THE UNIVERSITY OF ARIZONA

Intro to Parallel Computing on HPC

Summer 2024

Ethan Jahn
UITS Research Technologies

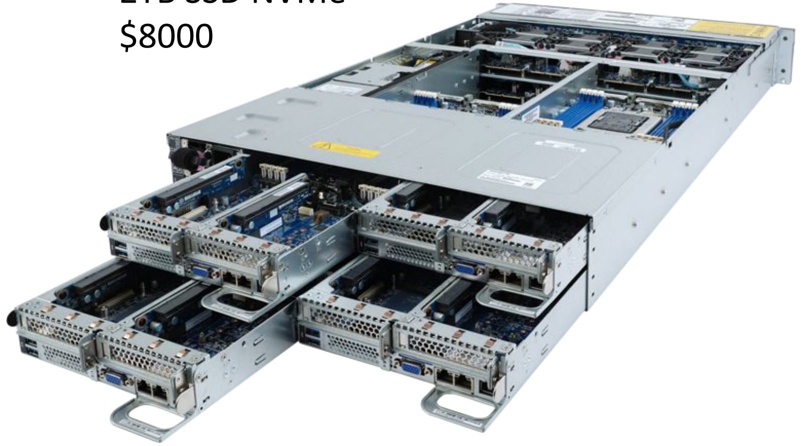
access these slides: https://bit.ly/4eZEQwD

Outline

Why Do We Care?

MacBook Pro 2021
Apple M1 Pro Chip
8-core CPU
16GB unified memory
512GB SSD
$1999

Penguin Altus XE2242
4in1 chassis. Each compute node has:
96 cores dual socket AMD EPYC 7642
512GB DDR4 3200MHz ECC memory
2TB SSD NVMe
$8000



THE UNIVERSITY OF ARIZONA

Why Do We Care?

MacBook Pro 2021
Apple M1 Pro Chip
8-core CPU
16GB unified memory
512GB SSD
$1999

Puma Cluster
269 compute nodes
25,824 cores dual socket AMD EPYC 7642
137,728GB DDR4 3200MHz ECC memory
2PB SSD NVMe shared storage
538TB local storage
$2.7M

The University of Arizona

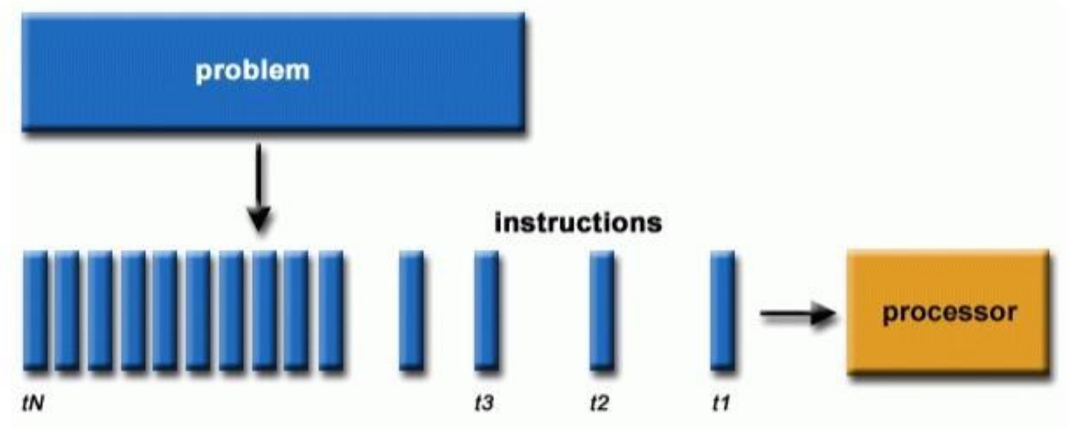
What is Parallel Computing?

Serial Computing

Problem is broken into a *discrete series* of instructions

Instructions executed sequentially **on a single processor (core)**

Only **one** instruction can execute in each time step



Serial computing generic example

What is Parallel Computing?

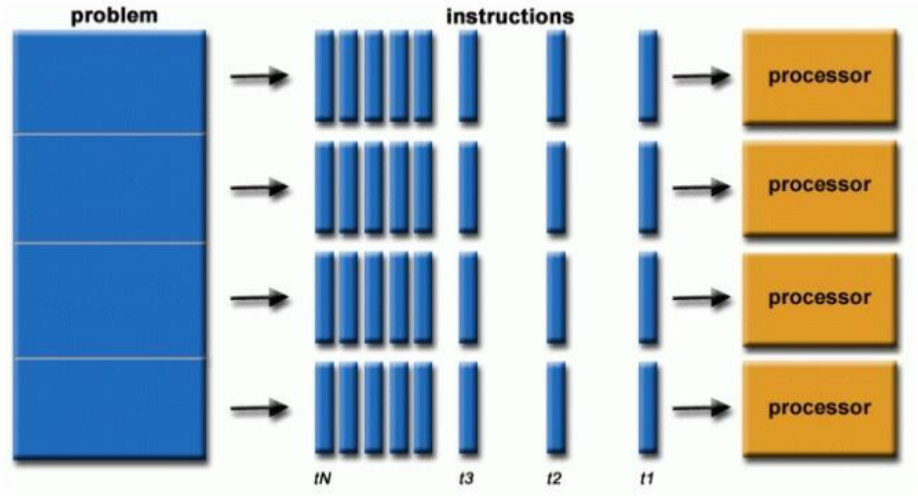
Parallel Computing

Problem divided into **`discrete`** parts that can be solved *concurrently*

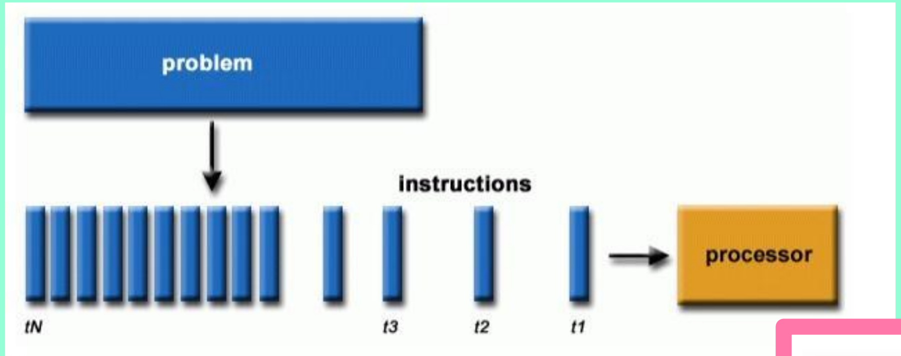
→ Further divided to series of instructions

Instructions from each section execute **simultaneously** on *different* processors

Need to employ some overall coordination method

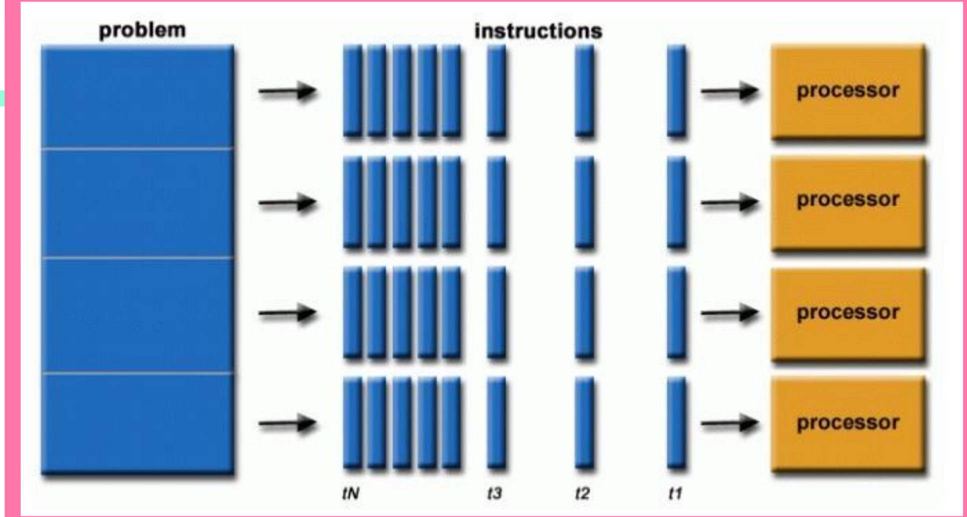


What is Parallel Computing?



Serial computing generic example

Serial Computing



Parallel Computing

What is Parallel Computing?

The **majority** of stand-alone computers today are *parallel* from a hardware perspective:

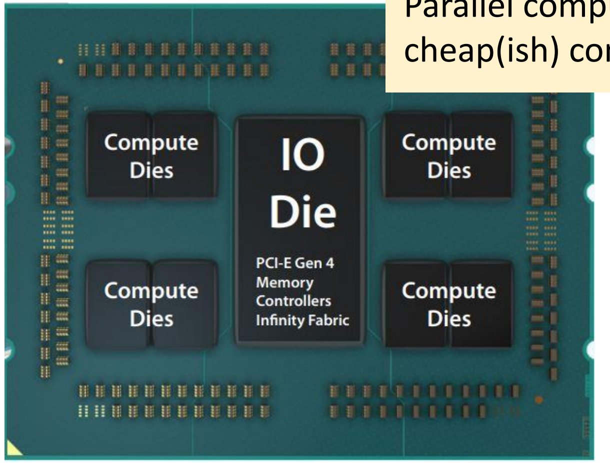
Multiple **functional** units

Multiple **execution** units/cores

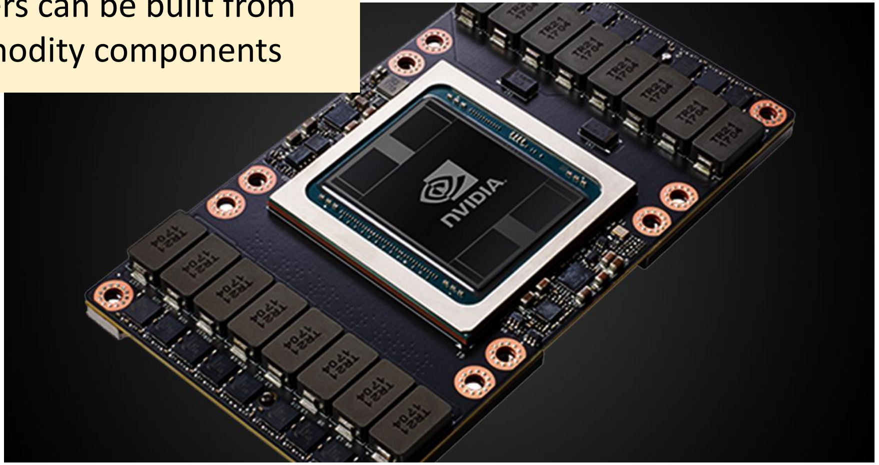
Multiple hardware **threads**

(L1 cache, L2 cache, branch, prefetch, decode, floating-point, graphics processing (GPU), integer, etc.)

Parallel computers can be built from cheap(ish) commodity components



AMD EPYC Rome
CPU



Nvidia V100
GPU

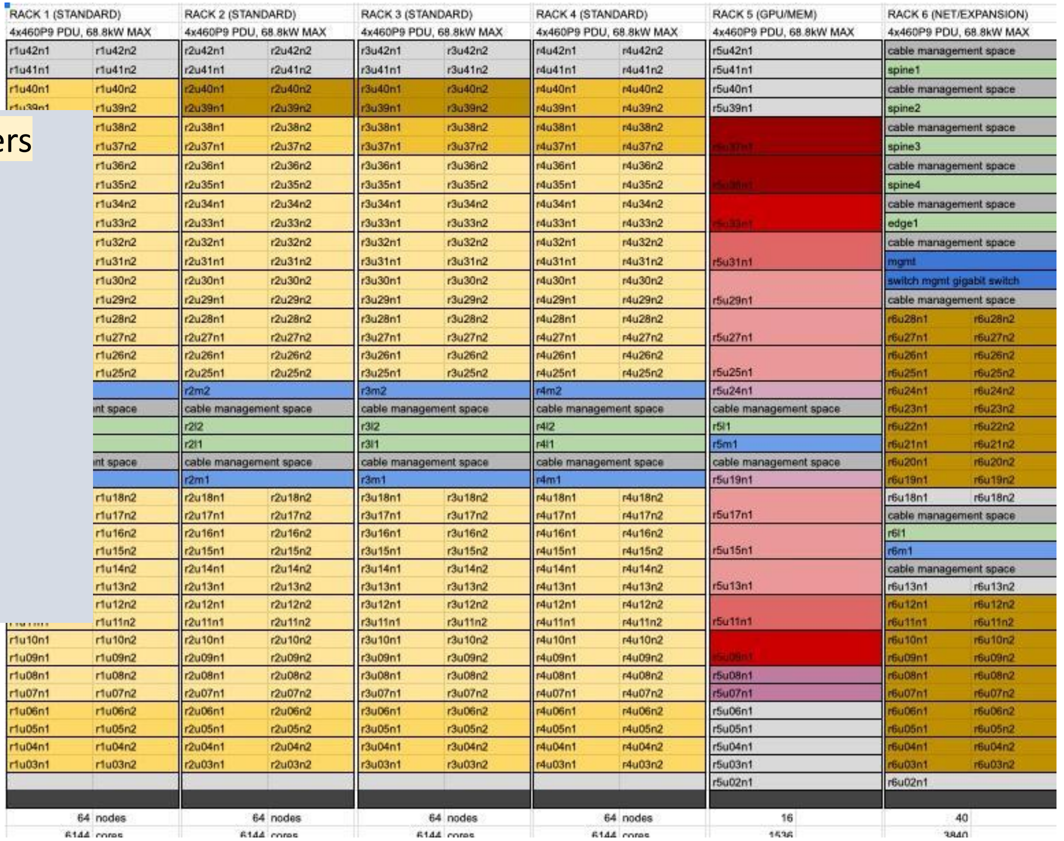
What is Parallel Computing?

Parallel Computers

Networks connect multiple stand-alone computers (***nodes***) to make larger `parallel computer clusters`.

Each compute node is a **multi**processor **parallel** computer in itself
→ connected via a high-speed network

Special purpose nodes (also multiprocessor)
→ GPU nodes
→ high memory nodes



RACK 1 (STANDARD)		RACK 2 (STANDARD)		RACK 3 (STANDARD)		RACK 4 (STANDARD)		RACK 5 (GPU/MEM)	RACK 6 (NET/EXPANSION)	
4x460P9 PDU, 68.8kW MAX		4x460P9 PDU, 68.8kW MAX		4x460P9 PDU, 68.8kW MAX		4x460P9 PDU, 68.8kW MAX		4x460P9 PDU, 68.8kW MAX	4x460P9 PDU, 68.8kW MAX	
r1u42n1	r1u42n2	r2u42n1	r2u42n2	r3u42n1	r3u42n2	r4u42n1	r4u42n2	r5u42n1	cable management space	
r1u41n1	r1u41n2	r2u41n1	r2u41n2	r3u41n1	r3u41n2	r4u41n1	r4u41n2	r5u41n1	spine1	
r1u40n1	r1u40n2	r2u40n1	r2u40n2	r3u40n1	r3u40n2	r4u40n1	r4u40n2	r5u40n1	cable management space	
r1u39n1	r1u39n2	r2u39n1	r2u39n2	r3u39n1	r3u39n2	r4u39n1	r4u39n2	r5u39n1	spine2	
[illegible]	r1u38n2	r2u38n1	r2u38n2	r3u38n1	r3u38n2	r4u38n1	r4u38n2		cable management space	
[illegible]	r1u37n2	r2u37n1	r2u37n2	r3u37n1	r3u37n2	r4u37n1	r4u37n2	r5u37n1	spine3	
[illegible]	r1u36n2	r2u36n1	r2u36n2	r3u36n1	r3u36n2	r4u36n1	r4u36n2		cable management space	
[illegible]	r1u35n2	r2u35n1	r2u35n2	r3u35n1	r3u35n2	r4u35n1	r4u35n2	r5u35n1	spine4	
[illegible]	r1u34n2	r2u34n1	r2u34n2	r3u34n1	r3u34n2	r4u34n1	r4u34n2		cable management space	
[illegible]	r1u33n2	r2u33n1	r2u33n2	r3u33n1	r3u33n2	r4u33n1	r4u33n2	r5u33n1	edge1	
[illegible]	r1u32n2	r2u32n1	r2u32n2	r3u32n1	r3u32n2	r4u32n1	r4u32n2		cable management space	
[illegible]	r1u31n2	r2u31n1	r2u31n2	r3u31n1	r3u31n2	r4u31n1	r4u31n2	r5u31n1	mgmt	
[illegible]	r1u30n2	r2u30n1	r2u30n2	r3u30n1	r3u30n2	r4u30n1	r4u30n2		switch mgmt gigabit switch	
[illegible]	r1u29n2	r2u29n1	r2u29n2	r3u29n1	r3u29n2	r4u29n1	r4u29n2	r5u29n1	cable management space	
[illegible]	r1u28n2	r2u28n1	r2u28n2	r3u28n1	r3u28n2	r4u28n1	r4u28n2		r6u28n1	r6u28n2
[illegible]	r1u27n2	r2u27n1	r2u27n2	r3u27n1	r3u27n2	r4u27n1	r4u27n2	r5u27n1	r6u27n1	r6u27n2
[illegible]	r1u26n2	r2u26n1	r2u26n2	r3u26n1	r3u26n2	r4u26n1	r4u26n2		r6u26n1	r6u26n2
[illegible]	r1u25n2	r2u25n1	r2u25n2	r3u25n1	r3u25n2	r4u25n1	r4u25n2	r5u25n1	r6u25n1	r6u25n2
[illegible]		r2m2		r3m2		r4m2		r5u24n1	r6u24n1	r6u24n2
[illegible]	nt space	cable management space		cable management space		cable management space		cable management space	r6u23n1	r6u23n2
[illegible]		r2l2		r3l2		r4l2		r5l1	r6u22n1	r6u22n2
[illegible]		r2l1		r3l1		r4l1		r5m1	r6u21n1	r6u21n2
[illegible]	nt space	cable management space		cable management space		cable management space		cable management space	r6u20n1	r6u20n2
[illegible]		r2m1		r3m1		r4m1		r5u19n1	r6u19n1	r6u19n2
[illegible]	r1u18n2	r2u18n1	r2u18n2	r3u18n1	r3u18n2	r4u18n1	r4u18n2		r6u18n1	r6u18n2
[illegible]	r1u17n2	r2u17n1	r2u17n2	r3u17n1	r3u17n2	r4u17n1	r4u17n2	r5u17n1	cable management space	
[illegible]	r1u16n2	r2u16n1	r2u16n2	r3u16n1	r3u16n2	r4u16n1	r4u16n2		r6l1	
[illegible]	r1u15n2	r2u15n1	r2u15n2	r3u15n1	r3u15n2	r4u15n1	r4u15n2	r5u15n1	r6m1	
[illegible]	r1u14n2	r2u14n1	r2u14n2	r3u14n1	r3u14n2	r4u14n1	r4u14n2		cable management space	
[illegible]	r1u13n2	r2u13n1	r2u13n2	r3u13n1	r3u13n2	r4u13n1	r4u13n2	r5u13n1	r6u13n1	r6u13n2
[illegible]	r1u12n2	r2u12n1	r2u12n2	r3u12n1	r3u12n2	r4u12n1	r4u12n2		r6u12n1	r6u12n2
[illegible]	r1u11n2	r2u11n1	r2u11n2	r3u11n1	r3u11n2	r4u11n1	r4u11n2	r5u11n1	r6u11n1	r6u11n2
r1u10n1	r1u10n2	r2u10n1	r2u10n2	r3u10n1	r3u10n2	r4u10n1	r4u10n2		r6u10n1	r6u10n2
r1u09n1	r1u09n2	r2u09n1	r2u09n2	r3u09n1	r3u09n2	r4u09n1	r4u09n2	r5u09n1	r6u09n1	r6u09n2
r1u08n1	r1u08n2	r2u08n1	r2u08n2	r3u08n1	r3u08n2	r4u08n1	r4u08n2	r5u08n1	r6u08n1	r6u08n2
r1u07n1	r1u07n2	r2u07n1	r2u07n2	r3u07n1	r3u07n2	r4u07n1	r4u07n2	r5u07n1	r6u07n1	r6u07n2
r1u06n1	r1u06n2	r2u06n1	r2u06n2	r3u06n1	r3u06n2	r4u06n1	r4u06n2	r5u06n1	r6u06n1	r6u06n2
r1u05n1	r1u05n2	r2u05n1	r2u05n2	r3u05n1	r3u05n2	r4u05n1	r4u05n2	r5u05n1	r6u05n1	r6u05n2
r1u04n1	r1u04n2	r2u04n1	r2u04n2	r3u04n1	r3u04n2	r4u04n1	r4u04n2	r5u04n1	r6u04n1	r6u04n2
r1u03n1	r1u03n2	r2u03n1	r2u03n2	r3u03n1	r3u03n2	r4u03n1	r4u03n2	r5u03n1	r6u03n1	r6u03n2
								r5u02n1	r6u02n1	
64	nodes	64	nodes	64	nodes	64	nodes	16	40	
6144	cores	6144	cores	6144	cores	6144	cores	1536	3840	

Puma Rack layout

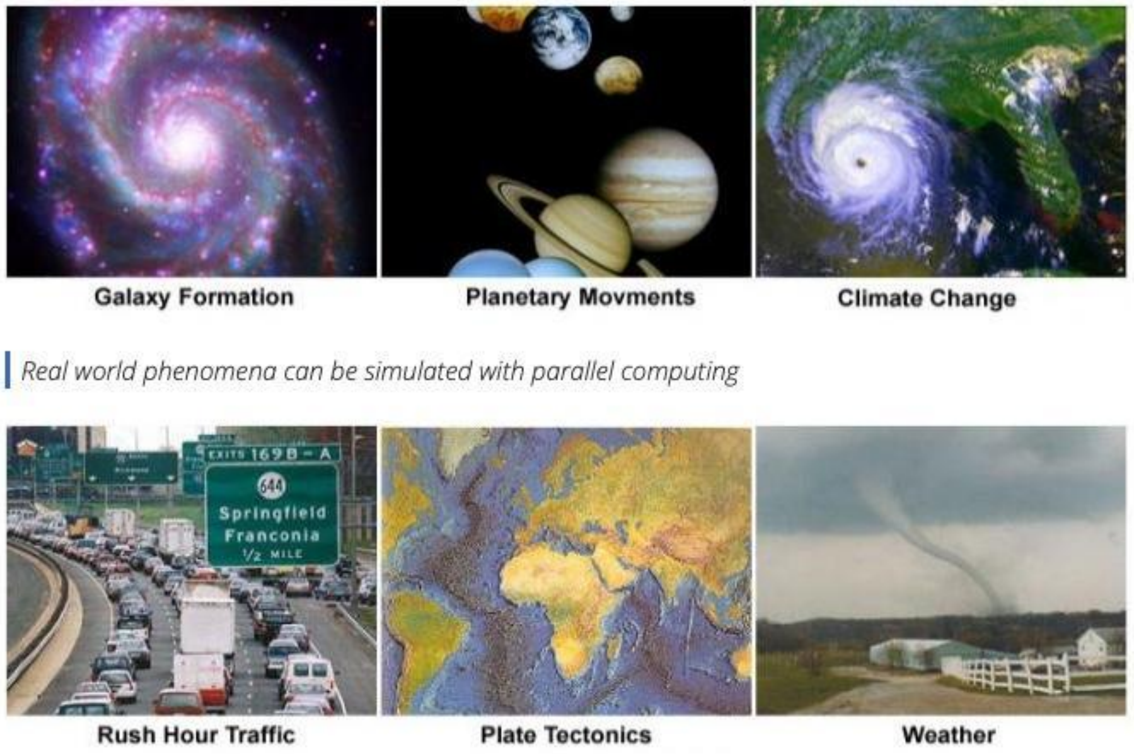
Why Use Parallel Computing?

Parallelization Accommodates Complexity

Natural processes can be accurately modeled with **high resolution** simulations or models

- large *number* of components
- multiple *types* of components
- interactions
- temporal sequence

Example: Natural Language Processing models have billions of parameters



Why Use Parallel Computing?

Parallelization Decreases Time to Result

Tasks with a greater number of *independent calculations* will benefit from **dividing the load** between more processors

Working in parallel shortens completion time

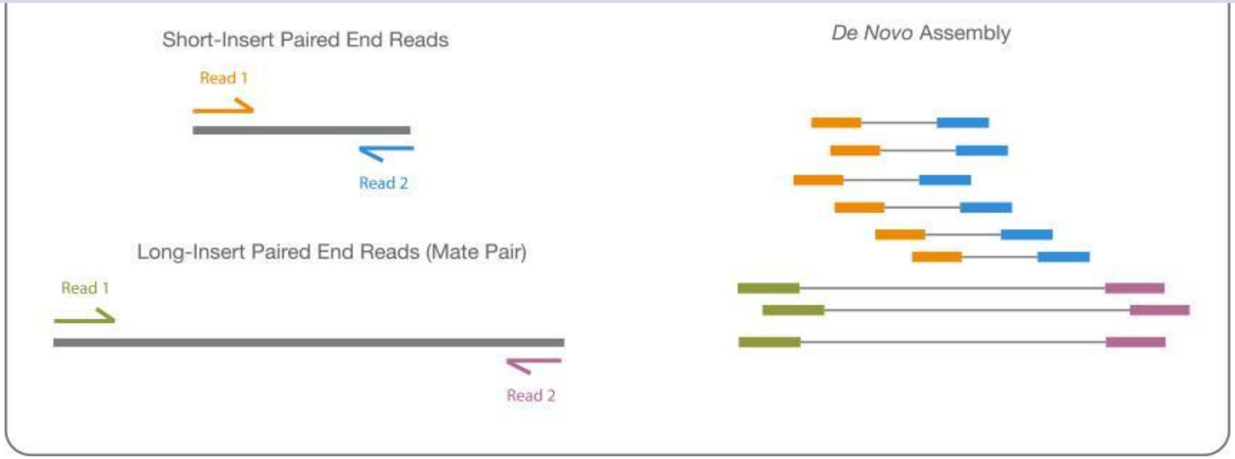
Why Use Parallel Computing?

Parallelization Provides Concurrency

A **single processor** can only perform **one** operation at a time.

Coordinating **multiple processors** allows for **many operations** to be performed in **one clock cycle**.

Example: The "shotgun" technique sequences a genome by breaking a long string of information into shorter segments, then reassemble


Short-Insert Paired End Reads
Read 1
Read 2
Long-Insert Paired End Reads (Mate Pair)
Read 1
Read 2
De Novo Assembly

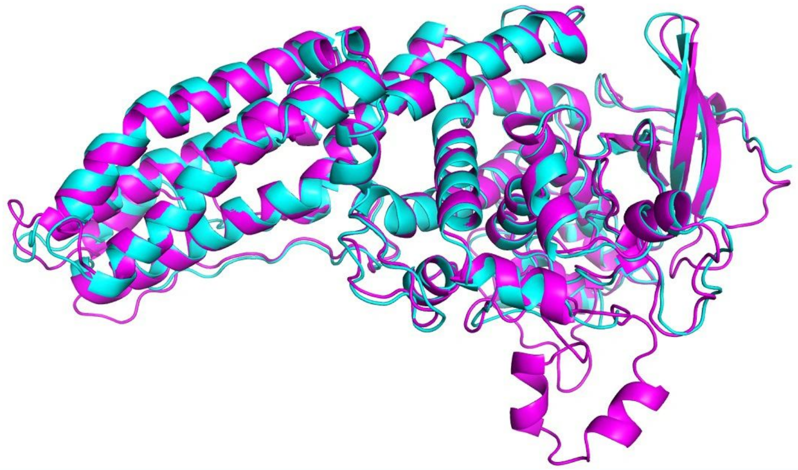
Why Use Parallel Computing?

Parallelization Takes Advantage Of Non-Local Resources

- Users don’t need to manage complex hardware
- Access **powerful computing** from anywhere with an internet connection
- **Distributed computing** allows for another meta-level parallelization

Example:

- Folding@Home is a distributed computing project to simulate protein dynamics.
- Supercomputers at the **three public Arizona universities** have contributed



Parallel Computing Terminology

Node –

an **individual computer**. A collection of them comprises a supercomputer

CPU –

AKA socket or **processor**. A physical device mounted on the motherboard. Puma nodes have two CPU's

Core –

- Part of CPU capable of conducting independent work.
- Puma CPU's have 48 cores for a total of 96 per node.
- 94 cores/node are usable
- ***However, in Slurm, "CPU" = "Core"***

Lawrence Livermore National Laboratory

The University of Arizona

Parallel Computing Terminology

Process –

instance of a program, with access to its own memory, state and file descriptors

Task –

a ***logically discrete*** section of computational work. By default, Slurm allocates one CPU per task

Thread –

highest level of code executed by a processor. Each process has at least one thread

Parallel Computing Terminology

NUMA: Non-Uniform Memory Access.

→ Global address space shared by all cores.

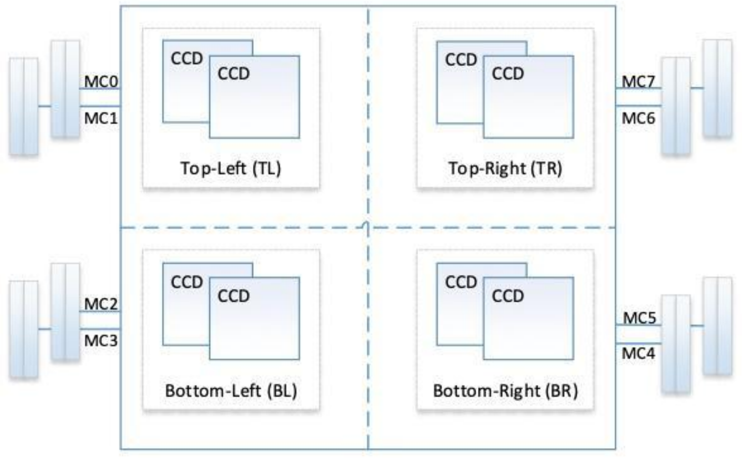
→ Memory is local to each processor or remote, which is slower.

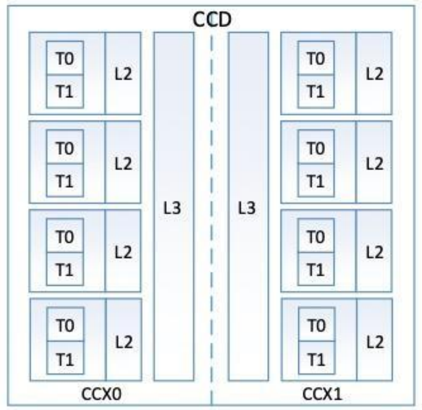
Cache memory:

→ memory that is much faster but smaller and expensive.

→ L1 cache is on the core, L2 is next to each core and L3 is shared between 4 cores.

AMD Rome Core Complex





THE UNIVERSITY OF ARIZONA

Parallel Computing Terminology

Types of HPC computation

Serial
- Computation runs on **one core** on **one node**
- Sometimes called High Throughput Computing

Shared Memory (AKA *multi-threading*)
- Single process with multiple threads
- Cores on ***single node*** work together
- Low level coordination
- Threads access shared memory space.

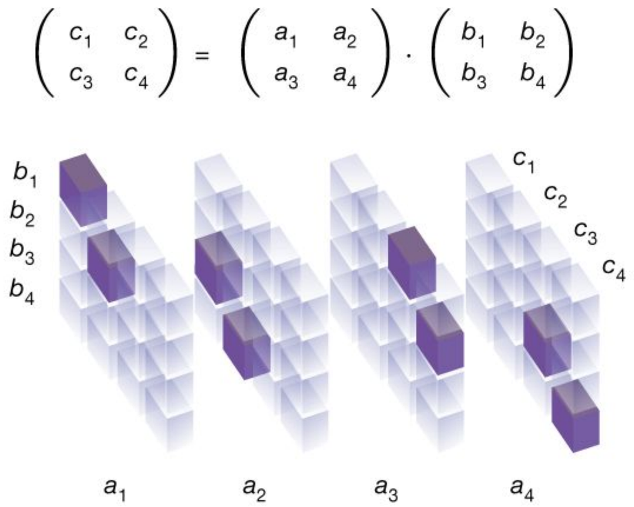
Distributed Memory (AKA *multi-node*)
- Cores on ***multiple nodes*** work *independently*
- High level coordination
- Coordination by **passing messages** over network.
- Supports *large memory* or *many CPU* workloads.

THE UNIVERSITY OF ARIZONA

Parallel Computing Terminology

Massively Parallel –
workloads that use many hundreds or thousands of cores

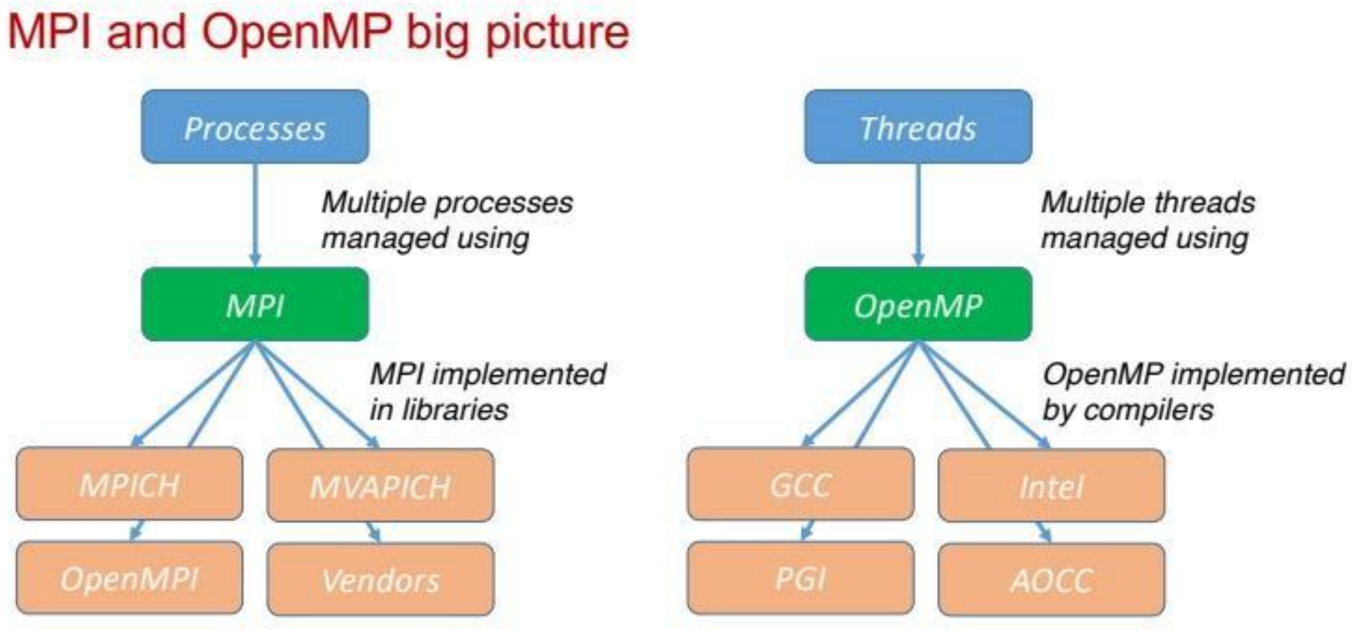
Embarrassingly Parallel –
A task that contains perfectly independent computations, e.g. matrix multiplication. Achieves ideal scaling.


$$\begin{pmatrix} c_1 & c_2 \\ c_3 & c_4 \end{pmatrix} = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} \cdot \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix}$$

MPI – Message Passing Interface

- Standard defining multi-node communication for distributed memory computing
- Implementations:
  → OpenMPI, Intel MPI, MPICH, MVAPICH
- OpenMPI and Intel MPI are encouraged on UA HPC clusters

Parallel Computing Terminology


MPI and OpenMP big picture
Processes
Multiple processes managed using
MPI
MPI implemented in libraries
MPICH
MVAPICH
OpenMPI
Vendors
Threads
Multiple threads managed using
OpenMP
OpenMP implemented by compilers
GCC
Intel
PGI
AOCC

MPI is a standard for parallelizing C, C++ and Fortran code to run on distributed memory systems *(multi-node)*

OpenMP is an application programming interface (API) for shared-memory parallel programming in C, C++ and Fortran *(single node)*

Parallel Computing Theory

von Neumann Computer Architecture

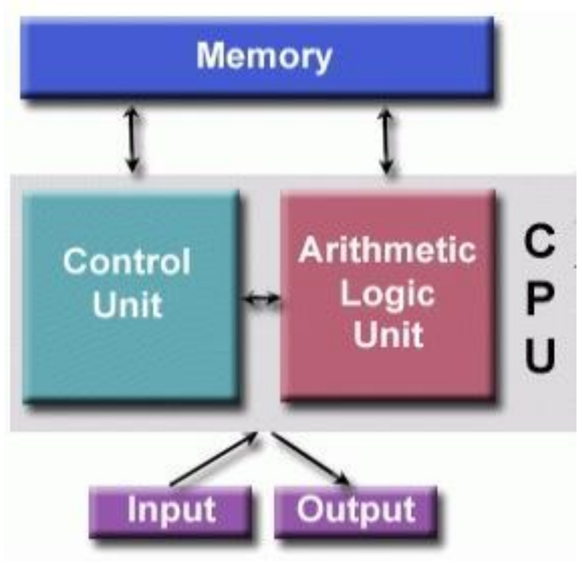
John von Neumann

- Hungarian mathematician
- authored the ***general requirements for an electronic computer*** in 1945

"stored-program computer"

- both program instructions and data are kept in electronic memory.
- earlier computers programmed through "hard wiring"

Since then, basically all computers have followed this basic design:



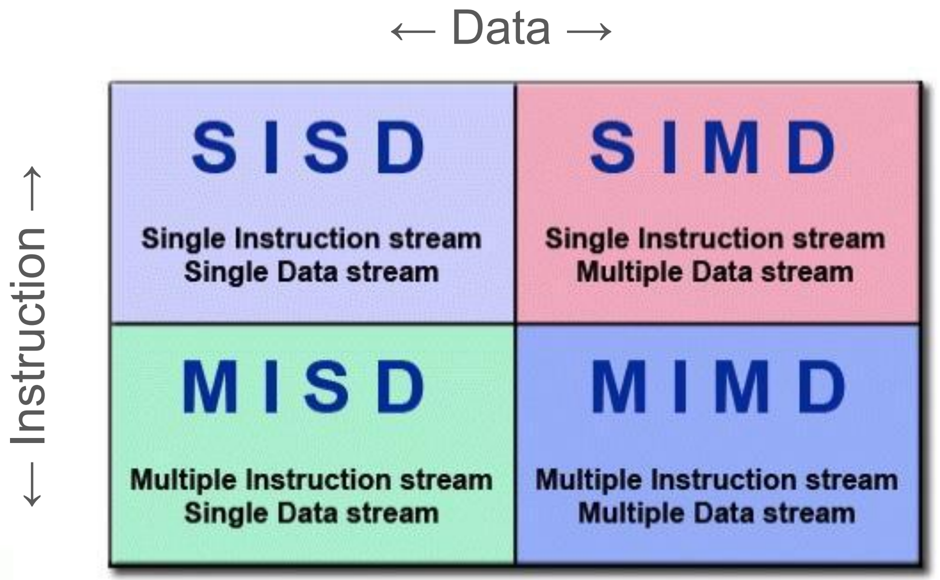
John von Neumann circa 1940s (Source: LANL archives)

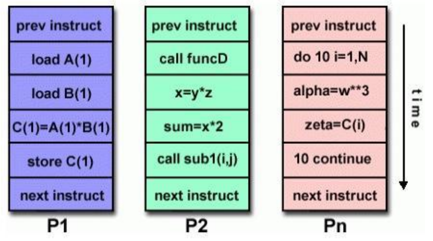
Parallel Computing Theory

Flynn's Classical Taxonomy (1966)

Scheme for classifying parallel computers.

- Distinguishes types of **multi-processor** computer architectures
- Classifies based on multiplicity of ***Instruction*** versus ***Data* Streams**
- Each of these can be ***Single*** or ***Multiple***





MIMD: HPC

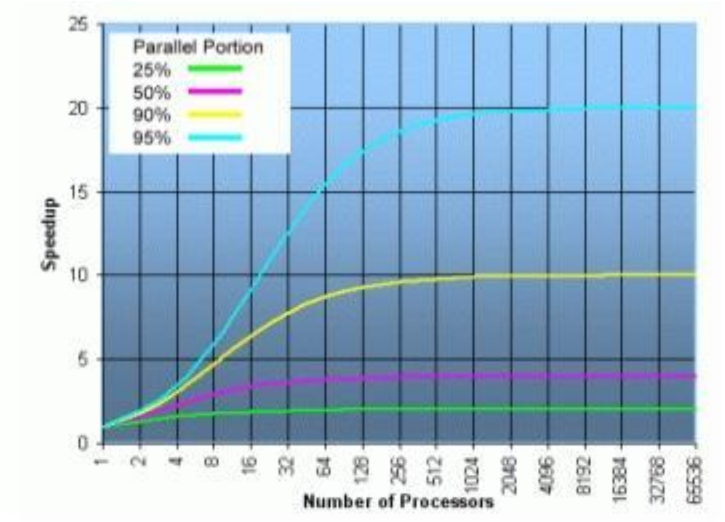
Lawrence Livermore National Laboratory

The University of Arizona

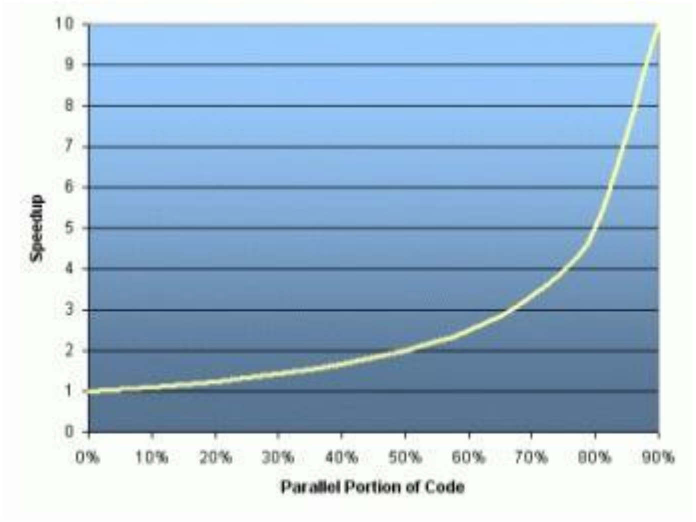
Parallel Computing Theory

Amdahl's Law

theoretical maximum speedup is determined by the fraction of code that can be run in **parallel**.



Speedup when introducing more processors



Amdahl's law

Parallel Computing Theory

Serialized code

A naïve inner product algorithm of two vectors of one million elements each

- All multiplications can be done in one time unit (parallel)
- Additions to a single accumulator in one million time units (serial)

$$\begin{array}{ccccl} x & & y & & \\ \left[\begin{array}{c} x_1 \\ \\ x_2 \\ \\ x_3 \\ \\ x_4 \\ \\ x_5 \end{array}\right] & \begin{array}{c} \cdot \\ \\ \cdot \\ \\ \cdot \\ \\ \cdot \\ \\ \cdot \end{array} & \left[\begin{array}{c} y_1 \\ \\ y_2 \\ \\ y_3 \\ \\ y_4 \\ \\ y_5 \end{array}\right] & \begin{array}{c} = \\ \\ = \\ \\ = \\ \\ = \\ \\ = \end{array} & \begin{array}{c} P_1 \\ + \\ P_2 \\ + \\ P_3 \\ + \\ P_4 \\ + \\ P_5 \end{array} \end{array}$$

$$= P$$

Amdahl's Law

- If a fraction X of a computation is run in serial, the parallel speedup cannot be more than 1/X

Exercise

- what **fraction of the code** for the operation to the left is **parallelizable**?
- what is the ***expected fractional speedup*** compared to serial?

Parallel Computing Theory

Scaling

Strong scaling (Amdahl):

Total problem size stays fixed as more processors are added.

Goal is to run the same problem size faster

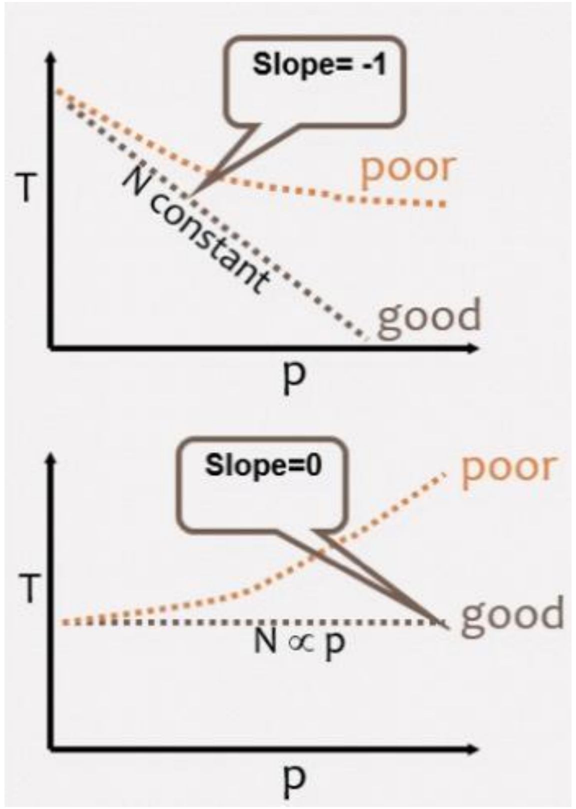
Perfect scaling means problem is solved in 1/P time (compared to serial)

Weak scaling (Gustafson):

The problem size *per processor* stays fixed as more processors are added. The total problem size is proportional to the number of processors used.

Goal is to run larger problem in same amount of time

Perfect scaling means problem P_N runs in same time as P_1



Parallel Computing Theory

Scaling

Strong scaling (Amdahl):

erial)

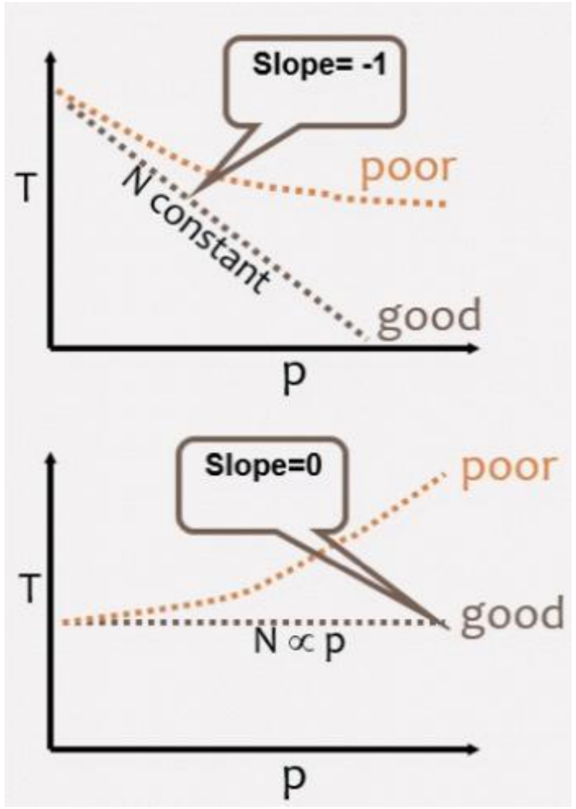
Most research developments are made by utilizing so-called "weak" scaling!

Weak scaling (Gustafson):

The problem size *per processor* stays fixed as more processors are added. The total problem size is proportional to the number of processors used.

Goal is to run larger problem in same amount of time

Perfect scaling means problem P_N runs in same time as P_1



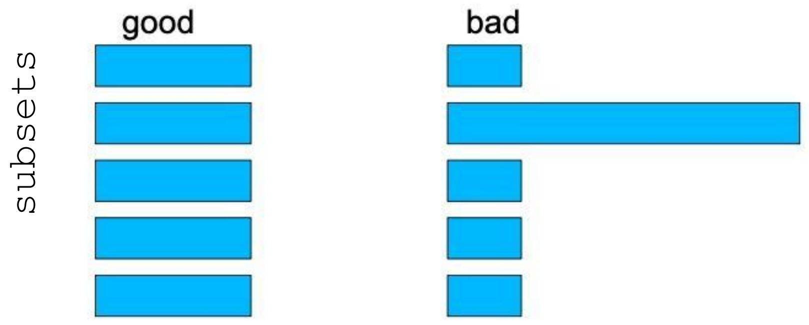
Lawrence Livermore National Laboratory

THE UNIVERSITY OF ARIZONA

Parallel Computing Theory

Load Balance

The total amount of time to complete a parallel job is limited by the thread that takes the longest to finish



Computation per subset

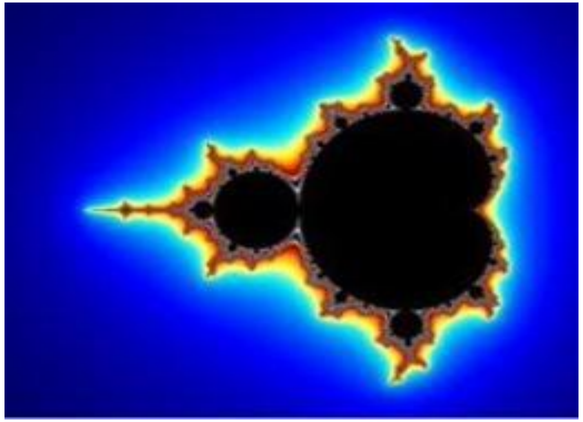
Parallel Computing Theory

Load Imbalance

Caused by **non-uniform data distributions**
- large regions of very low density
- small regions of very high density

Occurs in astronomy, medical imaging, rendering, etc.

If the space is divided evenly across threads
- some threads will do *very little work*
  → low density = **few elements**
- some threads will do a *lot of work*
  → high density = **many elements**



Parallel Computing Terminology

MPI Implementations

Julia has anMPI language wrapper

MATLAB has its own parallel extension library implemented using MPI and PVM

Python Implementations of MPI include pyMPI, mpi4py, para, and MYMPI
Boost C++ Libraries acquired Boost:MPI which include MPI Python Bindings.

R Bindings of MPI include Rmpi and pbdMPI.

On HPC we support OpenMPI and Intel MPI. OpenMPI is a default module that is loaded with GCC 8.3
Intel MPI is provided when you unload OpenMPI and GCC, and then load the Intel compiler.
By default, modules on HPC are compiled with OpenMPI

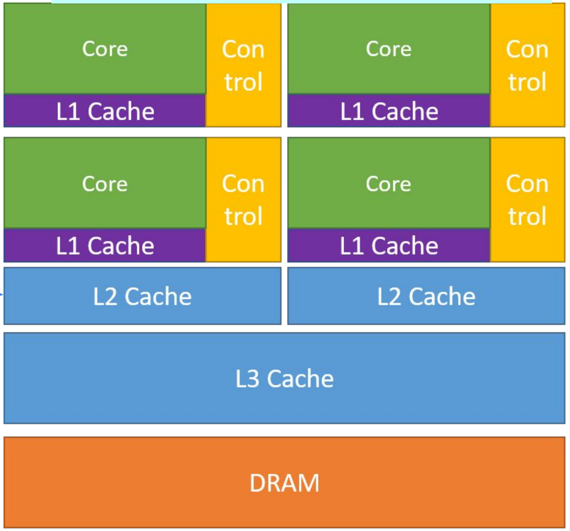
The University of Arizona

Parallel Computing **CPU** vs **GPU**

GPUs were made for parallel computing
→ ***very large number of less powerful cores***

CPU

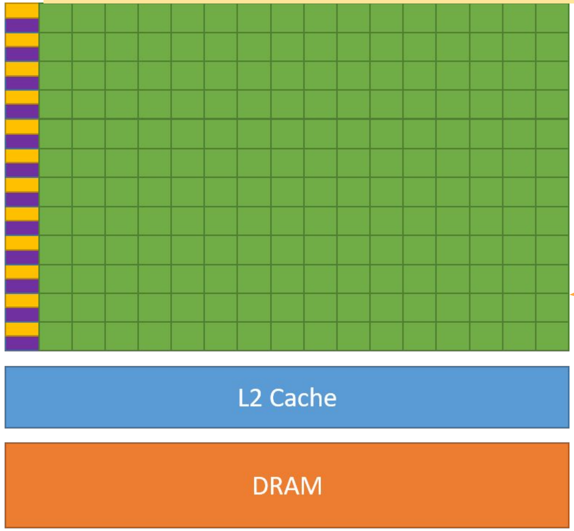
- Large caches
- Sophisticated control
- Powerful logic units



CPU

GPU

- Small cache
- Simple control
- Many Energy efficient logic units

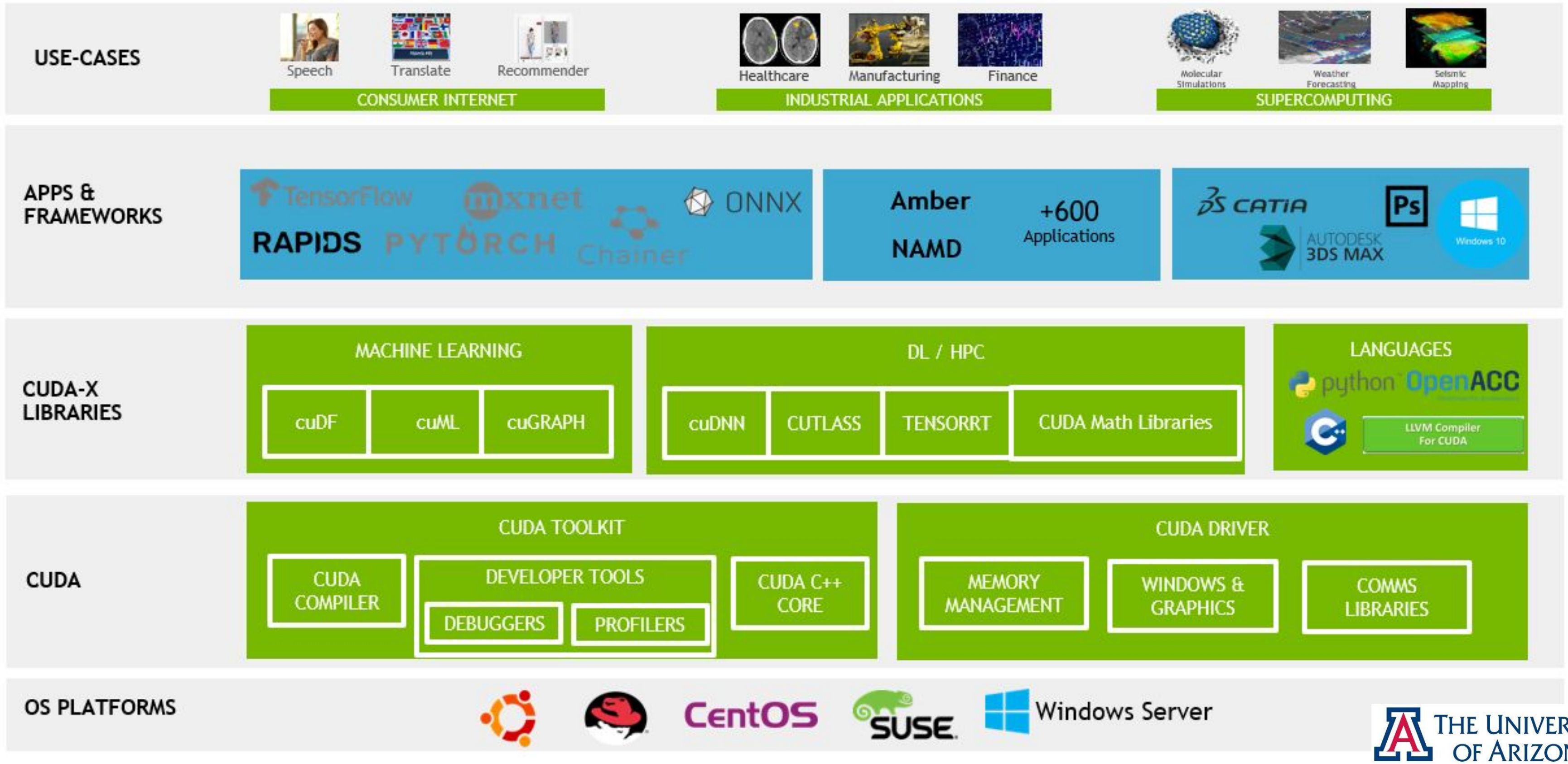


GPU

THE UNIVERSITY OF ARIZONA

Parallel Computing GPU

Nvidia has an elaborate and growing ecosystem based on **CUDA** which provides parallel support



THE UNIVERSITY OF ARIZONA

Parallel Computing GPU

CPUs for **sequential** code where latency matters

GPUs can be >20X faster for **parallel** code

Most of these applications are installed as modules on HPC

- Tensorflow
- PyTorch
- Matlab
- NAMD
- LAMMPS
- Quantum ESPRESSO
- Gromacs
- Relion
- Nvidia RAPIDS
- Julia
- Folding@home
- Caffe2
- Schrodinger

Nvidia V100

THE UNIVERSITY OF ARIZONA

Parallel Programming

The art of designing parallel algorithms, such as to calculate part of the Fibonacci series ..

$$F_n = \frac{\varphi^n - (-\varphi)^{-n}}{\sqrt{5}} = \frac{\varphi^n - (-\varphi)^{-n}}{2\varphi - 1}.$$

where

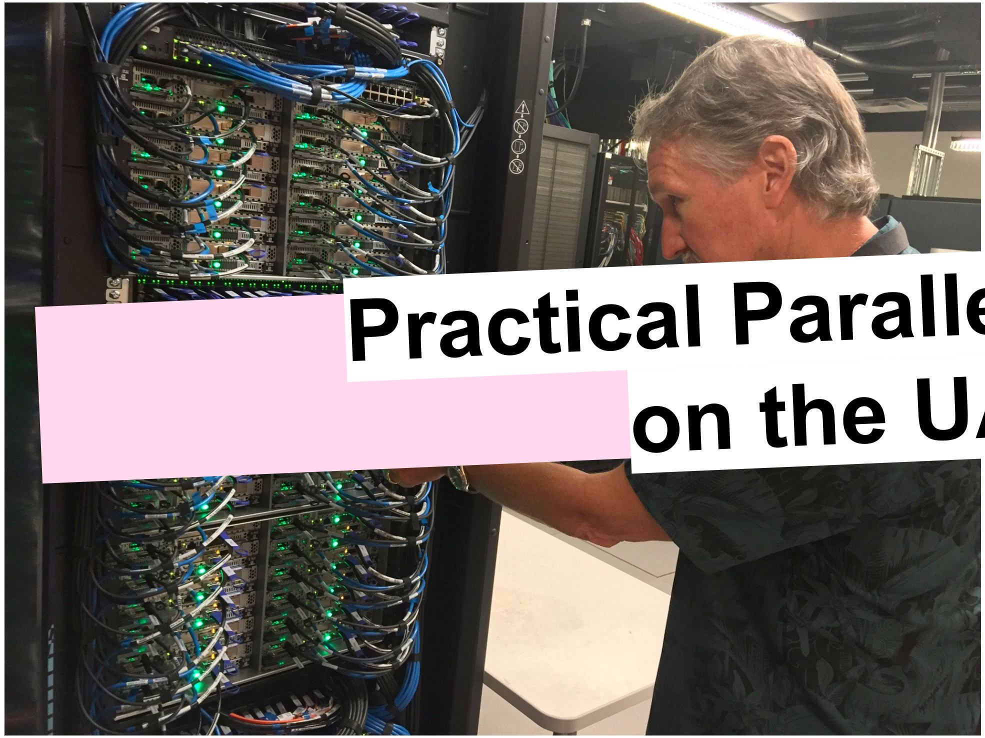
$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1.61803\,39887\ldots$$

… is beyond the scope of this workshop.

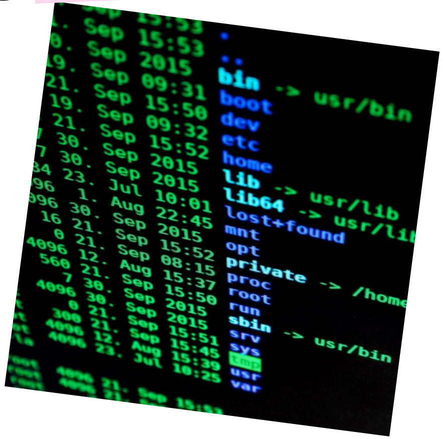
A more extensive overview of parallel programming can be found at:

https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial

THE UNIVERSITY OF ARIZONA



Practical Parallel Computing on the UA HPC



```
    Sep 15:53 .
 0. Sep 15:53 ..
19. Sep 2015  bin -> usr/bin
21. Sep 09:31 boot
19. Sep 15:50 dev
21. Sep 09:32 etc
30. Sep 15:52 home
30. Sep 2015  lib -> usr/lib
4 23. Sep 2015  lib64 -> usr/li
96  1. Jul 10:01 lost+found
096 30. Aug 22:45 mnt
16 21. Sep 2015  opt
0 21. Sep 15:52 private -> /home
4096 12. Sep 08:15 proc
560 21. Aug 15:37 root
7 30. Sep 15:50 run
4096 30. Sep 2015  sbin -> usr/bin
0 21. Sep 2015  srv
300 21. Sep 15:51 sys
4096 12. Sep 15:45 tmp
4096 23. Aug 15:39 usr
      Jul 10:25 var
```

UNIVERSITY INFORMATION TECHNOLOGY SERVICES
Research Computing

Parallelization Use Cases

Research Code End User

- Expert in domain science
- Som programming experience
- Inter h insights, writing publications
- Analysis consists of scripting and prewritten software packages
- **Needs functional understanding of parallelization to speed up analysis and produce results more quickly**

We can do this!

Research Software Engineer

- Majority of training in computer scien some domain knowledge/expe (varies)
- Interested in deve re for researcher
- Prim s git repository
- develops parallel algorithms

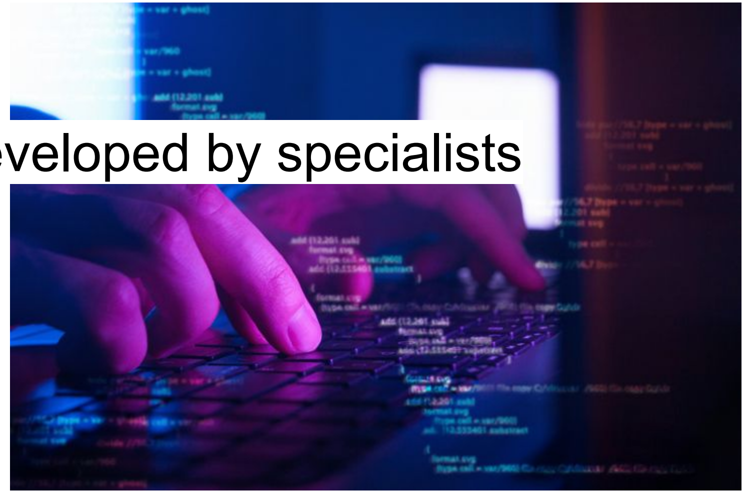
I can't teach you anything!

Realities of Developing Parallel Algorithms

- it can become very complex
- difficult to get right for non-trivial problems (weak scaling/strong scaling)

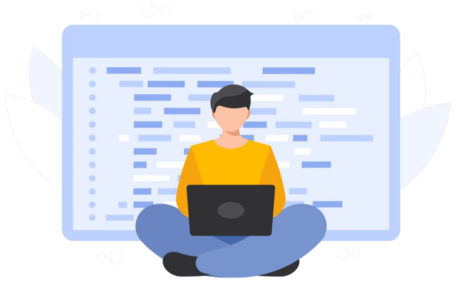
therefore..

- parallel programs for research developed by specialists
- implemented by researchers



Typical Research Workflow

1. Develop research question
2. Determine dataset
3. Find software that performs desired analysis
4. Download and learn software
5. Implement for research
    a. simulations generate data by using theoretical, empirical principles
    b. analysis software helps researchers extract useful information out of their experimental datasets using statistics, modeling, theory, etc



What does this mean?

Researchers often do not develop their own high performance analysis software from scratch

- tend to use existing modules and packages

- research software developers tackle the difficulties of implementing advanced computing algorithms

There are aspects of parallelization that are not always obvious or *well-communicated* to researchers

-> many programs *need to be told how to run in parallel*

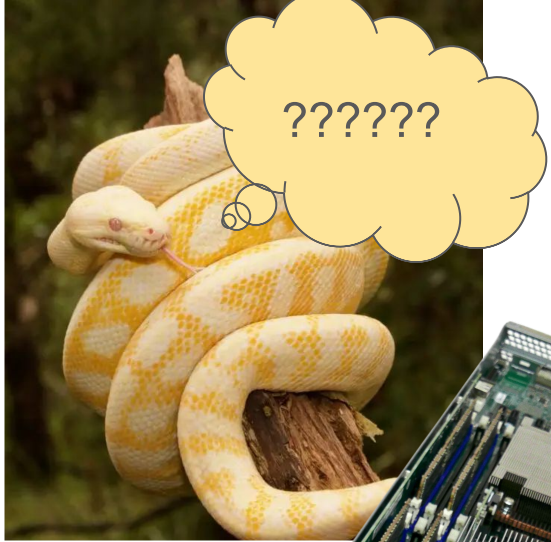
-> *it does not happy automagically!*

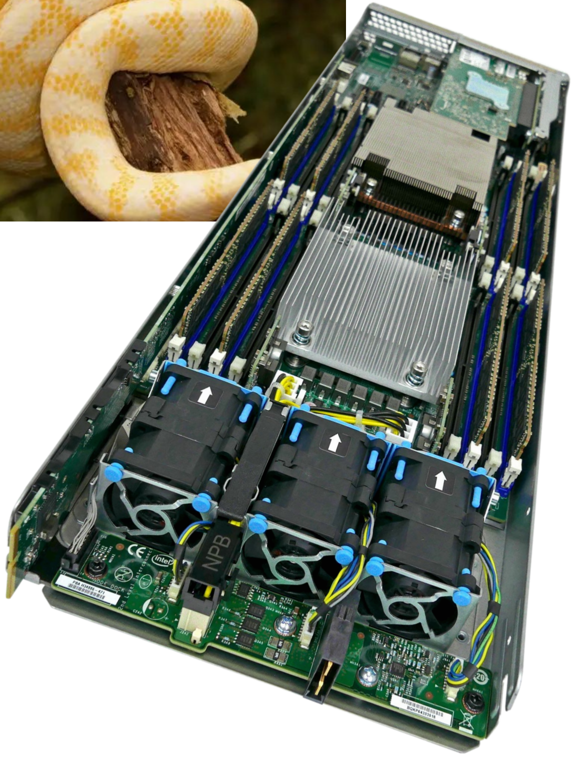
Check your software

- some programs are natively parallel
- ***many are not!***

MPI is necessary to facilitate multi-node parallelization

- programming languages like python, R, etc *do not automatically have information about the number of processors* available, nor how to communicate between nodes
- ***must implement proper packages to enable these features***





Adapting workloads, algorithms and code to parallel resources

- Simply dividing up a dataset and **running independent serial analyses** is *generally more efficient* than complex parallelization schemes

- Use **shared memory parallelism** when available (avoids internode communication overhead)

- Use **distributed memory parallelism** when one node does not provide enough memory or cores

Adapting workloads, algorithms and code to parallel resources

When writing or changing parallel code

- Do your homework
  - identify code hotspots
  - consider load balancing

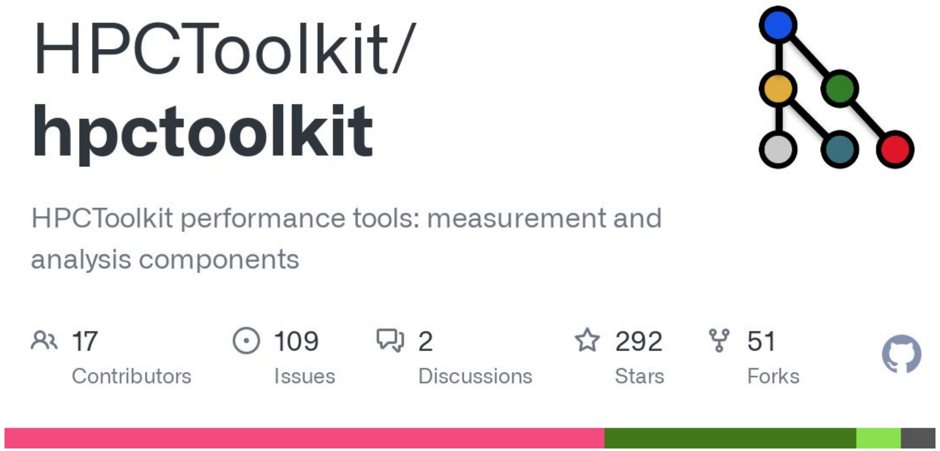
- Depending on language, algorithm, and type of parallel resources, efficiently parallelizing an algorithm can range between
  - Adding a few lines of code
  - Complete algorithm redesign

Other factors: not all numerical operations are equally fast

- integer < single precision FP < double precision FP
- addition < multiplication < division

Performance Analysis and Tuning

Installed as a module

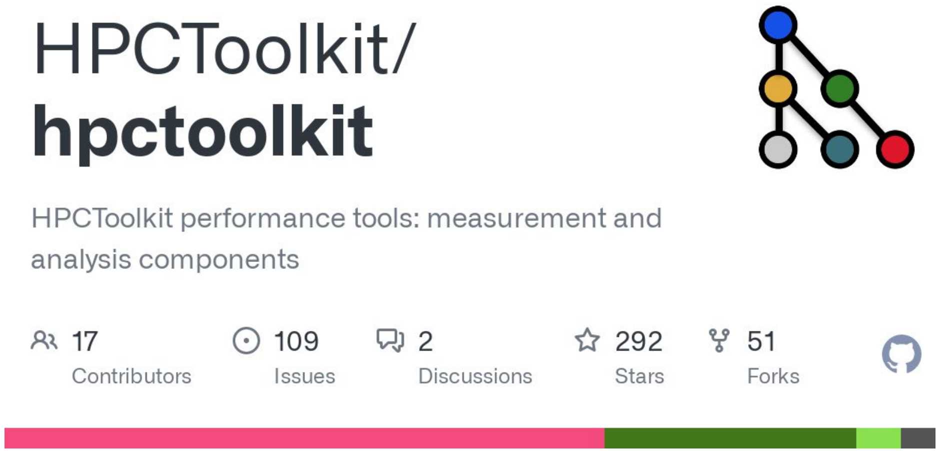


HPCToolkit performance tools: measurement and analysis components

An integrated suite of tools for measurement and analysis of program performance

Installed in operating system

Tools that can automatically detect many memory management and threading bugs, and profile your programs in detail.



Multithreaded program schematic

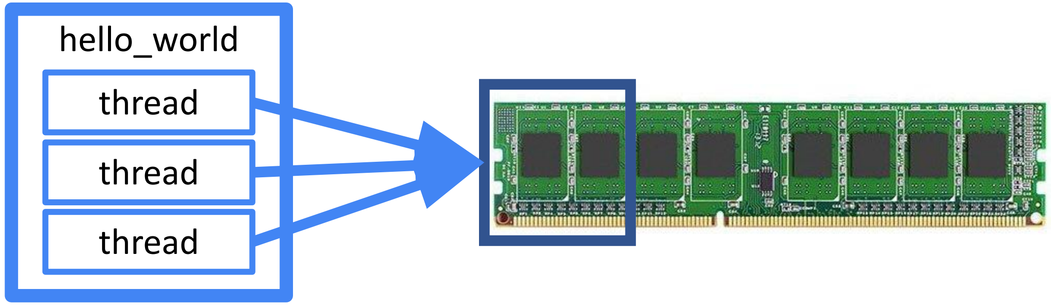
1 process (task)
with multiple threads

shared memory space

SLURM DIRECTIVES:

```
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=3
```

```
./hello_world <command line args>
```



Thread # control at program level:

- OpenMP: `export OMP_NUM_THREADS=3`
- Command line argument

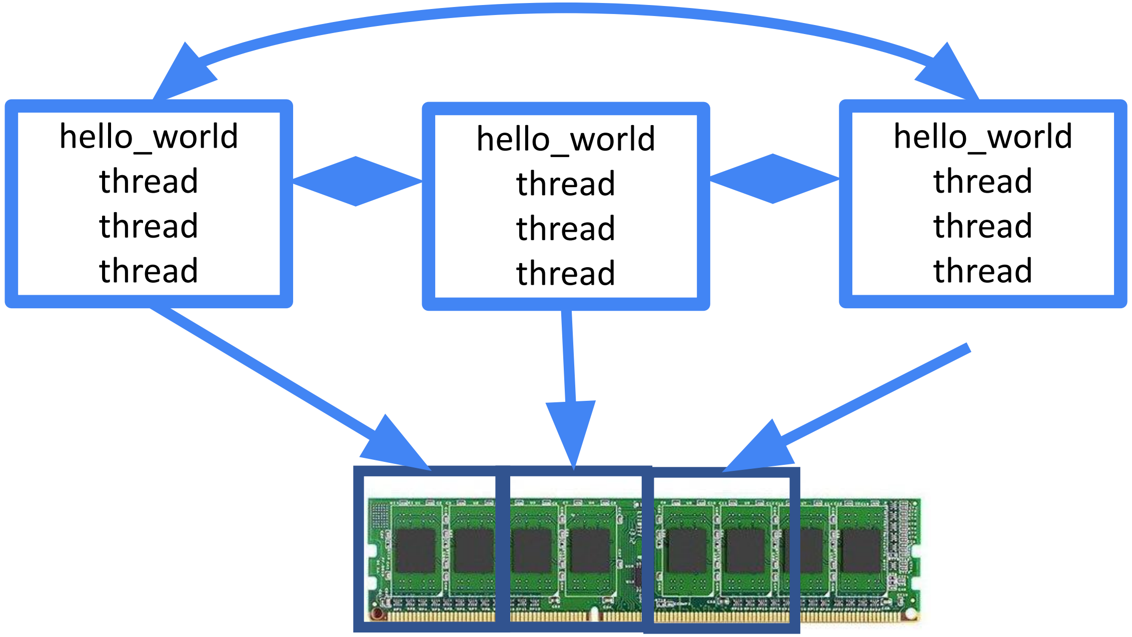
OpenMP is probably the easiest (but not only) method for creating multithreaded programs

MPI schematic

- 3 MPI processes (tasks)
  - Each potentially multithreaded
  - Independent memory spaces
  - May be on different nodes

```
mpirun -np 3 ./hello_world
```

MPI API allows many internode communication methods



RELEVANT SLURM DIRECTIVES

```
#SBATCH --ntasks=3
#SBATCH --cpus-per-task=3
#SBATCH --nodes=#
#SBATCH --tasks-per-node=#
```

Parallel Programming Examples!

Array Jobs

Array jobs allow for meta-level parallelization.

Array jobs are useful if you have to run the same analysis on many different data sets, and if the order of completion does not matter

DO NOT USE FOR LOOPS TO SUBMIT JOBS – USE ARRAY JOBS

Parallel Computing on HPC – job arrays

Using an **array** to submit multiple independent jobs

Instead of this:

```
for i in $( seq 1 10 ); do sbatch script.slurm <submission options> ;done
```

Do this:

```
#!/bin/bash
#SBATCH --ntasks=1
#SBATCH --nodes=1
#SBATCH --time=00:01:00
#SBATCH --partition=standard
#SBATCH --account=YOUR_GROUP
#SBATCH --array 1-10

echo "./sample_command input_file_${SLURM_ARRAY_TASK_ID}.in"
```

Seriously, please do not do this!

The above assumes that you have input files named `input_file_1.in, input_file_2.in, etc`

https://ua-researchcomputing-hpc.github.io/Array-and-Parallel/Basic-Array-Job/

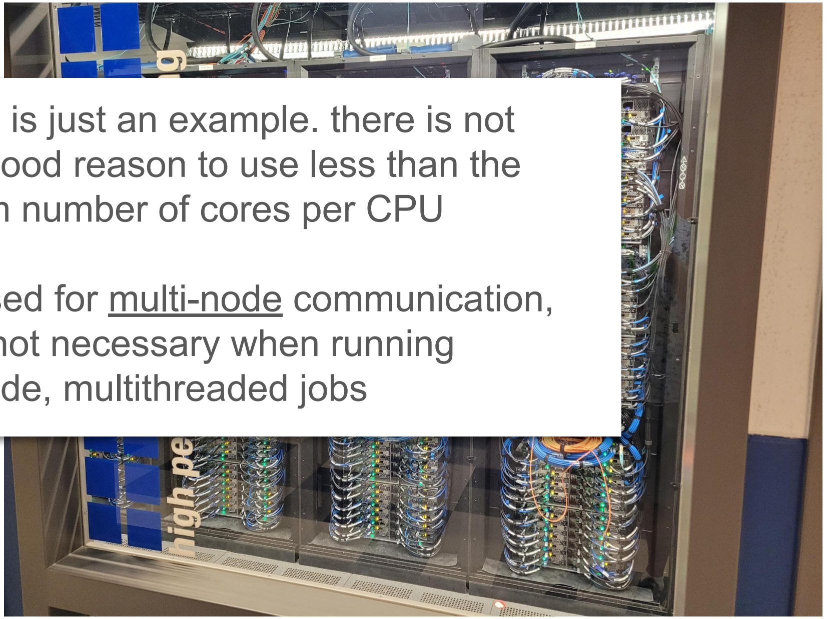
THE UNIVERSITY OF ARIZONA

Parallel Computing on HPC - MPI

MPI job submission - *Hello World*

This example Slurm script runs the Hello World executable on 10 cores on each of 3 nodes

```
#!/bin/bash
#SBATCH --job-name=Multi-Node-MPI-Job
#SBATCH --ntasks=30
#SBATCH --nodes=3
#SBATCH --ntasks-per-node=10
#SBATCH --time=00:01:00
#SBATCH --partition=standard
#SBATCH --account=YOUR_GROUP

module load gnu8 openmpi3

mpicc -o hello_world hello_world.c

mpirun -np $SLURM_NTASKS ./hello_world
```



note: this is just an example. there is not really a good reason to use less than the maximum number of cores per CPU

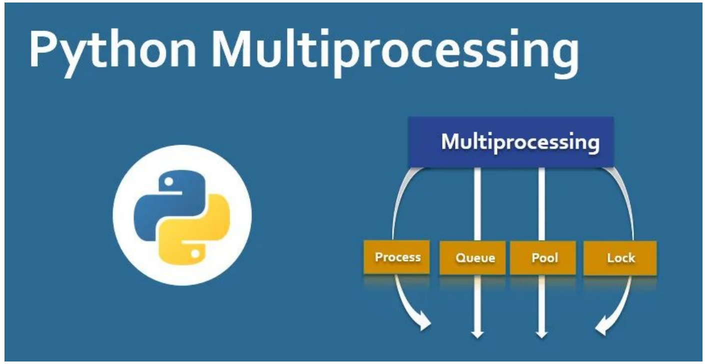
MPI is used for multi-node communication, and it is not necessary when running single-node, multithreaded jobs

https://ua-researchcomputing-hpc.github.io/MPI-Examples/Multi-Node-MPI-Job/

THE UNIVERSITY OF ARIZONA

Other tools!

For people who live life in the parallel lane



Parallel Computing on HPC - GNU parallel

Using GNU parallel to parallelize multiple tasks within one command

GNUparallel

For people who live life in the parallel lane

Access Compute Node

Use either batch job or interactive

```
$ elgato
$ interactive -a <your_group> -n 8
$ module load parallel
$ seq 1 100 | parallel 'DATE=$( date +"%T" ) && sleep 0.{} && echo \
        "Host: $(hostname) ; Date: $DATE; {}"'
```

Output

```
Host: junonia.hpc.arizona.edu ; Date: 15:47:06; 1
Host: junonia.hpc.arizona.edu ; Date: 15:47:06; 2
Host: junonia.hpc.arizona.edu ; Date: 15:47:06; 3
Host: junonia.hpc.arizona.edu ; Date: 15:47:06; 4
Host: junonia.hpc.arizona.edu ; Date: 15:47:06; 5
Host: junonia.hpc.arizona.edu ; Date: 15:47:06; 6
Host: junonia.hpc.arizona.edu ; Date: 15:47:07; 10
Host: junonia.hpc.arizona.edu ; Date: 15:47:06; 7
Host: junonia.hpc.arizona.edu ; Date: 15:47:07; 11
```

https://ua-researchcomputing-hpc.github.io/Array-and-Parallel/Basic-Parallel-Job/

The University of Arizona

Parallel Computing on HPC - GNU parallel

Parallel will create as many jobs as inputs:

```
parallel echo {#} ::: A.txt B.txt C.txt D.txt E.txt
1
2
3
4
5
```

Limit number of jobs:

```
parallel -j 2 echo {%} ::: A.txt B.txt C.txt D.txt E.txt
1
2
1
2
1
```

GNUparallel

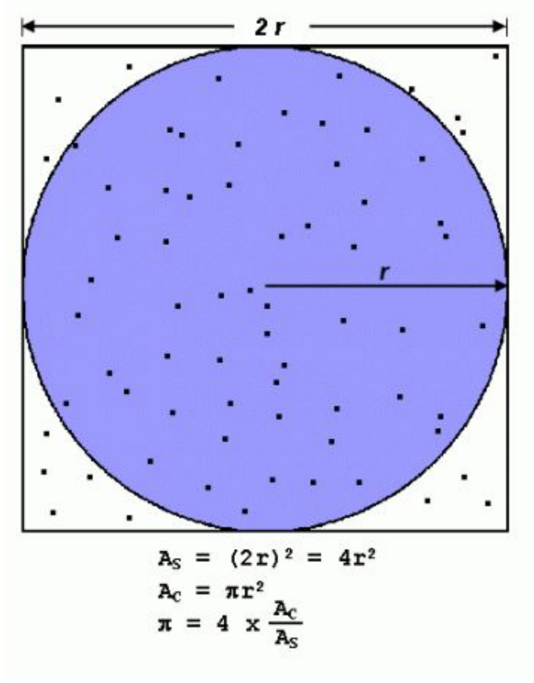
For people who live life in the parallel lane

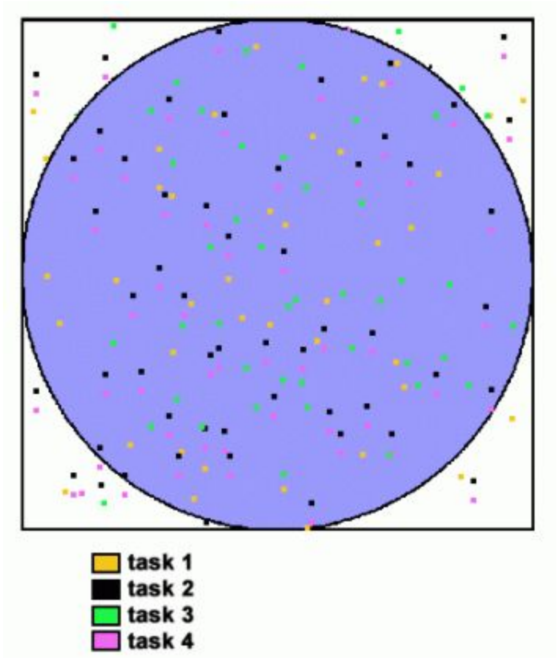
Tons of examples online:
https://www.gnu.org/software/parallel/parallel_examples.html

Parallel Computing on HPC - Python

A not-completely-trivial example of a parallelized calculation in Python

We can approximate Pi with a Monte-Carlo Simulation to guess area of circle

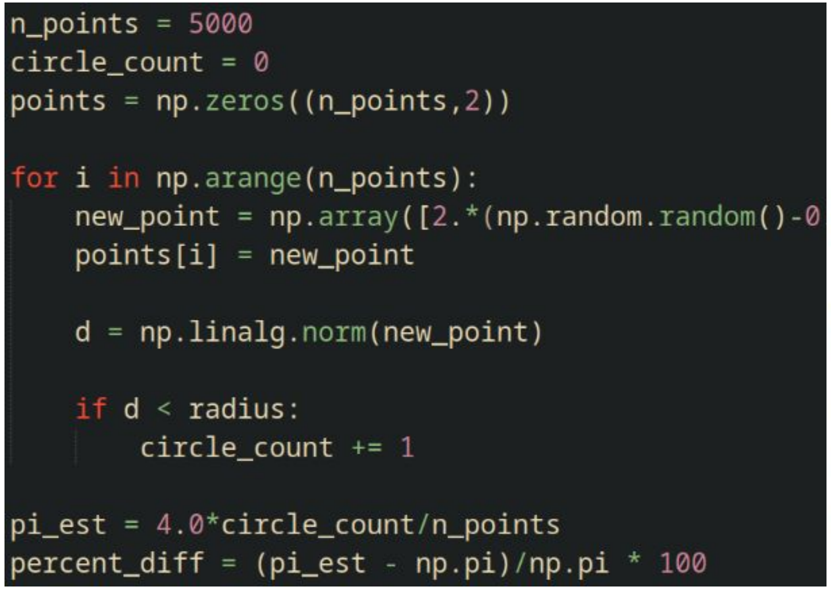




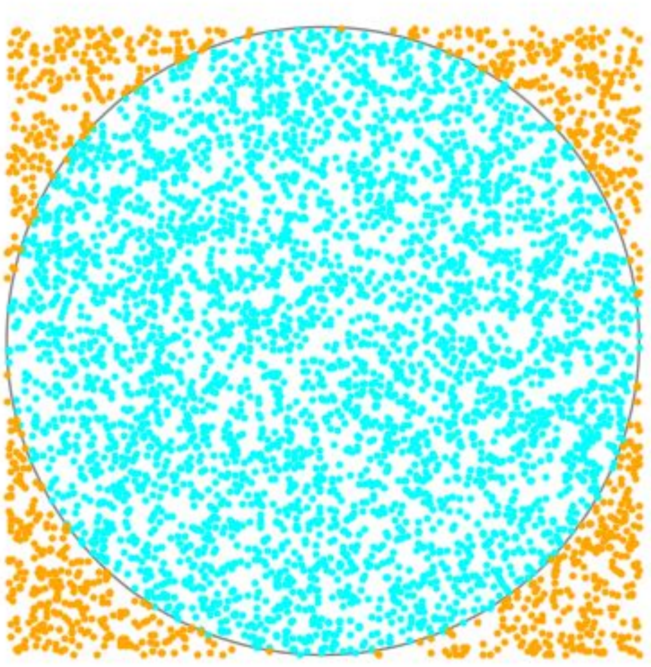
source: https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial##ExamplesPI

Parallel Computing on HPC - Python

Serial Version:

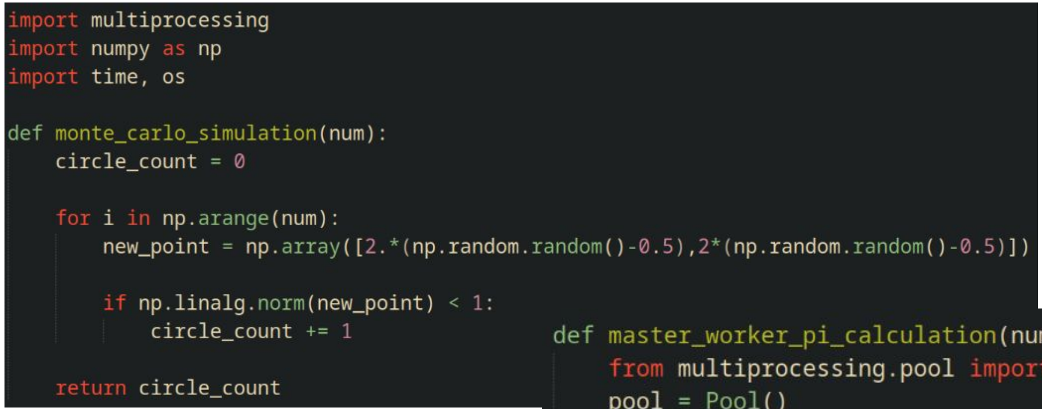


```
n_points = 5000
circle_count = 0
points = np.zeros((n_points,2))

for i in np.arange(n_points):
    new_point = np.array([2.*(np.random.random()-0
    points[i] = new_point

    d = np.linalg.norm(new_point)

    if d < radius:
        circle_count += 1

pi_est = 4.0*circle_count/n_points
percent_diff = (pi_est - np.pi)/np.pi * 100
```

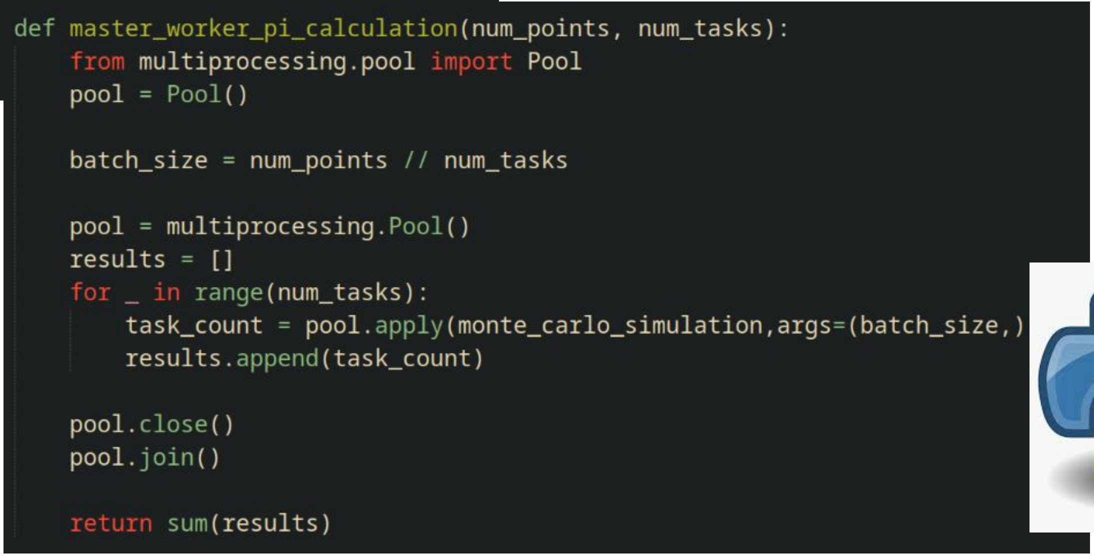


code available: https://github.com/ethan-jahn/picalc_example

Parallel Computing on HPC - Python



```
import multiprocessing
import numpy as np
import time, os

def monte_carlo_simulation(num):
    circle_count = 0

    for i in np.arange(num):
        new_point = np.array([2.*(np.random.random()-0.5),2*(np.random.random()-0.5)])

        if np.linalg.norm(new_point) < 1:
            circle_count += 1

    return circle_count
```

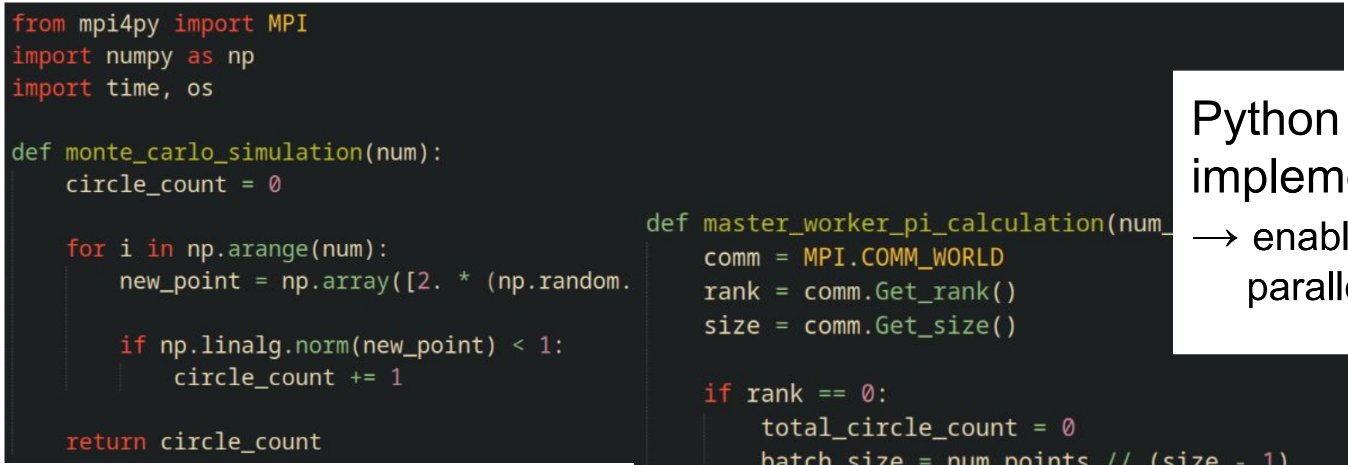


```
def master_worker_pi_calculation(num_points, num_tasks):
    from multiprocessing.pool import Pool
    pool = Pool()

    batch_size = num_points // num_tasks

    pool = multiprocessing.Pool()
    results = []
    for _ in range(num_tasks):
        task_count = pool.apply(monte_carlo_simulation,args=(batch_size,)
        results.append(task_count)

    pool.close()
    pool.join()

    return sum(results)
```

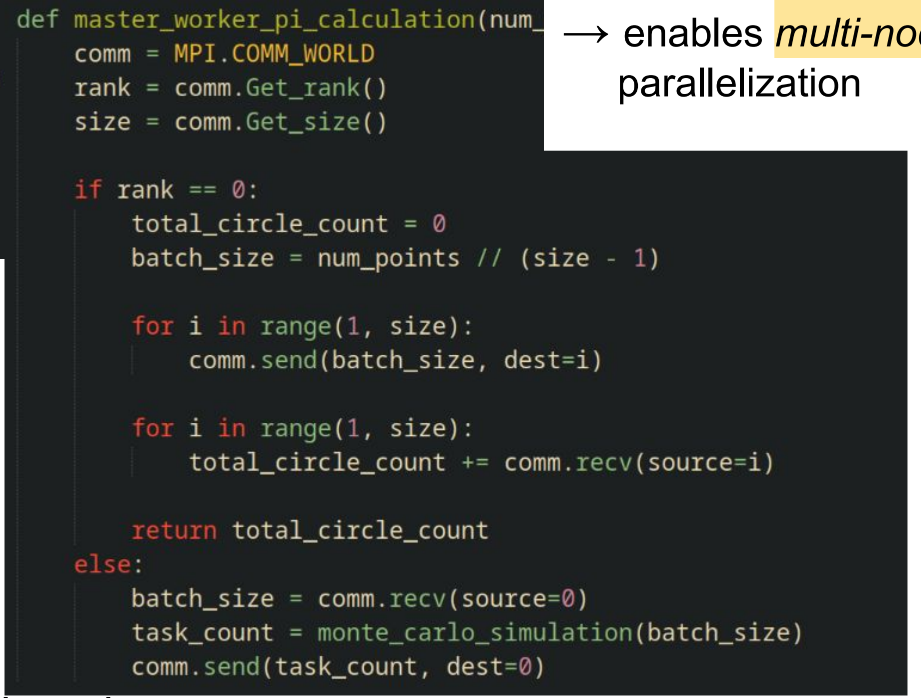
Python “**multiprocessing**” library implementation
⟶ enables *single-node* parallelization

code available: https://github.com/ethan-jahn/picalc_example

Parallel Computing on HPC - Python



```python
from mpi4py import MPI
import numpy as np
import time, os

def monte_carlo_simulation(num):
    circle_count = 0

    for i in np.arange(num):
        new_point = np.array([2. * (np.random.

        if np.linalg.norm(new_point) < 1:
            circle_count += 1

    return circle_count
```



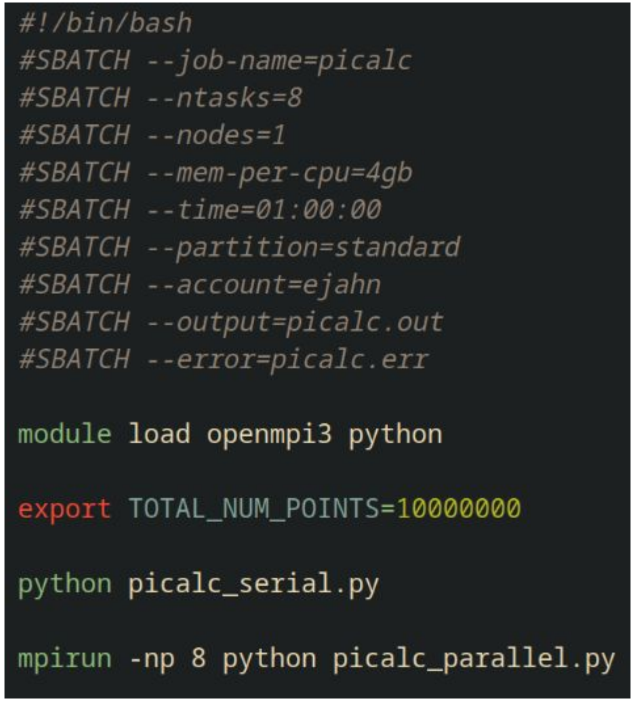
```python
def master_worker_pi_calculation(num_
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    if rank == 0:
        total_circle_count = 0
        batch_size = num_points // (size - 1)

        for i in range(1, size):
            comm.send(batch_size, dest=i)

        for i in range(1, size):
            total_circle_count += comm.recv(source=i)

        return total_circle_count
    else:
        batch_size = comm.recv(source=0)
        task_count = monte_carlo_simulation(batch_size)
        comm.send(task_count, dest=0)
```

Python “**mpi4py**” library implementation
⟶ enables *multi-node* parallelization

code available: https://github.com/ethan-jahn/picalc_example

Parallel Computing on HPC - Python

Example batch script



```bash
#!/bin/bash
#SBATCH --job-name=picalc
#SBATCH --ntasks=8
#SBATCH --nodes=1
#SBATCH --mem-per-cpu=4gb
#SBATCH --time=01:00:00
#SBATCH --partition=standard
#SBATCH --account=ejahn
#SBATCH --output=picalc.out
#SBATCH --error=picalc.err

module load openmpi3 python

export TOTAL_NUM_POINTS=10000000

python picalc_serial.py

mpirun -np 8 python picalc_parallel.py
```

Parallel Computing on HPC - Python Multiprocessing Library

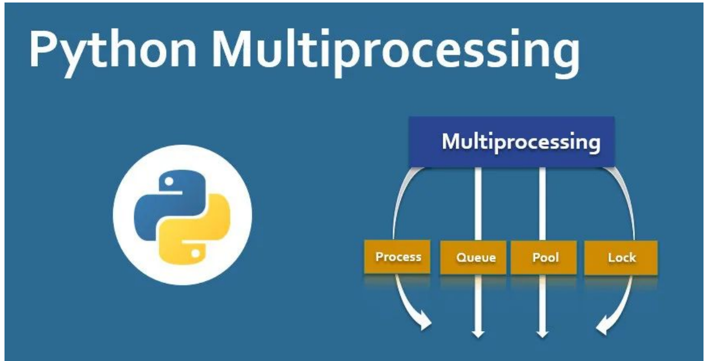
Estimate pi using a monte carlo simulation:
https://github.com/gelatinous-astronaut/picalc_example

On HPC:
Start an interactive session

```
elgato
interactive -a <your_group> -n 8
```

Set up environment

```
git clone https://github.com/gelatinous-astronaut/picalc_example.git
cd picalc_example
module load python
python3 -m venv --system-site-packages </path/to/env>
python3 -m pip install -upgrade pip
python3 -m pip install mpi4py multiprocessing
```



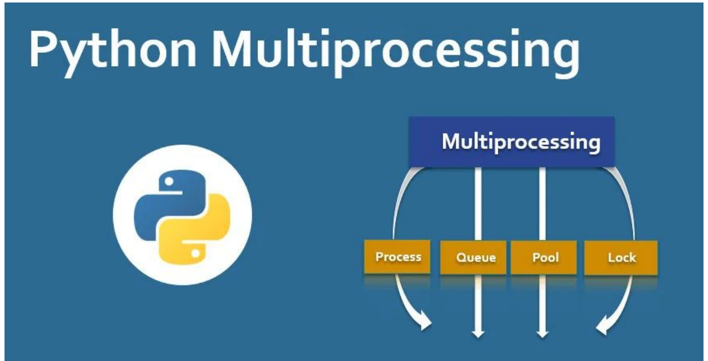
Parallel Computing on HPC - Python Multiprocessing Library

Estimate pi using a monte carlo simulation:
https://github.com/gelatinous-astronaut/picalc_example

On HPC:

Run the code

```
python3 picalc_serial.py

python3 picalc_multiprocessing.py

mpirun -n 8 python3 picalc_mpi4py.py
```



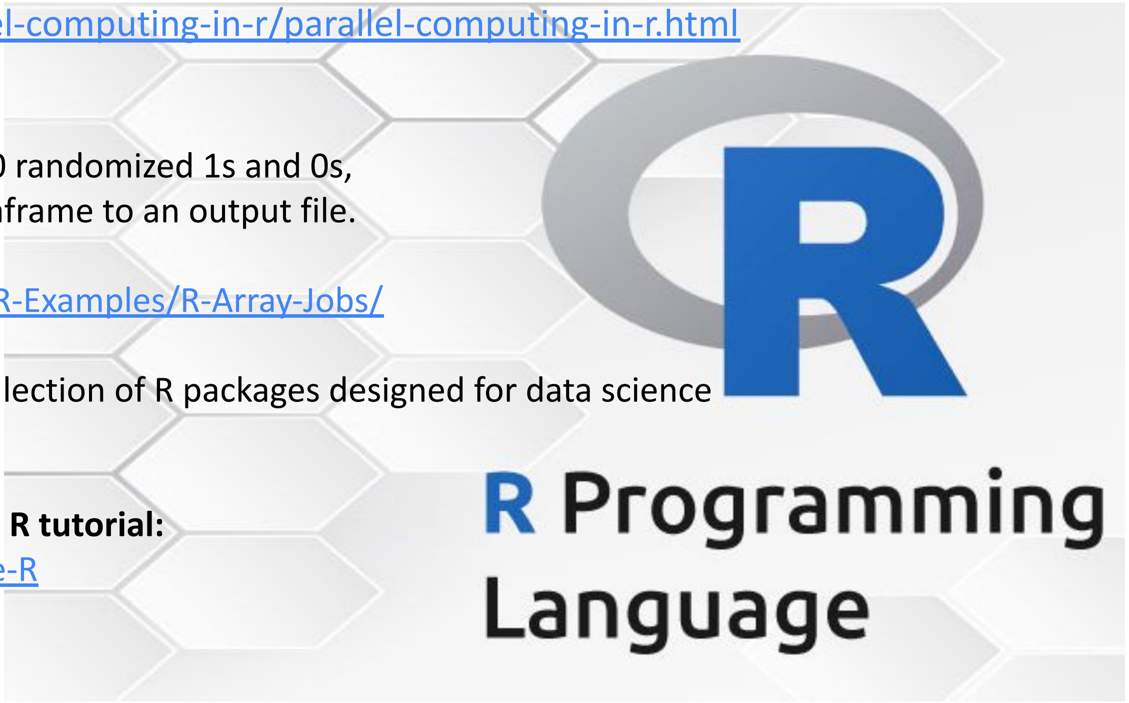
Parallel Computing on HPC - R

Quick Intro to Parallel Computing in R
https://nceas.github.io/oss-lessons/parallel-computing-in-r/parallel-computing-in-r.html

Using an Array with an R script
You can create an R script that generates 1000 randomized 1s and 0s, store them as a dataframe, then save the dataframe to an output file. Then run this R script as an array job.
https://ua-researchcomputing-hpc.github.io/R-Examples/R-Array-Jobs/

Check out the **tidyverse** – an opinionated collection of R packages designed for data science
install.packages("tidyverse")

For an excellent hands-on Parallel Analysis in R tutorial:
https://github.com/ljdursi/beyond-single-core-R
It covers these packages:
parallel, foreach, bigmemory, Rdsm, pbdR



Parallel Computing References

Introduction to Parallel Computing Tutorial

Author: Blaise Barney, Livermore Computing (retired), Donald Frederick, LLNL
https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial##Overview

Recommended reading

"Introduction to Parallel Computing", Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar.

University of Oregon - Intel Parallel Computing Curriculum https://ipcc.cs.uoregon.edu/curriculum.html

An Introduction to Linux - https://cvw.cac.cornell.edu/Linux/

Linux Tutorial for Beginners: Introduction to Linux Operating System (link)

"Introduction to Linux" - Boston University (link)

"Parallel Processing in Python:A Practical Guide with Examples", Selva Prabhakaran (link)