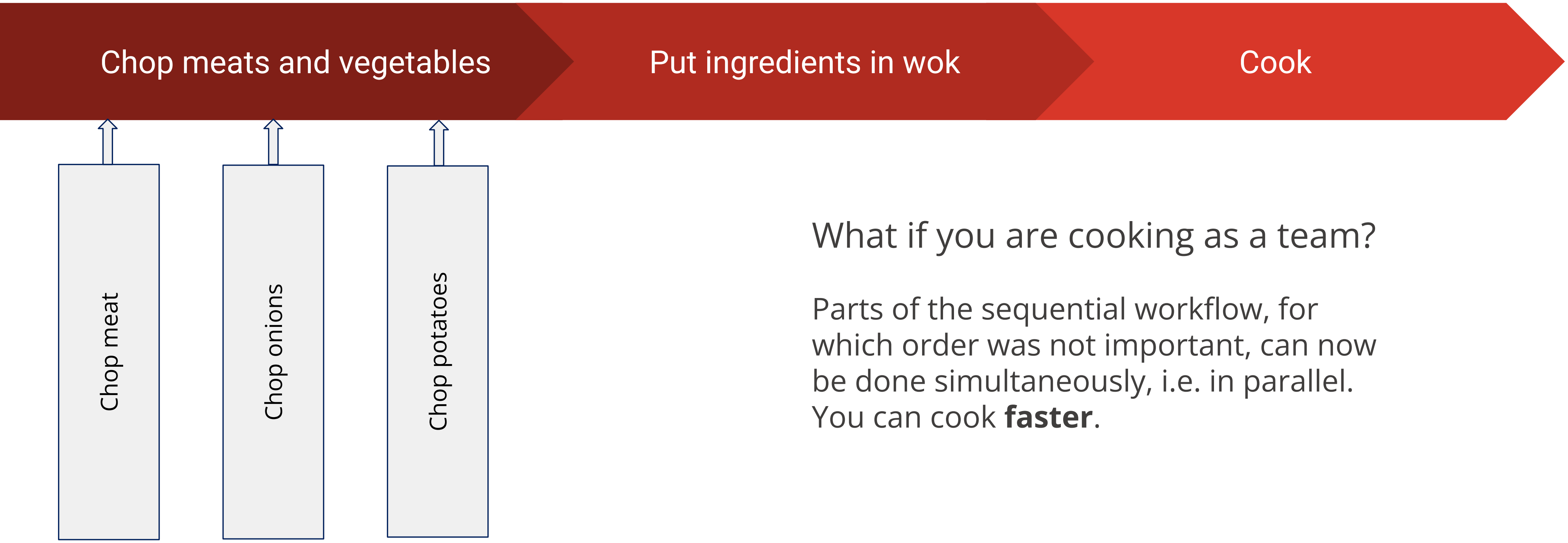
Chop onions

Chop potatoes

Order does not matter

If you are the only cook, you have to follow the steps in a sequential order. You can start one step only after completing the previous step.

Parallel computing - version 1



Chop meats and vegetables

Put ingredients in wok

Cook

Chop meat

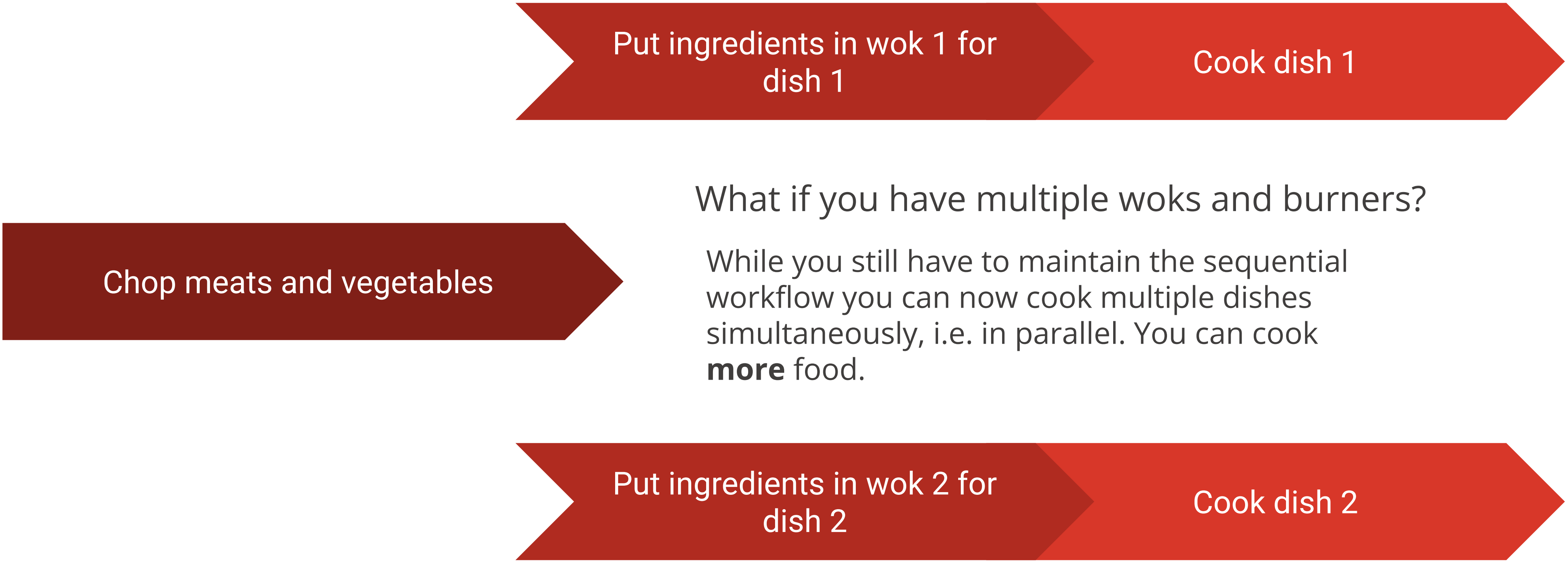
Chop onions

Chop potatoes

What if you are cooking as a team?

Parts of the sequential workflow, for which order was not important, can now be done simultaneously, i.e. in parallel. You can cook **faster**.

Parallel computing - version 2


Chop meats and vegetables

Put ingredients in wok 1 for dish 1

Cook dish 1

Put ingredients in wok 2 for dish 2

Cook dish 2

What if you have multiple woks and burners?

While you still have to maintain the sequential workflow you can now cook multiple dishes simultaneously, i.e. in parallel. You can cook **more** food.

Why parallel computing?

Parallel computing *can*:

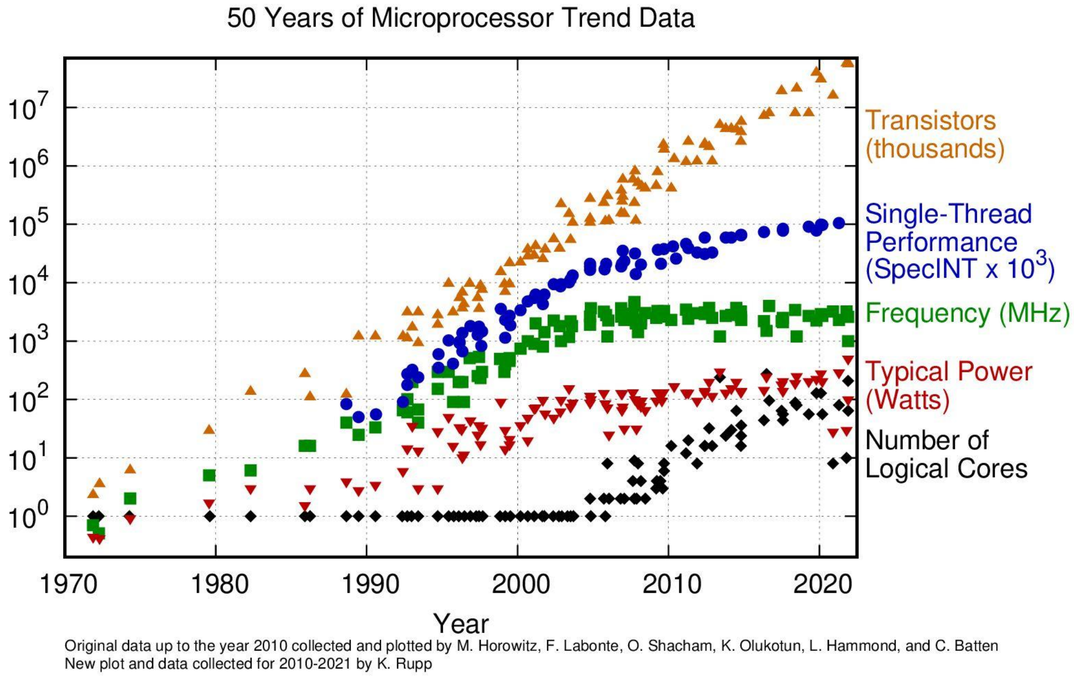
- make your workflow faster
- allow you to solve bigger problems

But is it necessary?

“The density of transistors in chips doubles every two years.” — Moore’s law

This is due to several reasons:

1. Increasing investments
2. Improving technology

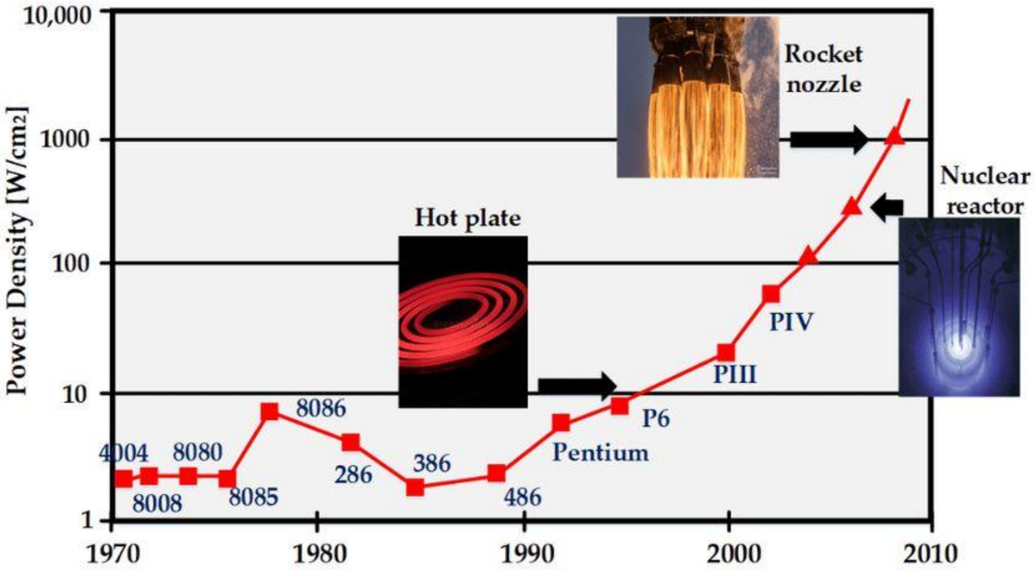


Source: https://github.com/karlrupp/microprocessor-trend-data

Why parallel computing?

At some stage physics intervenes:

1. Packing too many transistors in the same space while maintaining the speed of the processors makes things **too hot**
2. Shrinking transistors beyond a certain limit means quantum effects become important



Source: *Micromachines* **2021**, *12*(6), 665

How do computer programs work?

Computer programs basically work by performing some task (“do something”) with some data.

Two basic types of parallelism are possible:

1. Task parallelism — Perform different tasks on the same data
2. Data parallelism — Perform the same task on different data

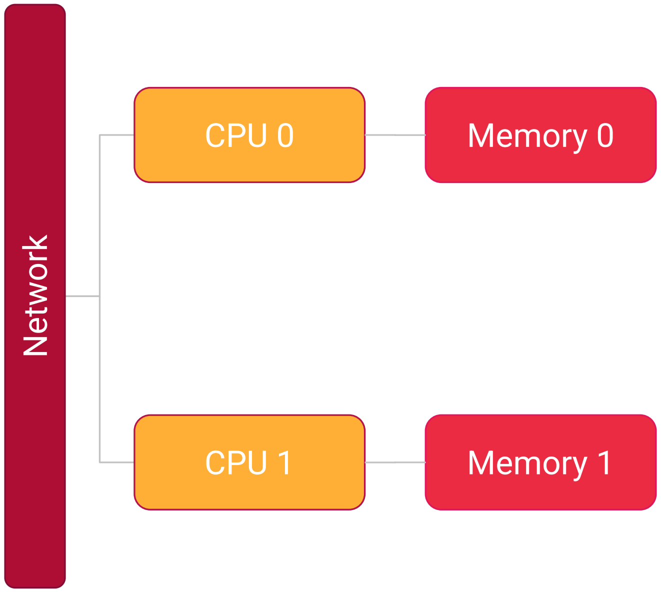
The two can be combined.

Processing units

1. CPUs — Typically contain a few powerful processors
2. GPUs — Typically many not so powerful processors

There are other specialized processing units such as TPUs, but we will not discuss them here. For the rest of the presentation we will focus on CPUs.

Memory



Network

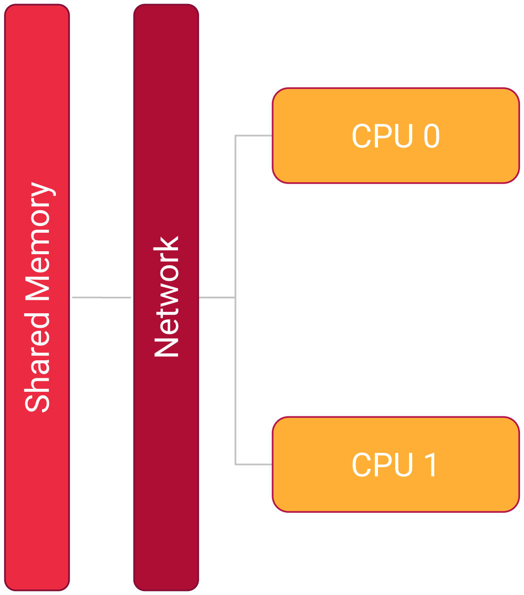
CPU 0

Memory 0

CPU 1

Memory 1

Distributed Memory



Shared Memory

Network

CPU 0

CPU 1

Shared Memory

We will look at this in this presentation.

Control flow

		Data Stream	
		Single	Multiple
Instruction Stream	Single	SISD	SIMD
	Multiple	MISD	MIMD

Packages like NumPy make good use of this.

This is how current supercomputers work.

Flynn’s classification of control flow

Some terminologies

- Thread: Smallest sequence of instructions that the OS scheduler can handle
- Process: Composed of threads, with its own allocated memory
- Core: Can run an independent thread of code
- Processors: Physical chips, typically has multiple cores
- Nodes: Physical unit with network connection
- Processing element: Smallest useful computing device, usually a core

Slurm

Supercomputers (or computing clusters) are shared resources, and thus need an efficient way to allocate resources and manage jobs. A job, for the purposes of this presentation, is a computational task.

Slurm is a resource allocation manager and job scheduler for computing clusters. There are others like Torque, PBS, though Slurm is probably the most common.

For more information, see https://slurm.schedmd.com/overview.html.

Parallelization in research workflows

Parallelization in research workflows usually happens in two ways:

1. Language-level parallelism. This is what is typically demonstrated in parallel computing training, and we will see this in this presentation.
2. Running the same program but with different inputs parallely. This is usually not talked about in parallel computing training, but is very common in research workflows. We will see some examples of this too.

Running the same program with different inputs in a case of embarrassingly parallel computation. Each run is completely independent of the other and therefore shows the best scaling, provided resources are available.

Language-level parallelism

Language-level parallelism is usually done with APIs like:

1. OpenMP (Open Multi-Processing) for shared memory parallelism
2. MPI (Message Passing Interface) for distributed memory parallelism

If using GPUs, OpenACC and NCCL (for NVIDIA GPUs) are useful APIs.

Typically C/C++/Fortran provide the best support for OpenMP and MPI. Other languages like Python and R have some support via third party libraries, though they might provide other alternatives.

For computational programming with Python it is best to use libraries like NumPy, instead of standard Python, which implicitly uses OpenMP and SIMD for many situations.

Same program, different input parallelism

This sort of parallelism can be done with tools like:

1. Slurm job arrays
2. GNU Parallel
3. Python’s multiprocessing library

We will look at examples of Slurm job arrays and Python’s multiprocessing library. I recommend checking out GNU Parallel for this kind of workflow. For more information, see https://www.gnu.org/software/parallel/.

We will use Slurm for today’s examples.

Demonstrations