Introduction to Parallel Programming Concepts

Summer 2026

Research Computing Services
IS & T

Outline

- Parallel Algorithms
- Hardware
- Parallel Implementations
- Processes and Threads
- Libraries
- Your code
- Pitfalls

Introduction

- Many programs can perform simultaneous operations, given multiple processors to perform the work.

- Generally speaking, the burden of managing this lies on the programmer:
  - Implement parallel code in the programming language
  - Make use of implicit parallelization in programming languages
  - Indirectly by using libraries that perform parallel calculations.
  - Deliberately by choosing libraries or software systems that assist in running parallel code.

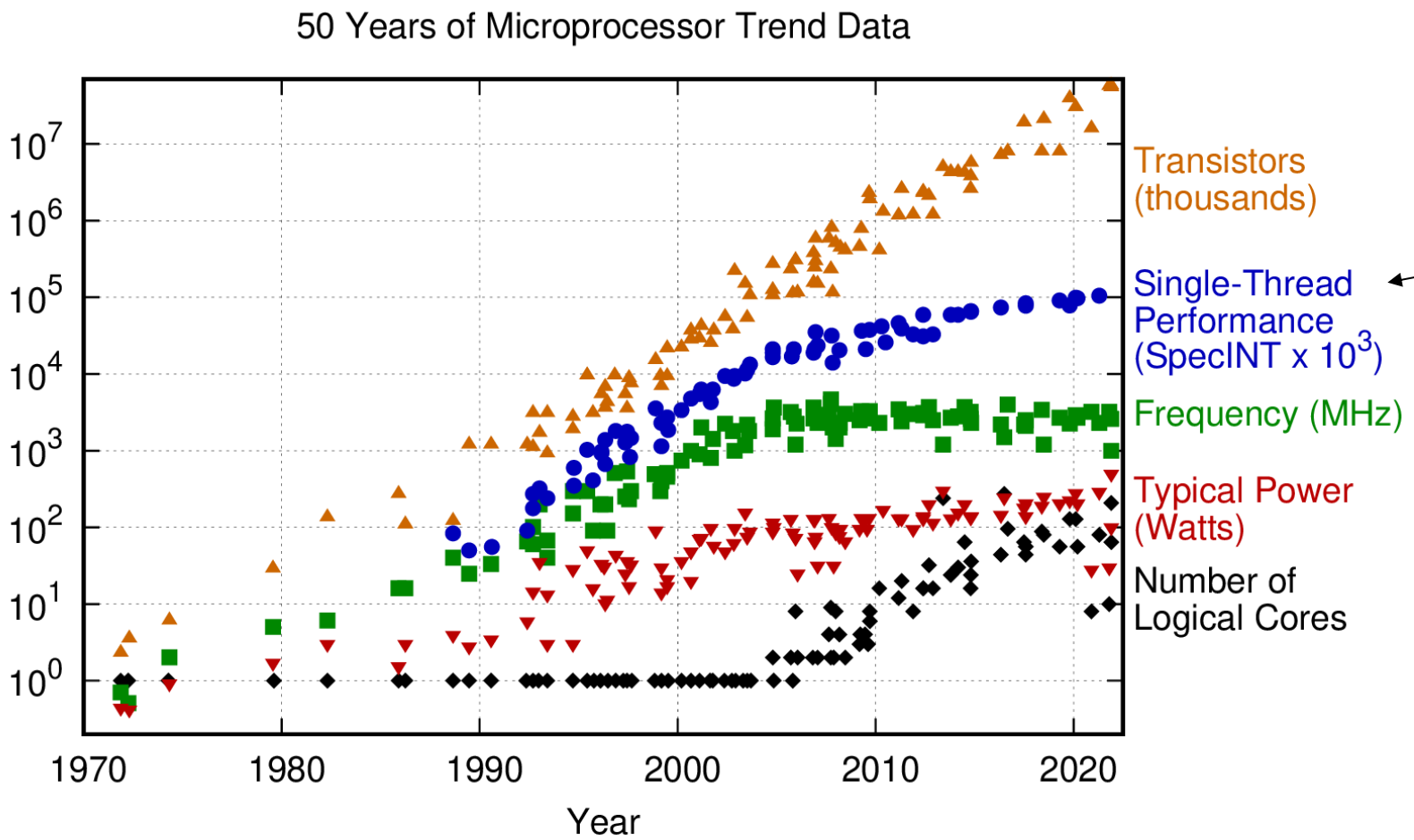
Limits (“bounds”) on Program Speed

- **Input/Output** (I/O): The rate at which data can be read from a disk, a network file server, a remote server, a sensor, a user’s physical inputs, etc. limits the performance of the program.

- **Memory**: The quantity of memory on the system limits performance.
  - Example: a computer has 16 GB of RAM, a data file is 64 GB in size.

- **CPU** (or compute): The speed of the processor is the limit on performance.

Why Parallelize?



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

- Processors are not getting much faster these days

Year start/end	SPECINT ratio
1990-1995	12.9
1995-2000	5.7
2000-2005	3.7
2005-2010	1.7
2010-2015	1.8
2015-2020	1.5

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

Hands-On Parallel Sorting

- Task:
  - take a shuffled deck of cards.
  - Sort it into the 4 suits in this order: Hearts ♥, Clubs ♣, Diamonds ♦, Spades ♠
    - Within each suit sort all the cards: Ace, 2 thru 9, Jack, Queen, King
  - You are finished when your deck is in the specified sorted order.

- Step 1: give your cards a few shuffles.
- Step 2: Decide how you'll proceed.
- Step 3: You will be timed, go when told to do so.
- We'll do this individually and in groups and will time the results.

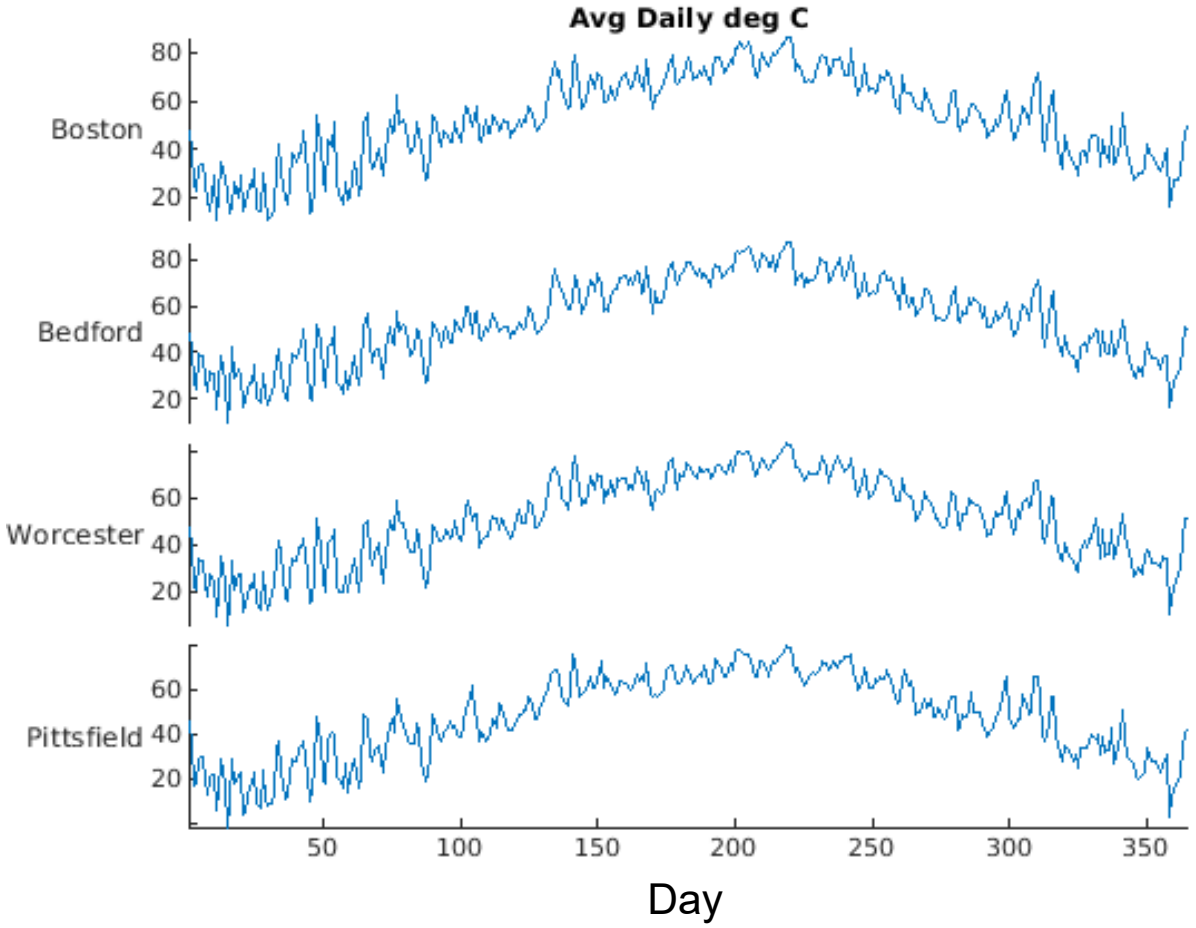
BOSTON UNIVERSITY

Some things can’t be done in parallel.

- Gestation time for 1 female elephant to produce 1 calf: 18 months.
- 18 elephants cannot produce a calf in 1 month.

Example 1: Daily Average Temperatures

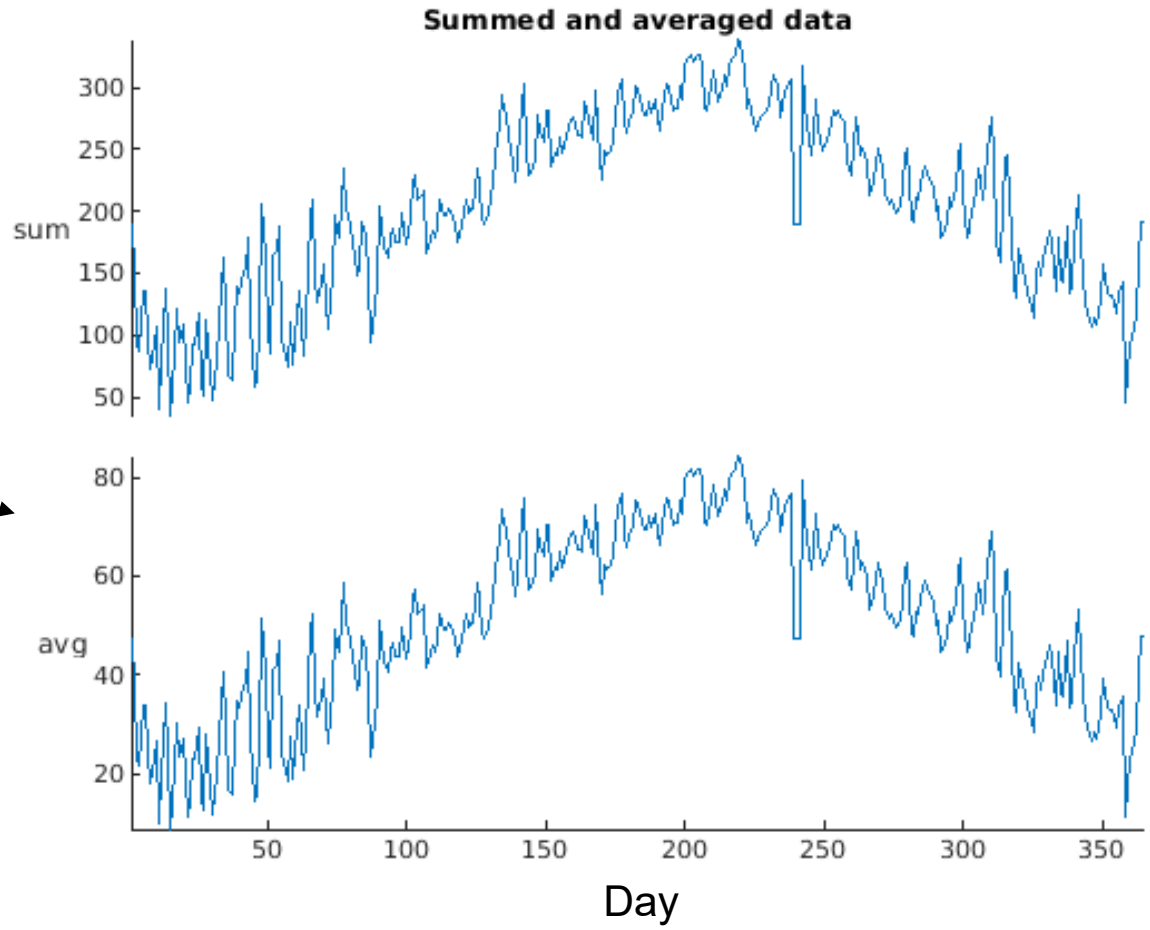
- Each row is the average daily temperature from 4 airports in Massachusetts for 2022.
- We want to find the average of the daily temperature across all the airports.



Example 1: Daily Average Temperatures

- Serial calculation:
  - For each day:
    - add the temperatures for each airport
    - then divide by 4.

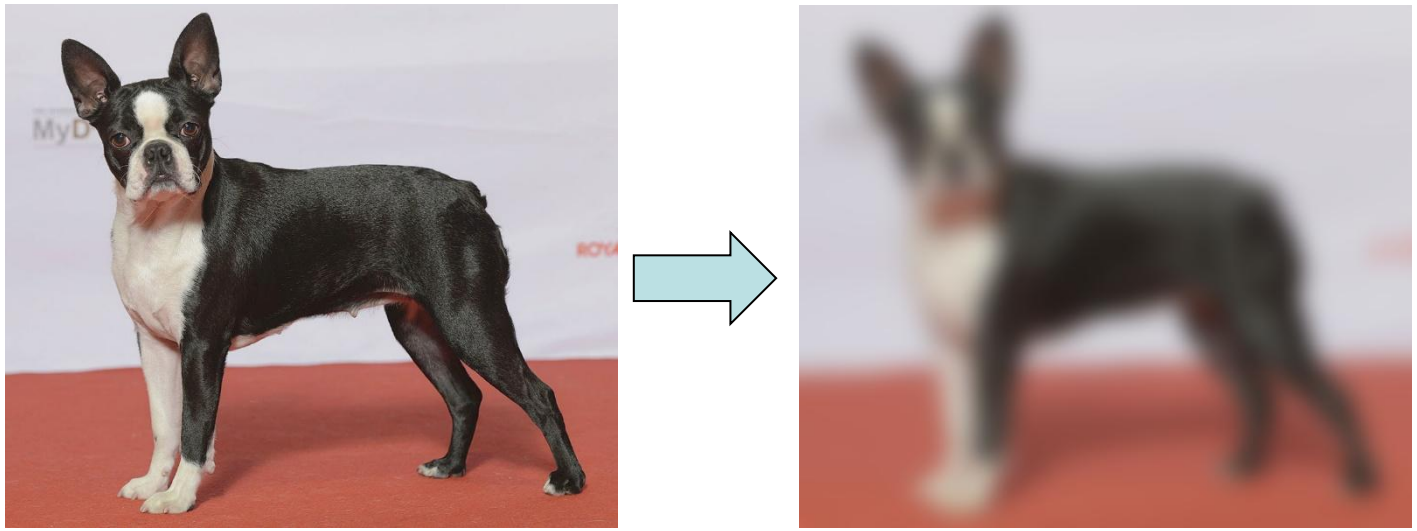
- Given 2 computer processors, how can we do this so that they do this computation together?

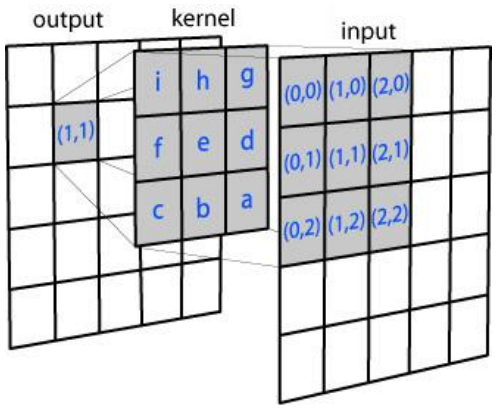


BOSTON UNIVERSITY

Example 2: Gaussian Image Blurring

- A selected block of pixels is multiplied by numeric values in a *kernel* and summed to produce a single output value.
- That value is written to the output image.
- The selection is moved by 1 row (or 1 col) and the new block of pixels is again multiplied to get a new value, and so on.



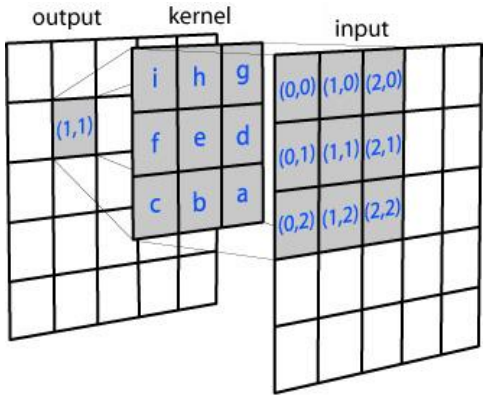


BOSTON UNIVERSITY

Example 2: Gaussian Image Blurring

- How to parallelize this? Let's use 4 processors.

- The image is 1033x882 pixels. The kernel is 20x20 pixels.



BOSTON UNIVERSITY

Example 3: *k*-mer counting

- *k*-mers are repeated sets of nucleotides in genomic sequences. *k* is the length of the set.
- AGTCCC
  - Split into *k*-mers of length 3: AGT, GTC, TCC, CCC

- A common problem in genomics is creating a histogram of all possible *k*-mers from a data file for a given length.

```
AGTCCCCGTCTTGCCGCGCGGGGGCGGGCGCGGGAAAAAAGCCGCGCGGGGGCGC
CCGCGGGAAGGCAGCCCCGCGGCGCGCGGGGGGAGGGGCGGCGCCCGCGGGGGAG
CGGCCGGCTCCGGGGGAGGGACGGGGAAGGGGGCGCGCGGGGCTGCCCTGCCGCC
CGCCCGCCGCCGCCGCCCGCCTTCGCGCCCCCCCCCAAAAAACACCCCCCCCGGA
```

…imagine this in a file a few dozen GB in size…

BOSTON UNIVERSITY

Example 3: *k*-mer counting

How can we split this up into parallel computations?

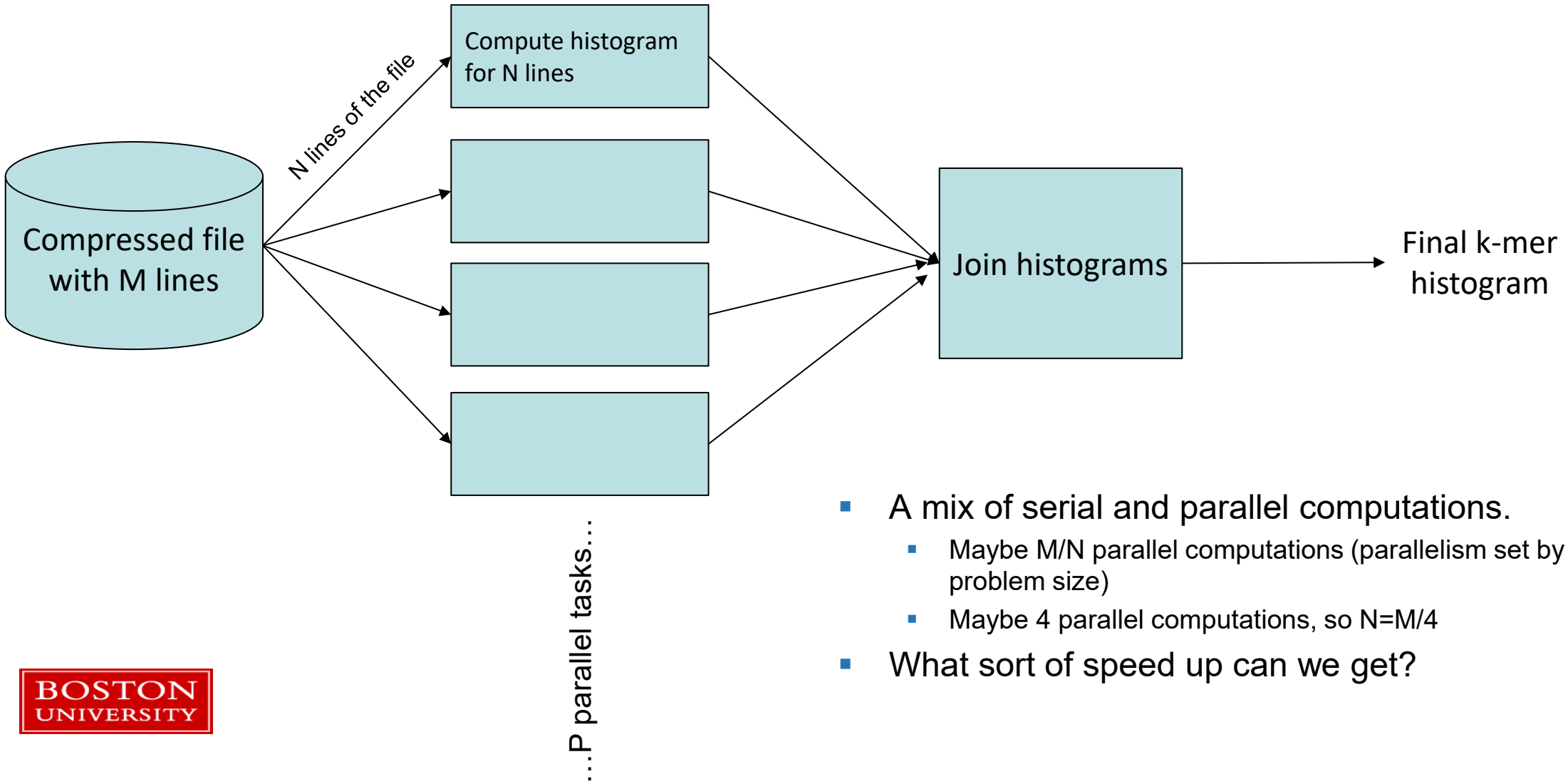
Which steps can happen in parallel?

- Tasks:
  - Read each line from the file. The file is compressed to save disk space.
  - In each line, find all possible *k*-mers for a fixed value *k*.
  - Store all *k*-mers that are found and how often they occurred.
  - Repeat for the next line.
  - The output is the histogram for the whole file:

3-mer	Occurrences
AGT	203
GTC	123
TCC	583
CCC	875

…etc…

Example 3: *k*-mer counting



- A mix of serial and parallel computations.
  - Maybe M/N parallel computations (parallelism set by problem size)
  - Maybe 4 parallel computations, so N=M/4
- What sort of speed up can we get?

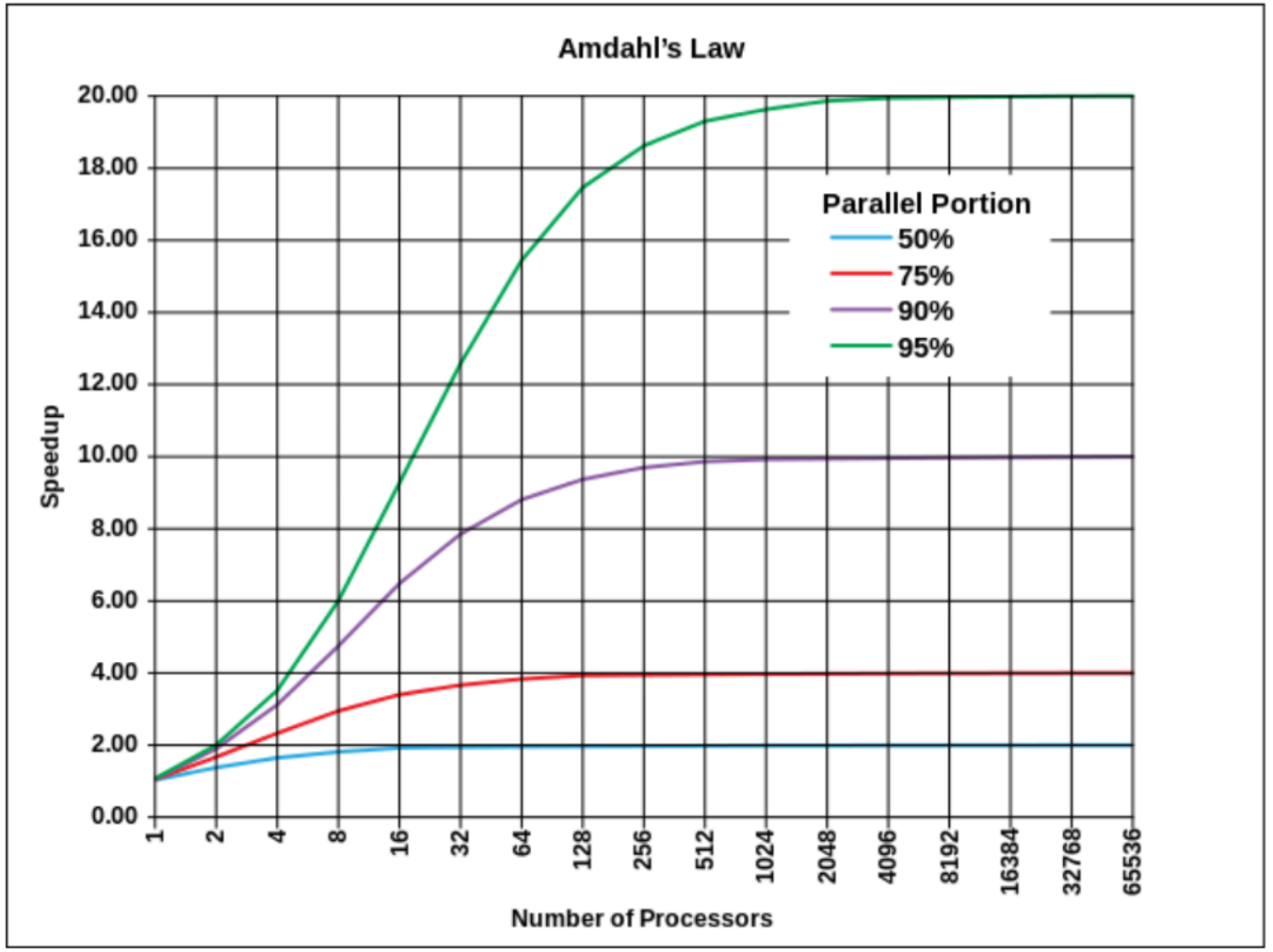
BOSTON UNIVERSITY

Amdahl's Law

- The speedup ratio *S* is the ratio of time between the serial code (T_1) and the time when using N workers (T_N):

$$S = \frac{T_1}{T_N} = \frac{T_1}{\left(f + \frac{1-f}{N}\right) T_1}$$

N = number of threads or processes
f = fraction of program that is serial



- This is the **theoretical** best speedup achievable with parallelization.

Figure from Wikipedia.

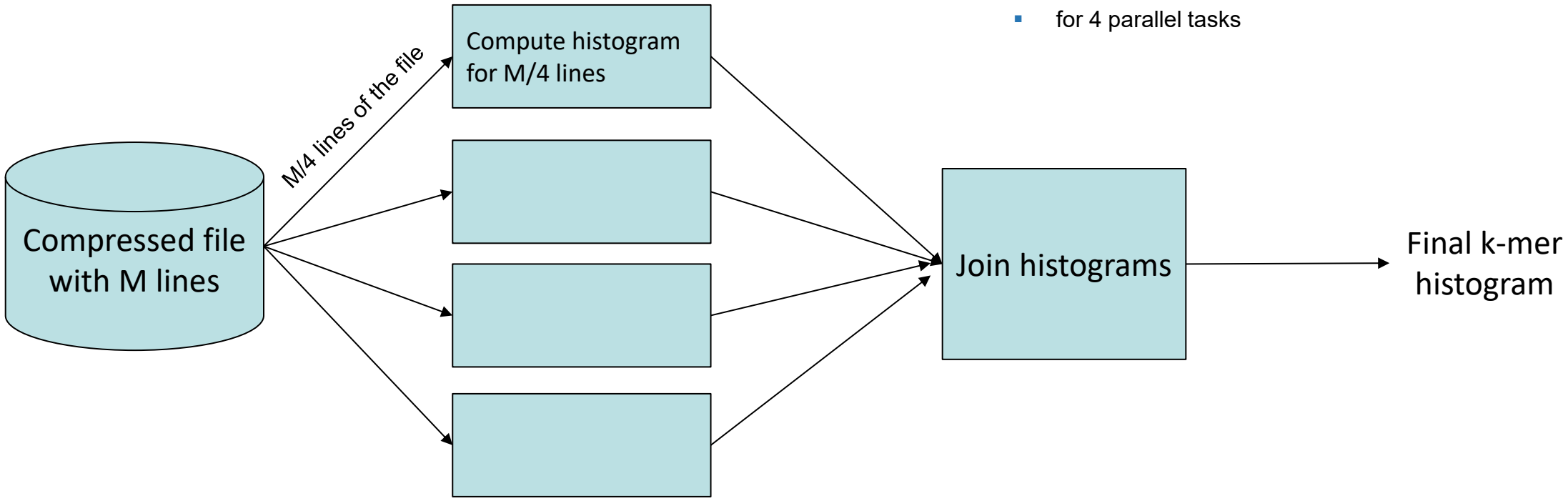
The basics of the Work-Depth Model

- An abstract way to think about parallel algorithms
  - Abstract away number of processors, I/O, and so on so the analysis is independent of implementation, inter-process communication, etc.
  - This can be connected back to Amdahl's Law for parallel speedups.
- Work ($\mathbf{T_1}$): total amount of tasks to complete for a single processor
- Depth ($\mathbf{T_\infty}$): longest length of serial computations that must be performed.
  - i.e. the time taken if you have an infinite number of parallel computations so that their computation time can be ignored.
- Maximum speedup vs. serial: $S_{max} = \frac{T_1}{T_\infty}$

BOSTON UNIVERSITY

Work-Depth Model

- **Work**: Number of tasks when run serially.
  - Just count 'em
- $\mathbf{T_1}$ = (4 par + 2 seq) = 6
  - for 4 parallel tasks



4 parallel tasks as a concrete example.

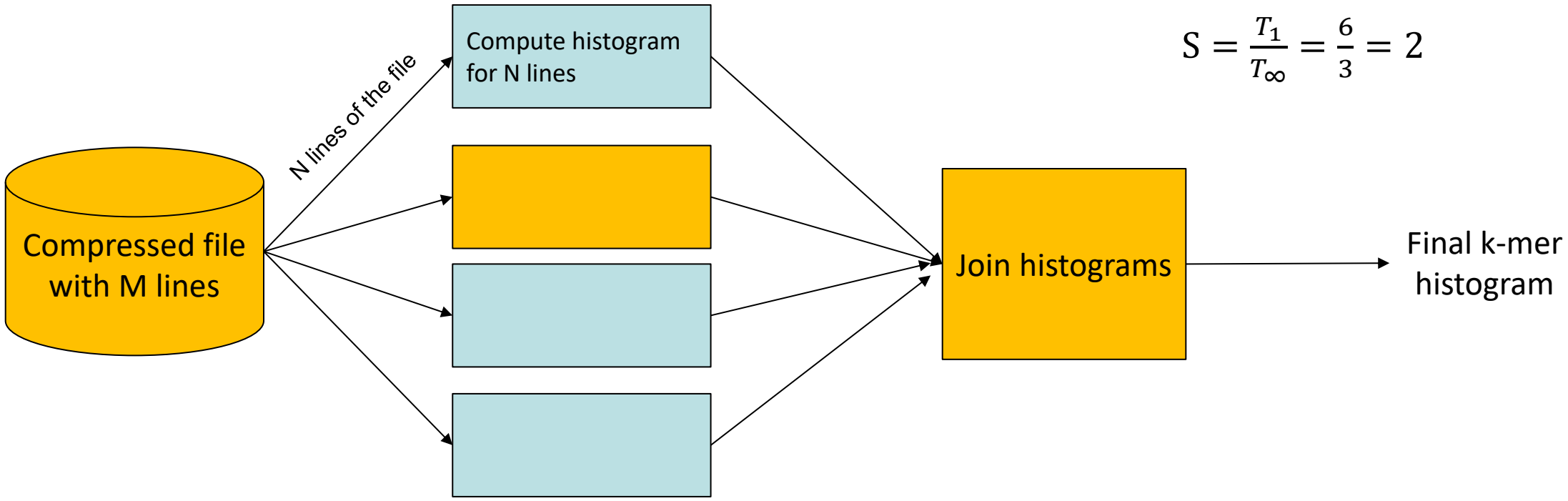
- $\boldsymbol{T_1} \approx \boldsymbol{P}$ for a large number of parallel tasks P if $P \gg 2$

BOSTON UNIVERSITY

Work-Depth Model

- Depth: Number of sequential tasks.
- $T_\infty = 3$
- Speedup with 4 processors vs. serial:

$$S = \frac{T_1}{T_\infty} = \frac{6}{3} = 2$$

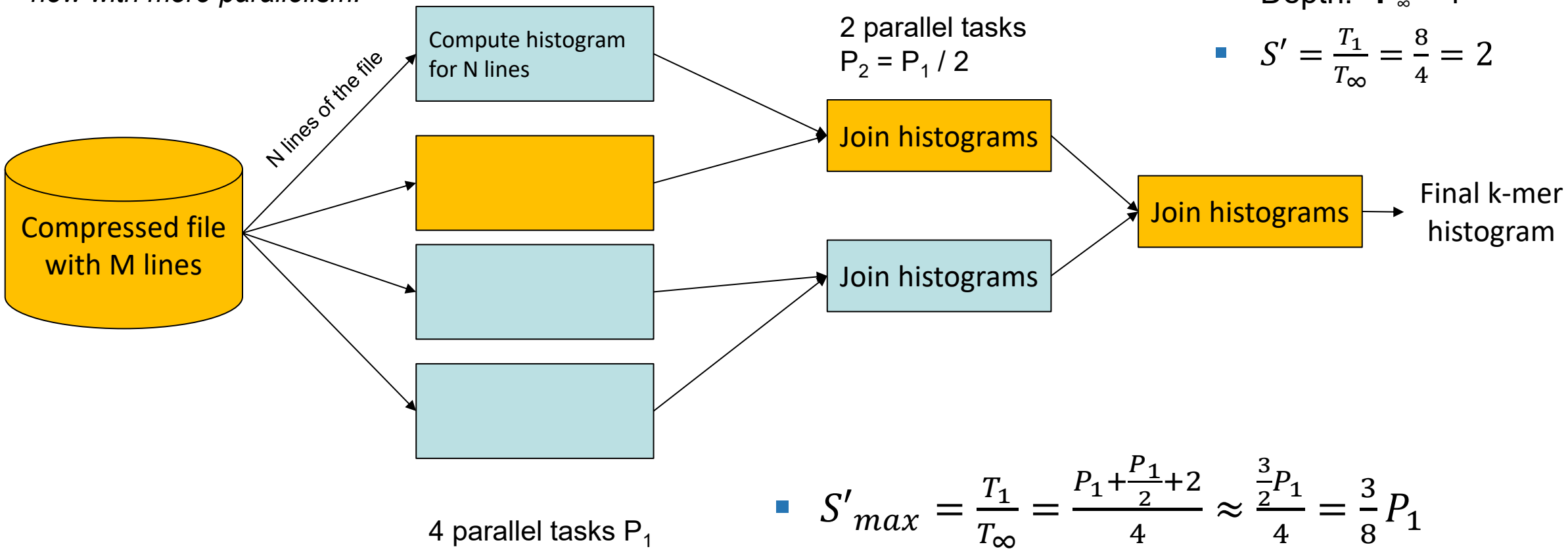


4 parallel tasks

- $S_{max} = \frac{T_1}{T_\infty} = \frac{1}{3}P$ when *P* is large.

Work-Depth Model

now with more parallelism!



With P_1=4, P_2=2

- Work: $\mathbf{T_1}$ =8
- Depth: $\mathbf{T_\infty}$ = 4
- $S' = \frac{T_1}{T_\infty} = \frac{8}{4} = 2$

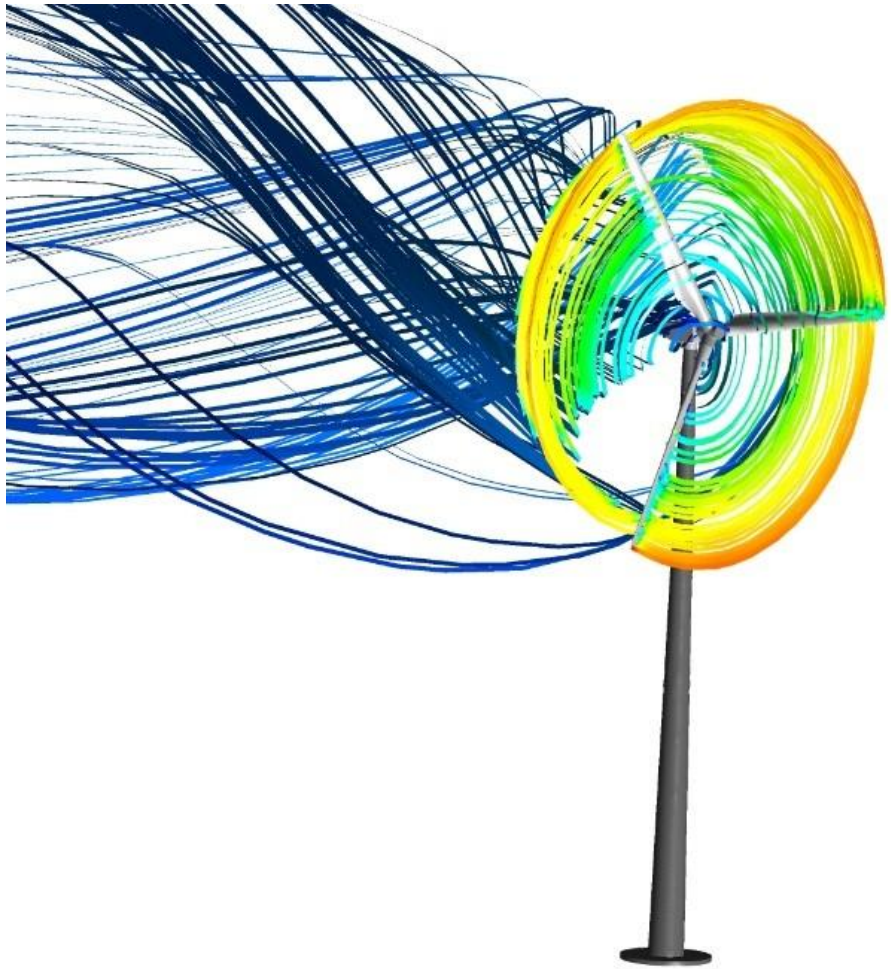
- $S'_{max} = \frac{T_1}{T_\infty} = \frac{P_1 + \frac{P_1}{2} + 2}{4} \approx \frac{\frac{3}{2}P_1}{4} = \frac{3}{8}P_1$

- Compare with the previous slide: this is a bit faster but is more complicated due to the extra parallel component.

BOSTON UNIVERSITY

Example 4. Physical Modeling

- Simulate the air flow over a wind turbine.
  - Pressure, speed, direction at thousands of points in 3D space.
  - Run the simulation for several wind speeds (say 10,15, 20, and 30 kph)
- How might this computation be parallelized?



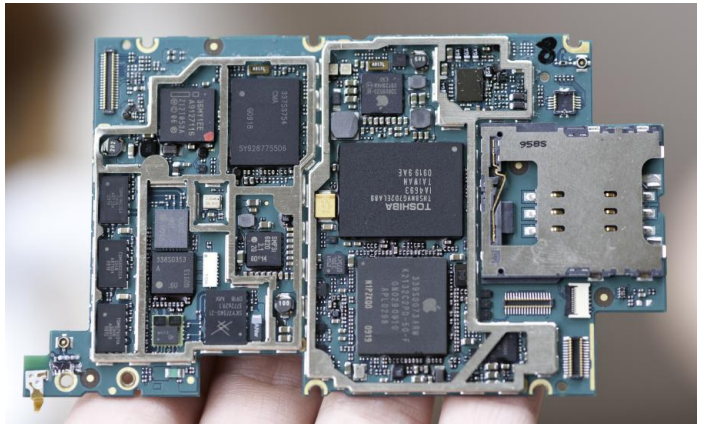
BOSTON UNIVERSITY

Outline

- Parallel Algorithm
- Hardware
- Parallel Implementations
- Processes and Threads
- Libraries
- Your code
- Pitfalls

Hardware for Parallel Computation

Lenovo ThinkSystem HPC cluster



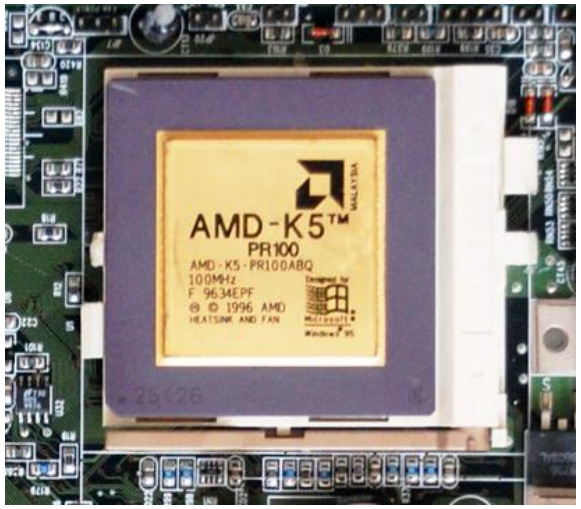
iPhone motherboard

- Parallel computing is used on systems of all sizes, from your smartphone to clusters of computers with thousands of processors in total.

BOSTON UNIVERSITY

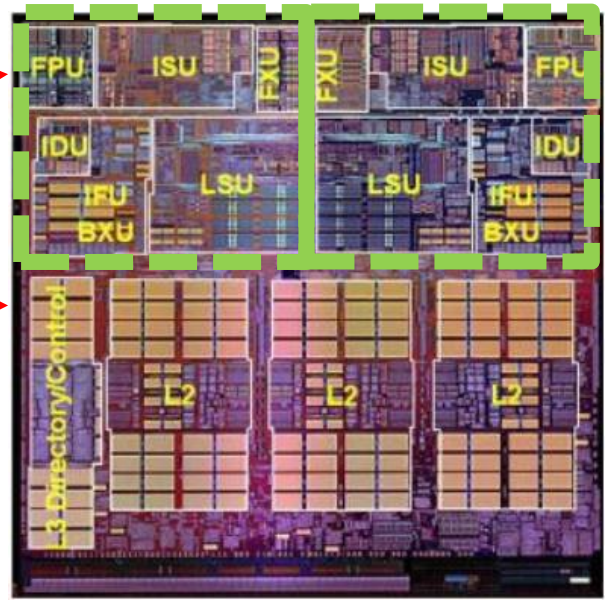
CPUs and cores

- In the beginning...a CPU plugged into a socket in the computer.
  - The term “core” wasn’t in use but we’d call this a 1-core CPU today.
  - Multiple CPU computers had multiple CPU sockets.



AMD K5 in a Socket 7 (1996)

- In 2001 IBM introduced their POWER4 CPU which embedded 2 “cores” into one physical CPU package.
  - The two cores are manufactured on the same physical semiconductor die.
  - 1 socket



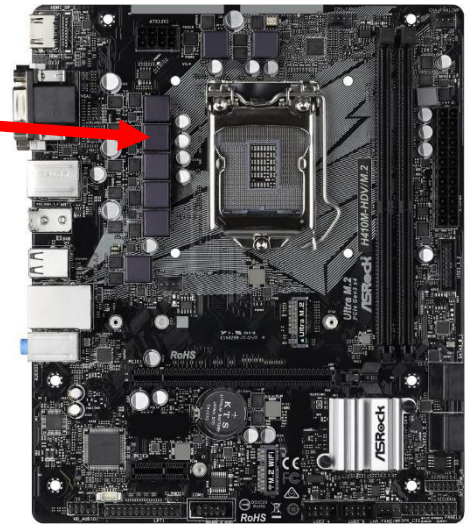
POWER4 circuit view

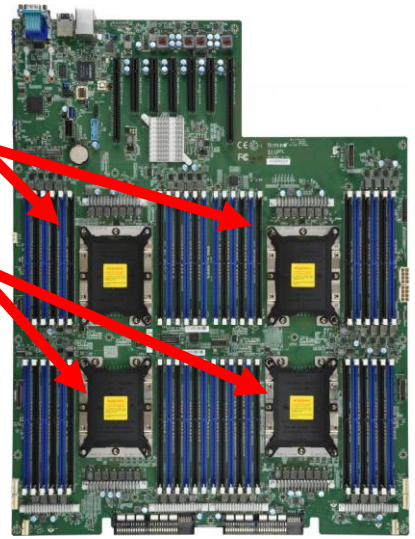
BOSTON UNIVERSITY

Modern configurations

Common desktop

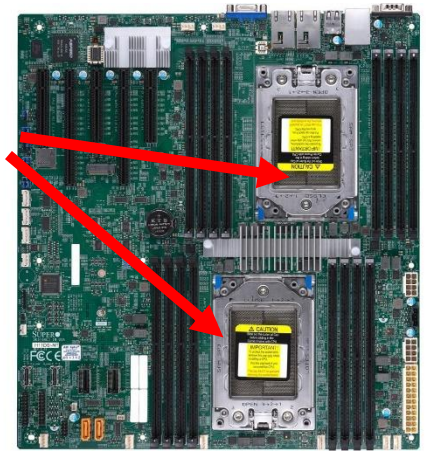
- Single Intel CPU
- 4 cores (Core-i3, ~$100)
- 4-12 cores are very common





High-end servers

- Dual AMD Epyc CPUs
- Up to 160 cores per CPU



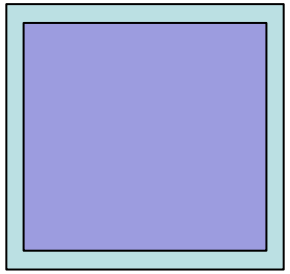
- Quad Intel Xeon CPUs
- Up to 192 cores per CPU
  - “Diamond Rapids” CPU
  - Other CPU models have as many as 288 cores with dual sockets

- For workstation and server hardware the high end has very high core counts.
- Entry-level systems still have multiple cores.

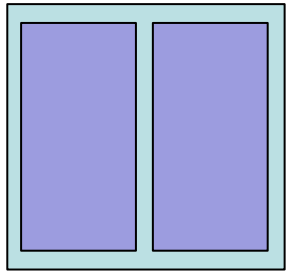
- Most SCC compute nodes are dual socket Intel-based systems.
  - 16-64 total cores per compute server

BOSTON UNIVERSITY

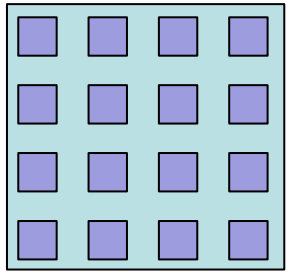
CPUs and cores



1 CPU, 1 core
1 program at a time



1 CPU, 2 cores
2 programs simultaneously



1 CPU, 16 cores
16 programs simultaneously

- "CPU" typically refers today to the physical packaging of multiple cores.
- CPU, processor, and core are sometimes used interchangeably to mean "core".

The Illusion of Parallelism

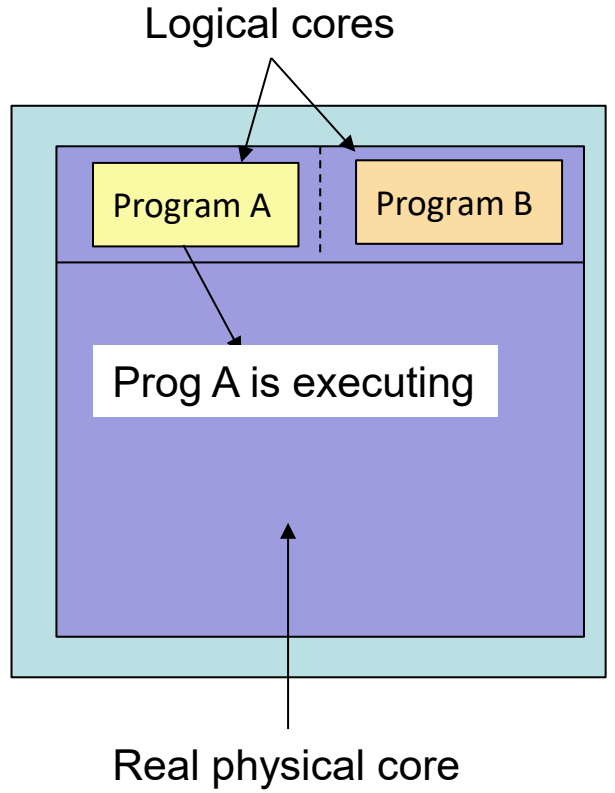
- All modern operating system will run more programs than there are available cores.

- The OS will swap the programs on and off the cores, so some execute while the others wait their turn.
  - Some programs are just “sleeping”, i.e. waiting for some OS event to occur

- If N programs are trying to compute things, then on a single core in a given timeframe each gets 1/N of the runtime.
  - 4 programs, each running “for” loops and doing calculations. On 1 core in 1 minute each will execute for ¼ of a minute (15 sec).

Logical Cores

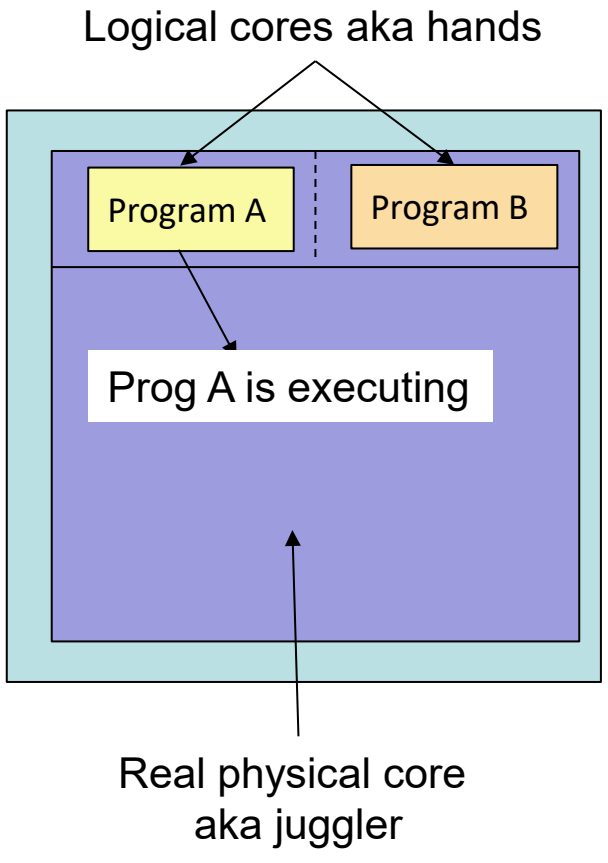
aka “hyperthreading” or “hardware threads”



- CPUs with logical cores have:
  - additional hardware that lets a program (B) have its *execution state* pre-loaded onto a core while another program (A) is executing on that core.
  - The extra hardware allows the OS to switch the physical core to run the other program (from A to B) very quickly and vice-versa.
- For many sets of programs (especially I/O bound) this makes better use of the *physical* core.
  - When program A is waiting for data, program B quickly swaps in to run.

- Intel claims overall system performance can be 30% better.

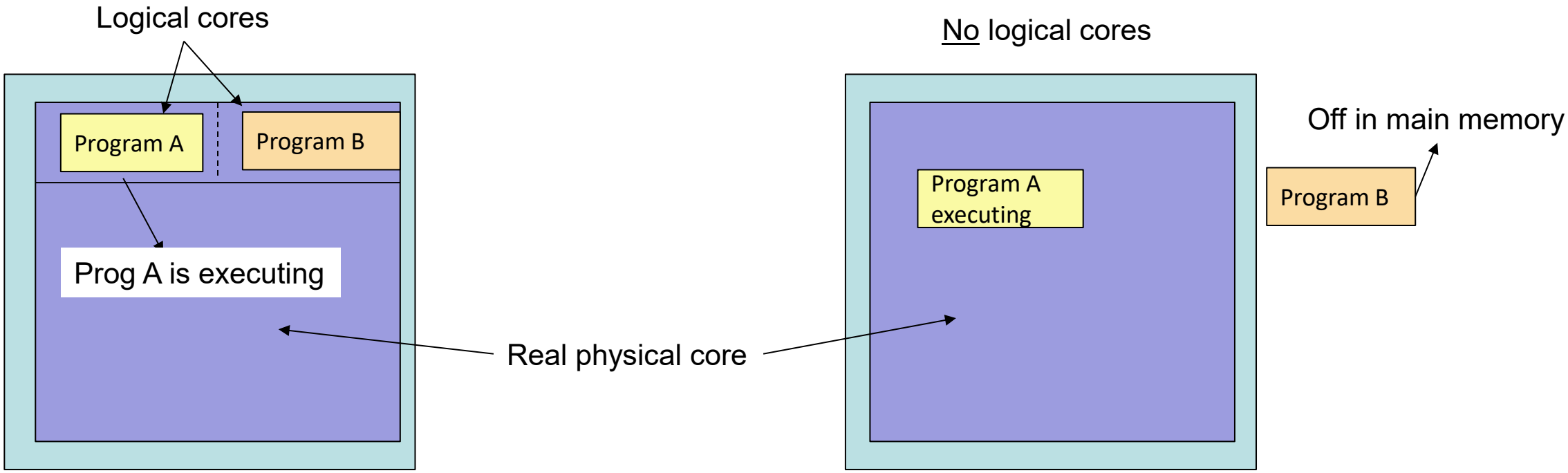
Logical Cores (by analogy)



No logical cores
1 juggler

2 logical cores
Still 1 juggler

BOSTON UNIVERSITY

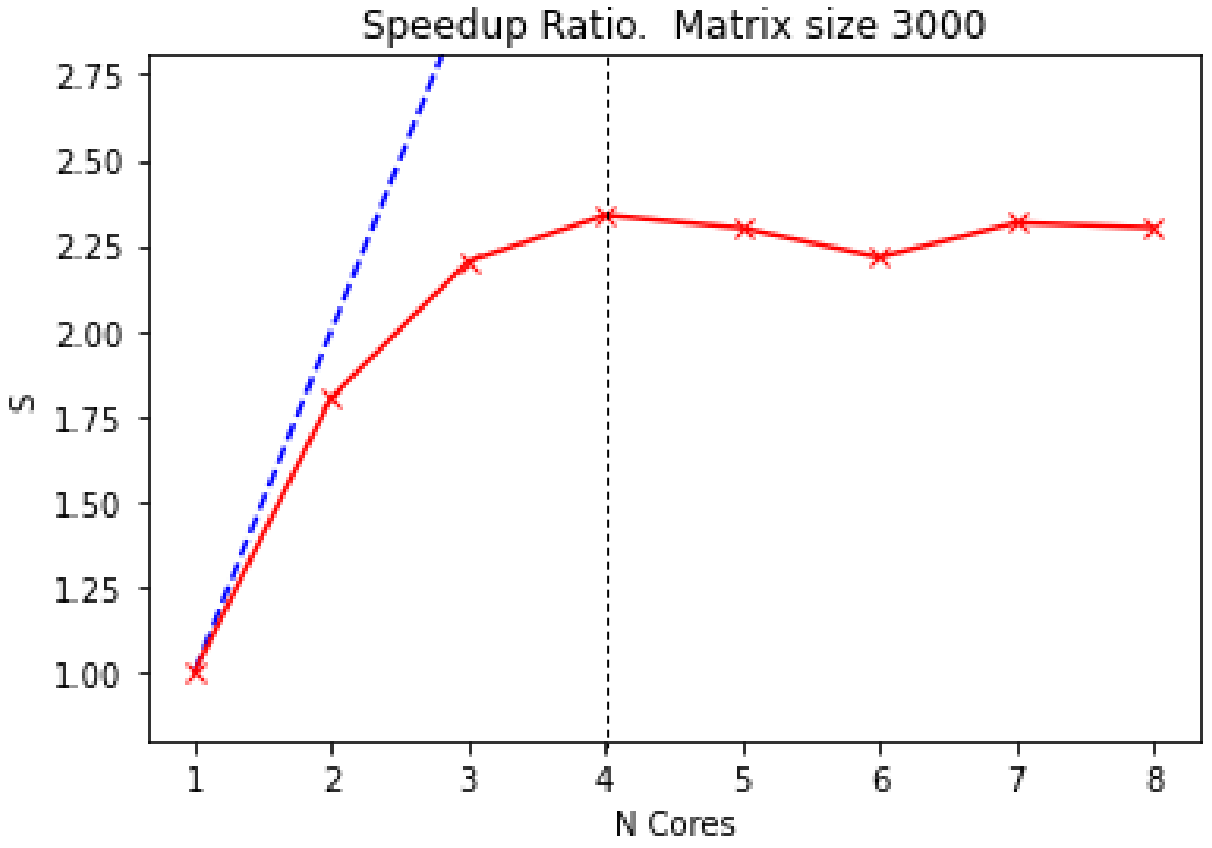


- *Without* logical cores the program switching is slower.
  - Physical core: maybe ~5-15 μs to switch the running program
  - Logical core: ~2-4x faster.

  } ballpark numbers
- Logical cores do **not** double the computational resources.

BOSTON UNIVERSITY

Logical cores on an Intel i5-9300h CPU

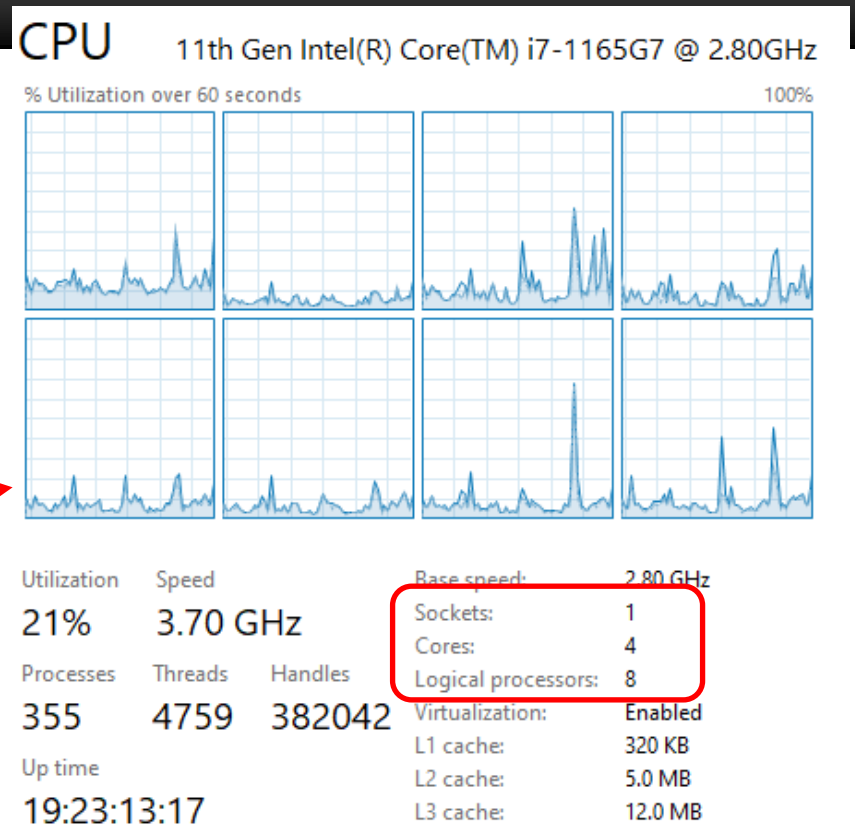


- A linear algebra matrix-matrix multiply.
  - Absolutely a CPU-bound computation!
- 4 real cores, 8 logical cores.
- Note performance increases stop for cores > 4.
- CPU-bound programs can only benefit from **real** cores.
  - You can slow down parallel code using logical cores…

BOSTON UNIVERSITY

Count Your Cores

- Operating system utilities are the easiest way.
- Windows Task Manager
  - Right-click on the taskbar, select from the list



- Linux command: *lscpu*

```
[bgregor@scc2 ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                28
On-line CPU(s) list:   0-27
Thread(s) per core:    1
Core(s) per socket:    14
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 79
Model name:            Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz
```

- Mac OSX:

```
[~] sysctl -n hw.logicalcpu
8
[~] sysctl -n hw.physicalcpu
4
```

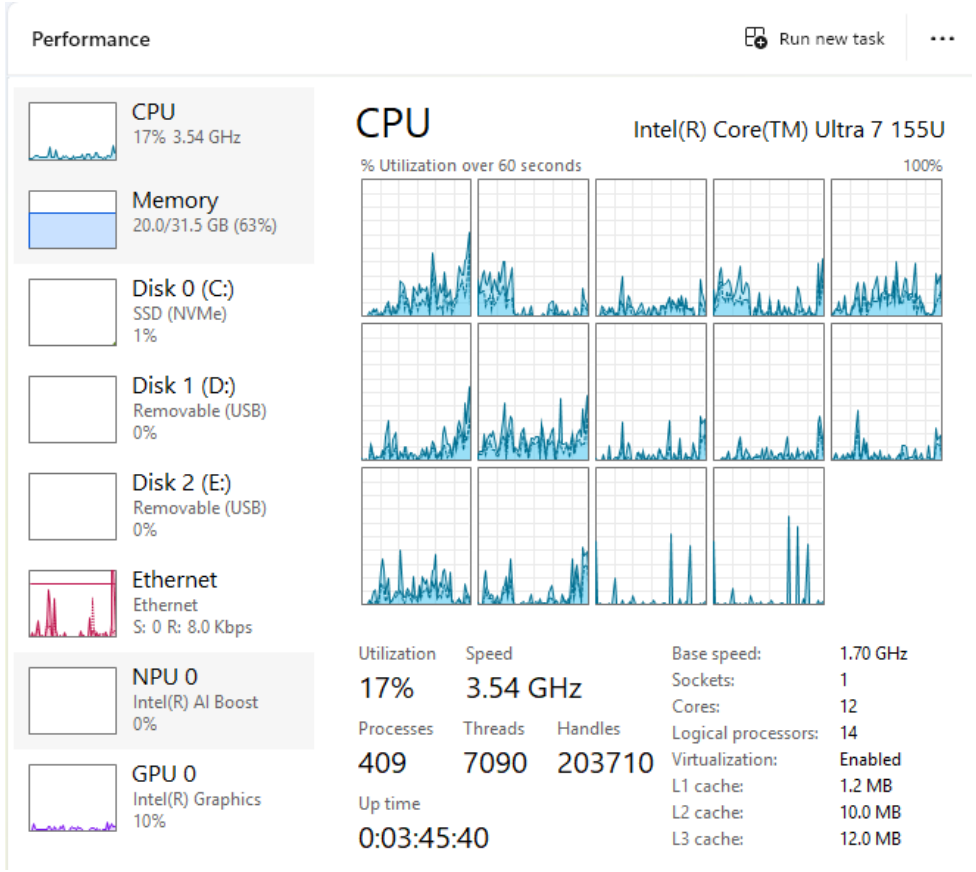
Logical Cores and You

- On your personal or lab computers, check to see if logical cores are present.
- If so, beware of using **all** of them:
  - If you're CPU-bound, only use physical cores for your code.
  - If not … test your parallel code and **time it** with physical cores only and with logical cores.
  - Ultimately – parallel speedups depend on the nature of the algorithm so you must test.

- On the SCC any compute node that supports logical cores has this feature **disabled**.
  - **All SCC core counts are real physical cores.**

And now for more complexity 

- Current CPUs for desktop/laptop use have 2+ kinds of cores
  - “Performance”: used by programs that need to run fast.
  - “Efficiency”: used for background/low priority programs.
    - Ex. The program that renews your network address from the WiFi network every few hours.
  - The operating system will generally decide what cores are used for a program.
- Intel Core Ultra 7 155U:
  - 2 “Performance” cores that have 2 logical cores
  - 8 “Efficient” cores that are the same but without logical cores
  - 2 “Low Power Efficient” cores for low-priority stuff.
    - These are much simpler cores that run slower.
- MacBook Air with the M5 CPU:
  - 4 “Performance” cores (Apple calls these “super”)
  - 6 “Efficient” cores
    - Roughly 2.5x slower, at 1/5 the power consumption



Total Cores	12
# of Performance-cores	2
# of Efficient-cores	8
# of Low Power Efficient-cores	2
Total Threads	14

Intel specs

- When running parallel computations on CPUs like these how many cores should you use?

SCC cores

- All SCC jobs set a variable, NSLOTS, that indicates the number of cores assigned to a job.
- Multiple cores in a qsub script: *-pe omp 4*
- There are options in OnDemand for multiple cores.
- “best core numbers”: 2, 4, 8, 16, 28, 32, 36*
  - There are job queues specifically for these multi-core jobs
- Example of using NSLOTS:

```
#!/bin/bash -l

# 8 core job
#$ -pe omp 8

module load python3/3.13.8

# program written to read the amount of
# parallelism from the command line
python my_parallel_prog.py --npar $NSLOTS
```

BOSTON UNIVERSITY

Outline

- Parallel Algorithm
- Hardware
- Parallel Implementations
- Processes and Threads
- Libraries
- Your code
- Pitfalls

Basics of Parallelization

- Certain patterns of program execution lend them selves to specific parallelization solutions.
- Recognizing these patterns in your code will help you choose which parallelization approach to use.
- The solutions are strategies – it's up to you to adapt them to your specific program.
- Here's a few examples. There are *lots* more than we have time for here!

Embarrassingly Parallel

- Take a list of numbers:

1 2 3 4 5 6 7 8 9 10

- And calculate its sum:

1+2+3+4+5+6+7+8+9+10

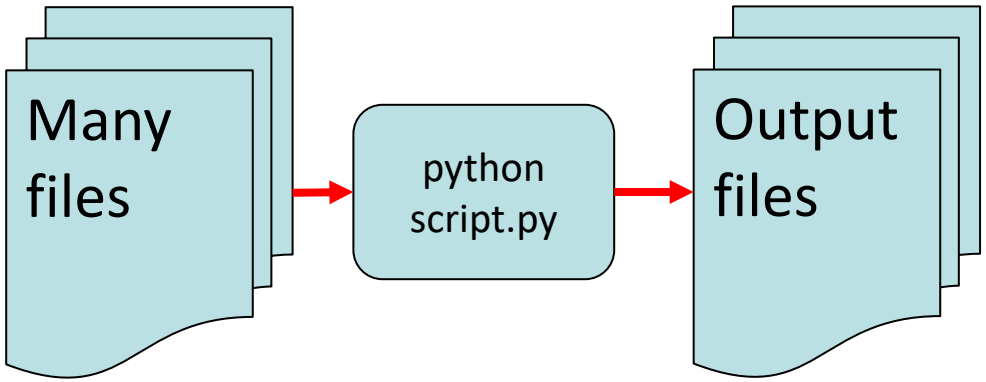
- This can easily be computed in parallel. Break into 2 chunks, sum them, and sum the chunks:
  - Or break it down into even smaller computations.

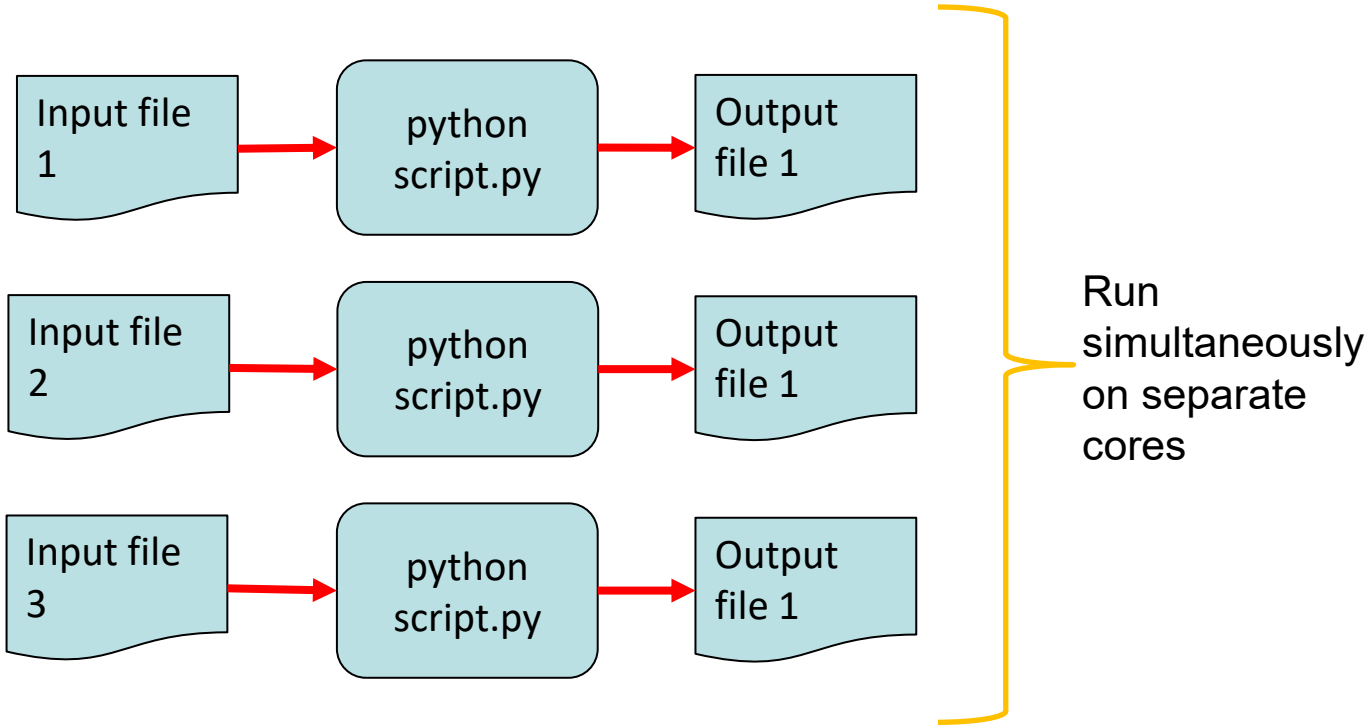
1+2+3+4+5 + 6+7+8+9+10

This is called a *fork-join* pattern.

Embarassingly Parallel

- Completely independent steps.
- Ex.: multiple runs of a simulation, processing multiple data files with the same script, calling 1 function over every element of an array.





BOSTON UNIVERSITY

This is called a *map* pattern.

Embarassingly Parallel

- Each iteration of a *for* loop might be completely independent of each other.

```
y = [1,2,3,4,5];
x = zeros(1,5);
% Each loop iteration has no dependence
% on any other loop iteration.
for i = 1:5
        x(i) = some_func(y(i));
```

Serial Matlab code

```
y = [1,2,3,4,5]
x = zeros(1,5);
% Execute in a parallel pool
parpool(5) ;
parfor i = 1:5
        x(i) = some_func(y(i));
% optional - shut down pool
poolobj = gcp('nocreate');
delete(poolobj) ;
```

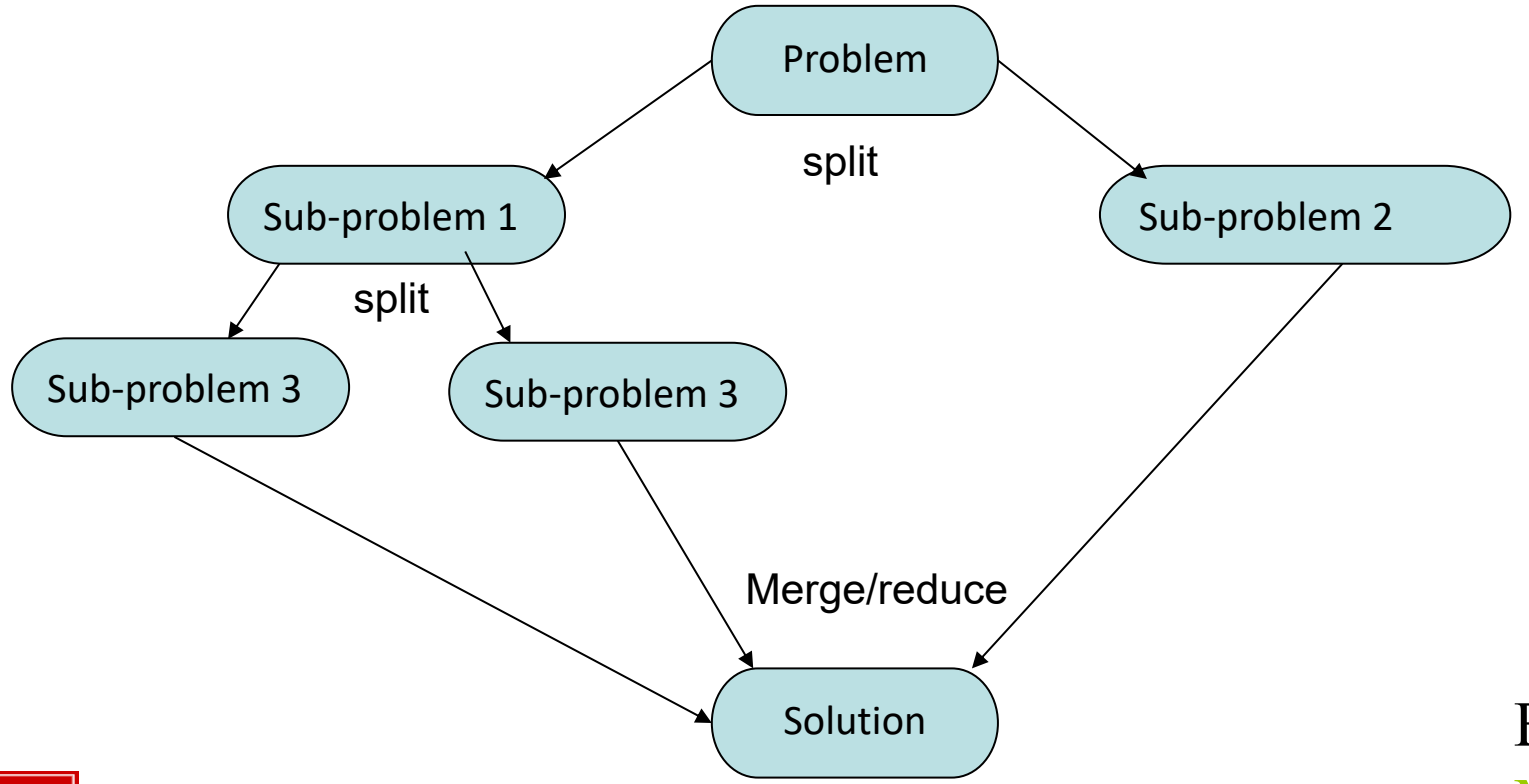
Parallel Matlab code

This is also called a *map.*

BOSTON UNIVERSITY

Divide & Conquer

- A problem can be broken into sub-problems that are solved independently.



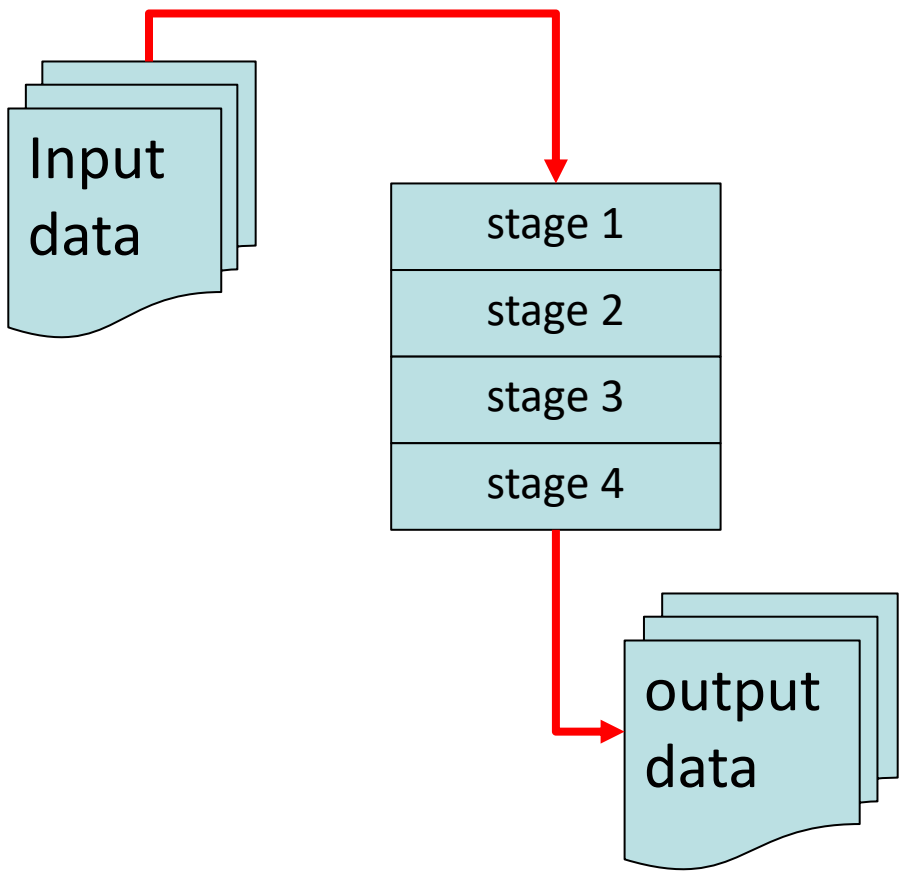
Sub-problems 1 and 2 can be executed in parallel.

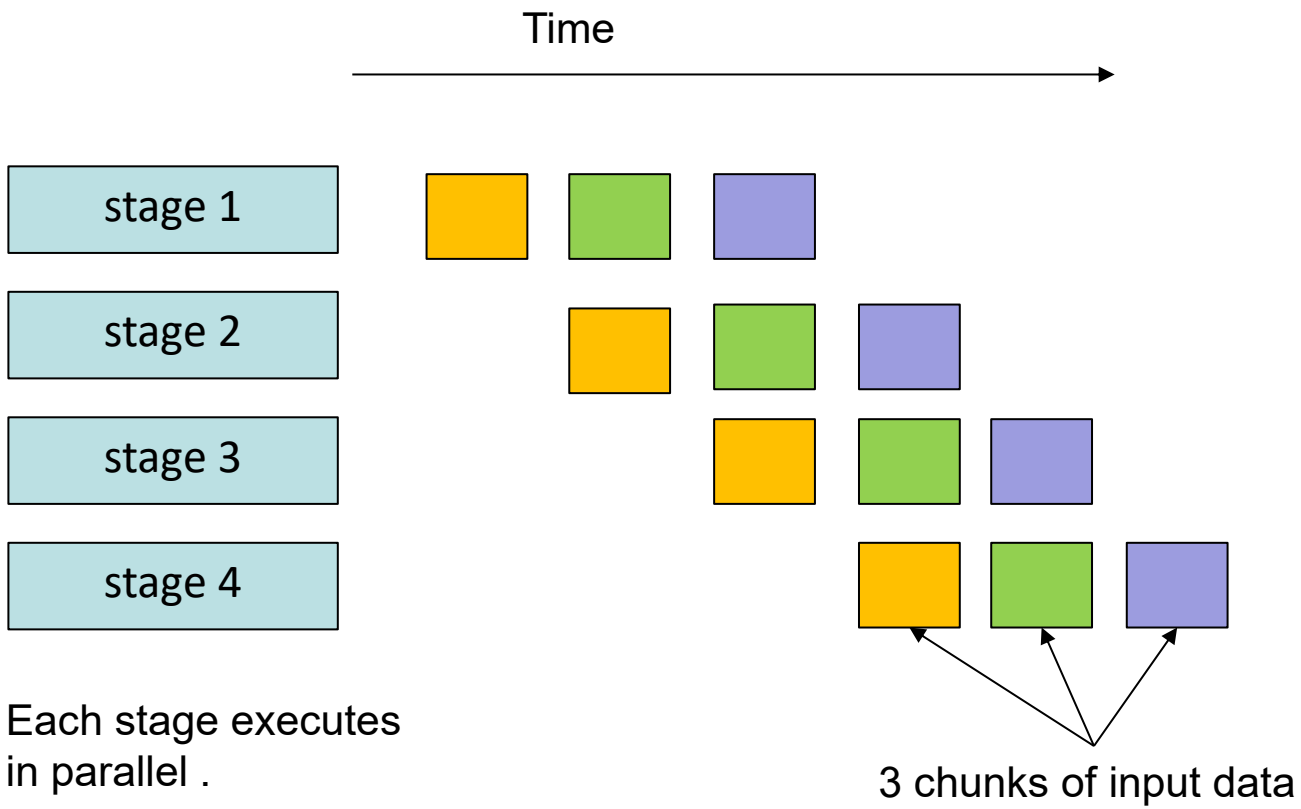
Or both 3's with 2.

Example: the famous MapReduce algorithm.

BOSTON UNIVERSITY

Pipeline

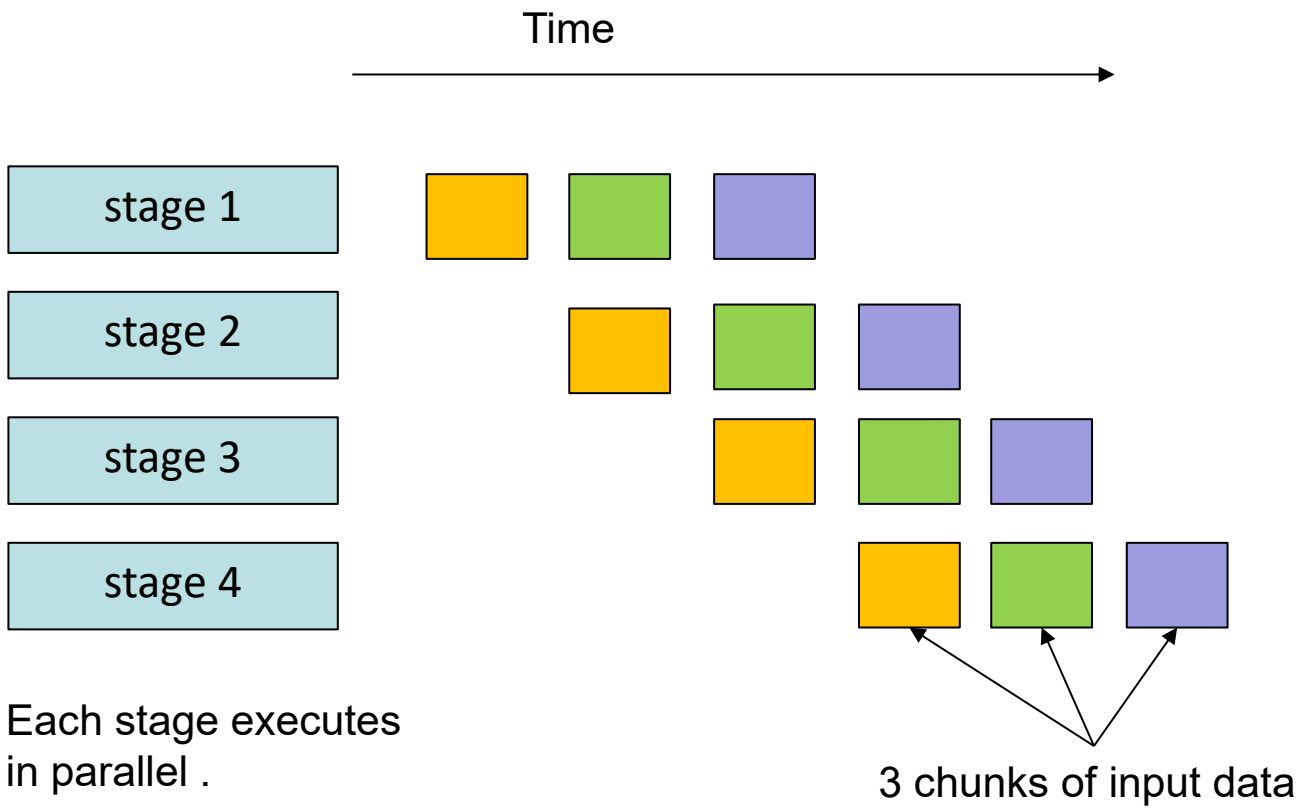




- Steps in a pipeline must run sequentially.
- These stages could be internal functions in a program.

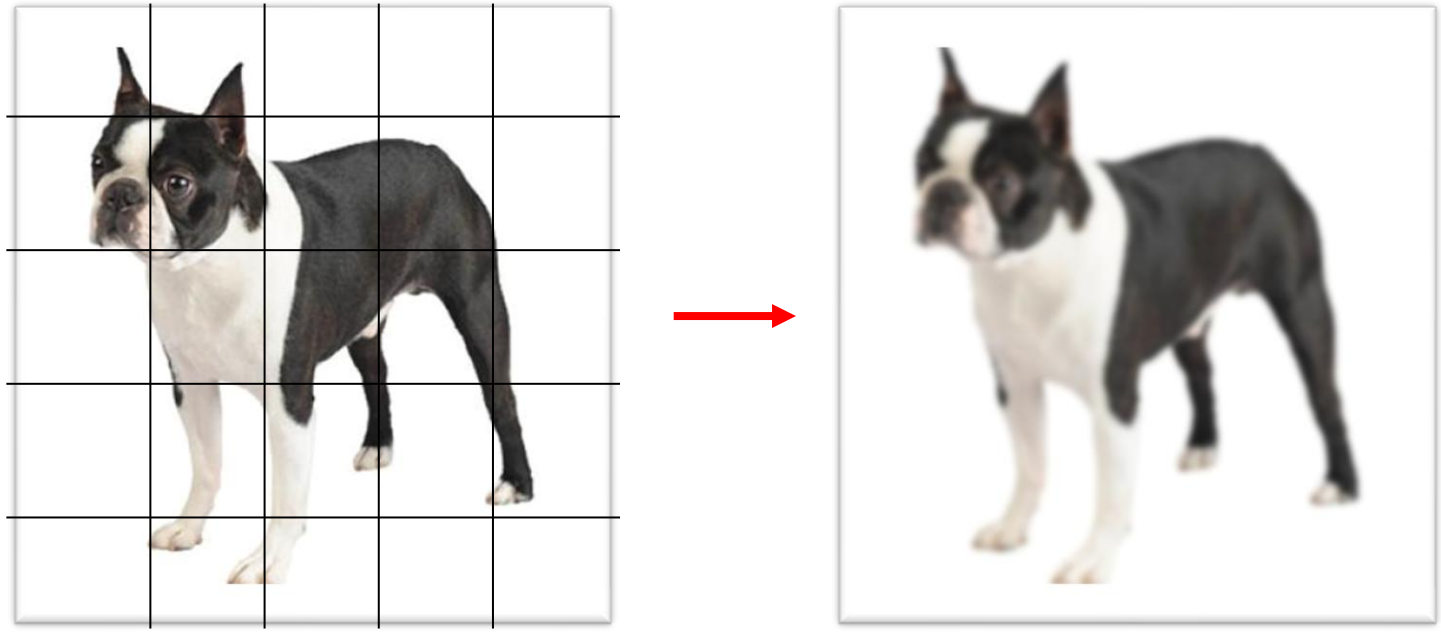
Pipeline

- Each stage could be launched in parallel.
- Some way of communication between stages is created.
  - This is very language dependent!
- As each stage completes a block of work it passes on the results and starts the next one.

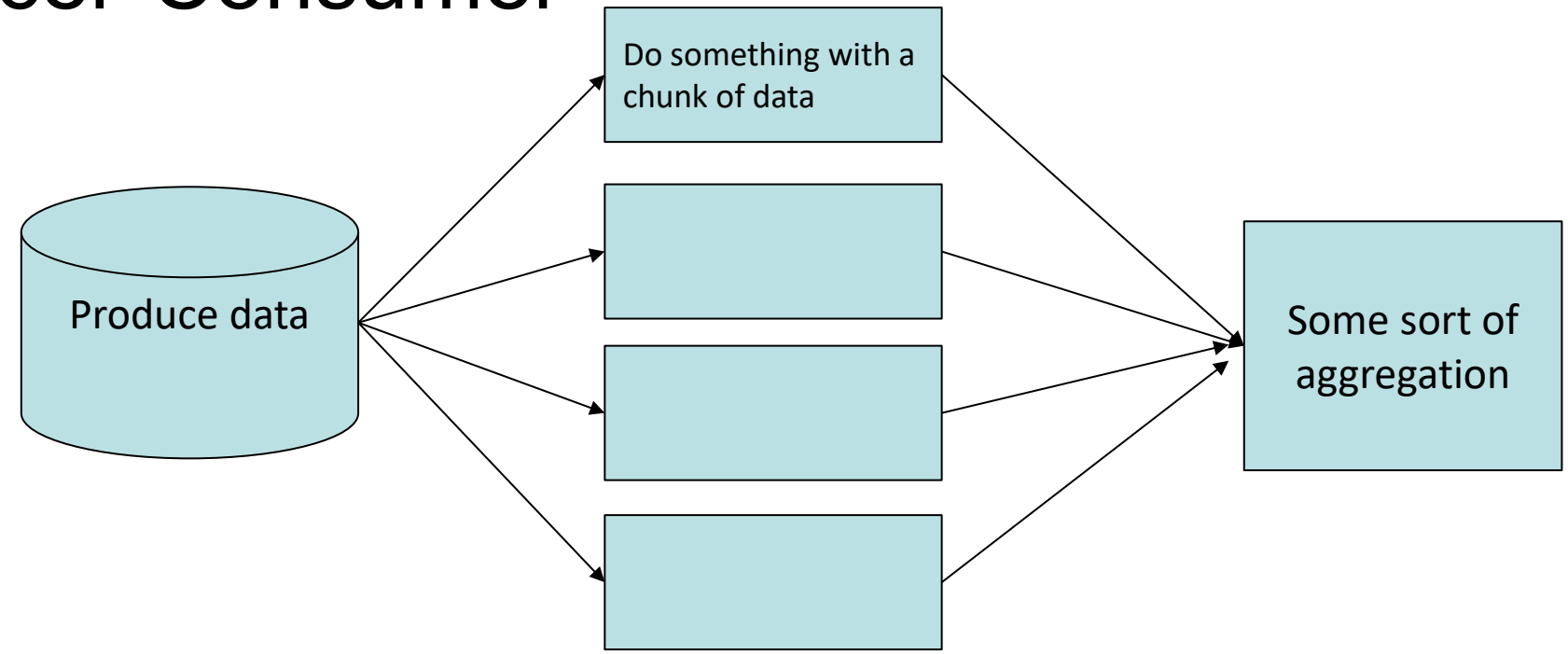


Geometric

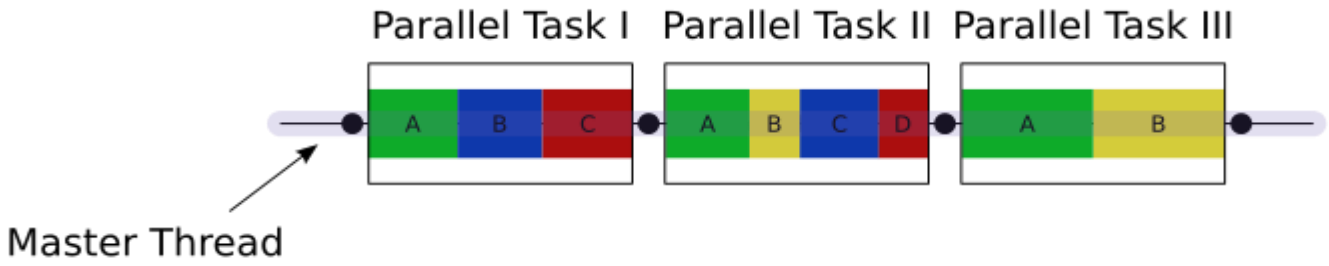
- The problem can be broken up into predictable patterns.
- Frequently used in image processing and physical simulations.

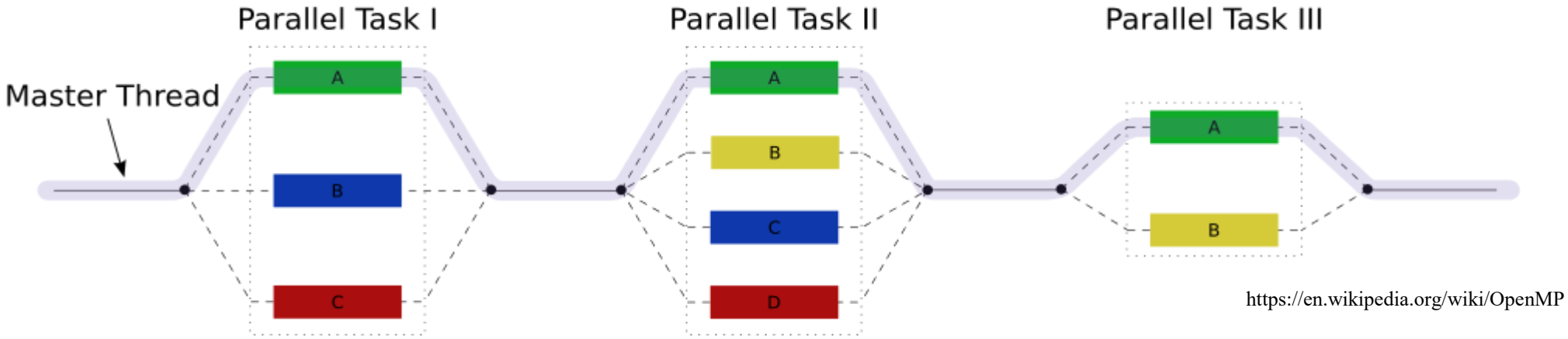


Producer-Consumer



- Something (read a file, read sensors, output of a calculation, etc.) produces data.
- A set of consumers grab a chunk of data when it's ready, does something.
  - The consumers can run in parallel.
- Typically the results are aggregated as appropriate.





https://en.wikipedia.org/wiki/OpenMP

- Different parts of a program may use *different parallel strategies* during execution.
  - Or they can be combined: a pipeline step might involve an embarrassingly parallel computation.

Outline

- Parallel Algorithm
- Hardware
- Parallel Implementations
- Processes and Threads
- Libraries
- Your code
- Pitfalls

Monitoring with the *top* tool

```
top - 10:45:13 up 45 days,  4:15, 109 users,  load average: 11.04, 5.48, 4.87
Tasks: 2753 total,   7 running, 2726 sleeping,   5 stopped,  15 zombie
%Cpu(s): 88.2 us,  2.4 sy,  0.0 ni,  8.3 id,  0.4 wa,  0.0 hi,  0.7 si,  0.0 st
KiB Mem : 26387792+total,  4700312 free, 76957904 used, 18221971+buff/cache
KiB Swap:  8388604 total,   444048 free,  7944556 used. 18075526+avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
20092 bgregor   20   0 3240428  99356  30496 R  2186  0.0   5:48.43 python
 7401 bgregor   20   0  622700 326404   3840 S   4.9  0.1 658:38.82 Xvnc
23940 bgregor   20   0 3065384 218048  72748 S   1.9  0.1  39:47.71 Web Content
16998 bgregor   20   0   59492   4948   1516 R   1.3  0.0   0:00.65 top
 7572 bgregor   20   0  359472  14244   7556 S   0.3  0.0  14:14.01 xfwm4
 8031 bgregor   20   0  785524  37588  10200 S   0.3  0.0  15:40.29 xfce4-term+
```

- On the SCC, use *top*
- To see your processes only: top -u username
- To exit *top*: press ‘q’

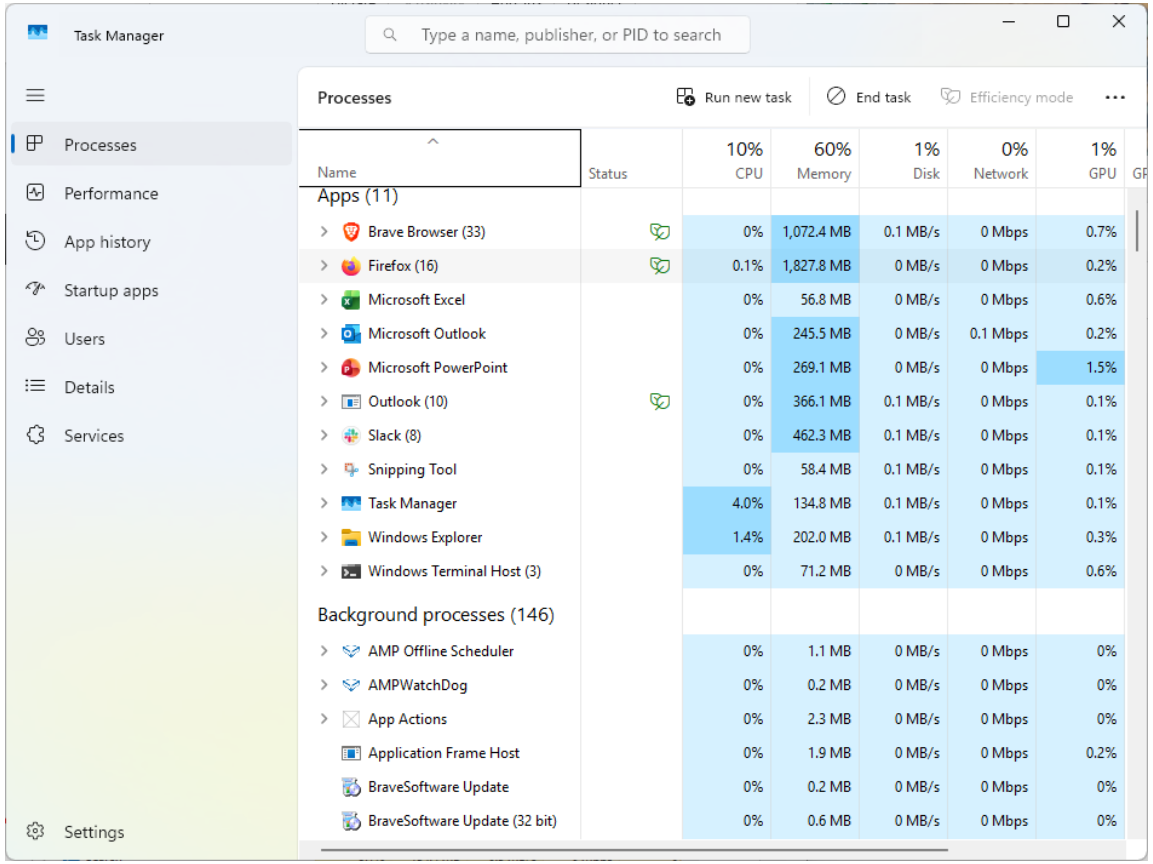
- 100% of CPU means 1 core is 100% occupied.
  - 200% means 2 cores are used, etc.
- The RES column is the amount of RAM actively in use by the process.
- VIRT is the virtual memory – essentially the maximum amount of RAM the process might request.

BOSTON UNIVERSITY

Process Monitoring - Windows

- Windows 11 Task Manager
  - Right-click on task bar
  - Same in other versions of windows, except for minor GUI differences.

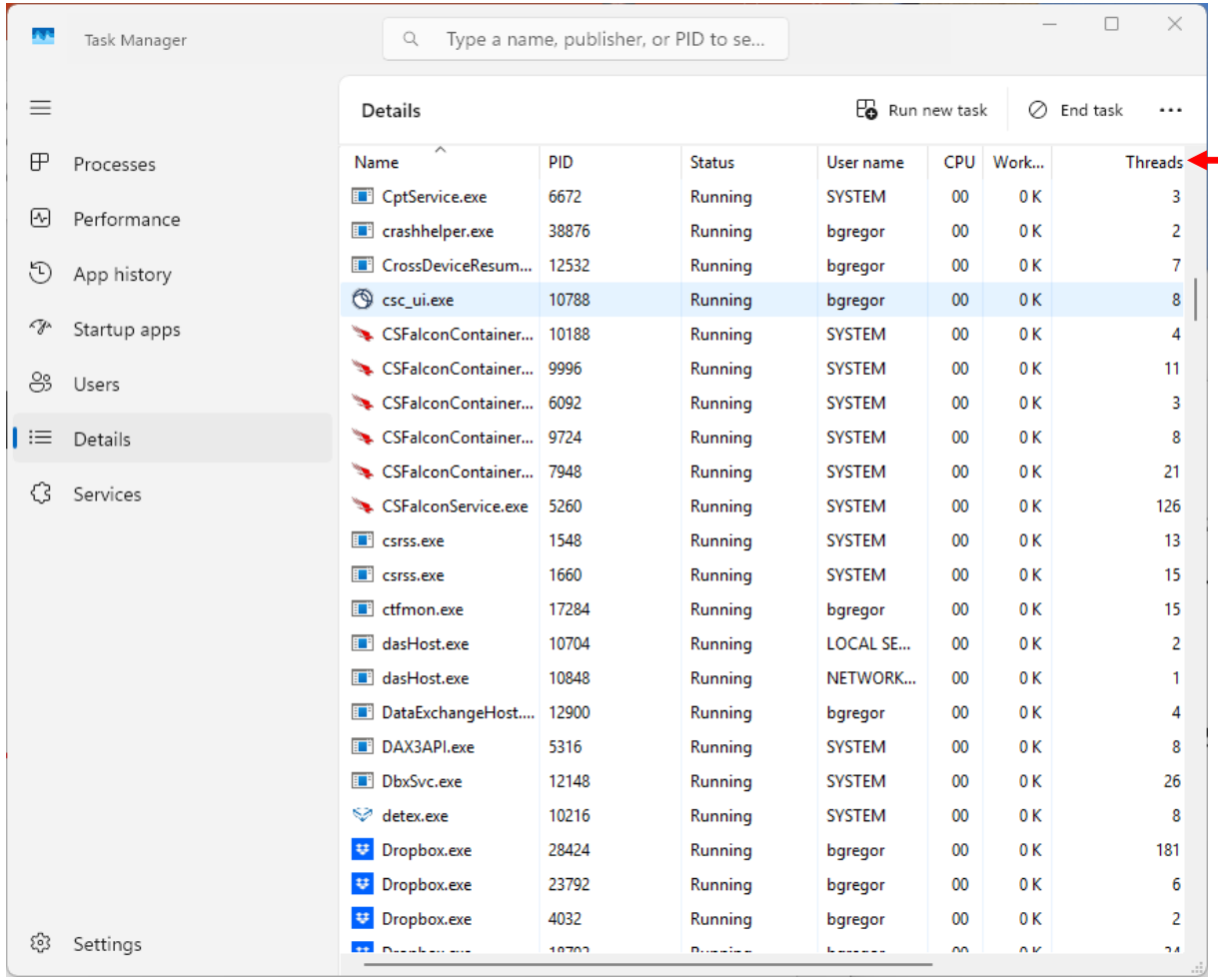
- 100% of CPU for a process means *all cores* are utilized.
  - 14 cores → (100 x 1/14)=7.1% means 1 cores is completely utilized by a process.



BOSTON UNIVERSITY

Process Monitoring - Windows

click



Right-click on the column headers → "Select Columns" → choose Threads

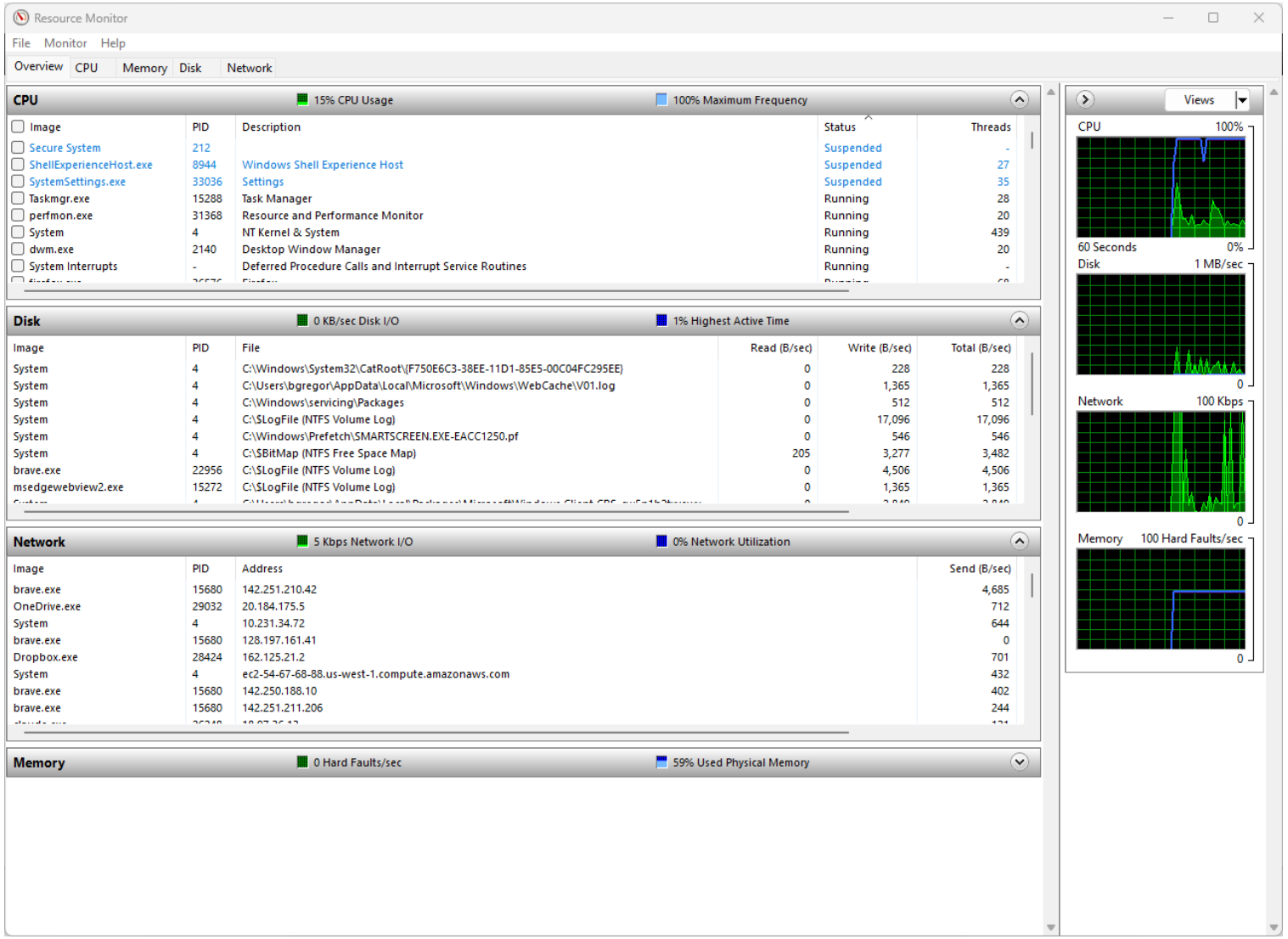
BOSTON UNIVERSITY

Process Monitoring - Windows

For even more detail – run the Resource Monitor

Press the  key, type “Resource Monitor”

This lets you see process details along with CPU, memory, disk, and network utilization on a per-process basis.

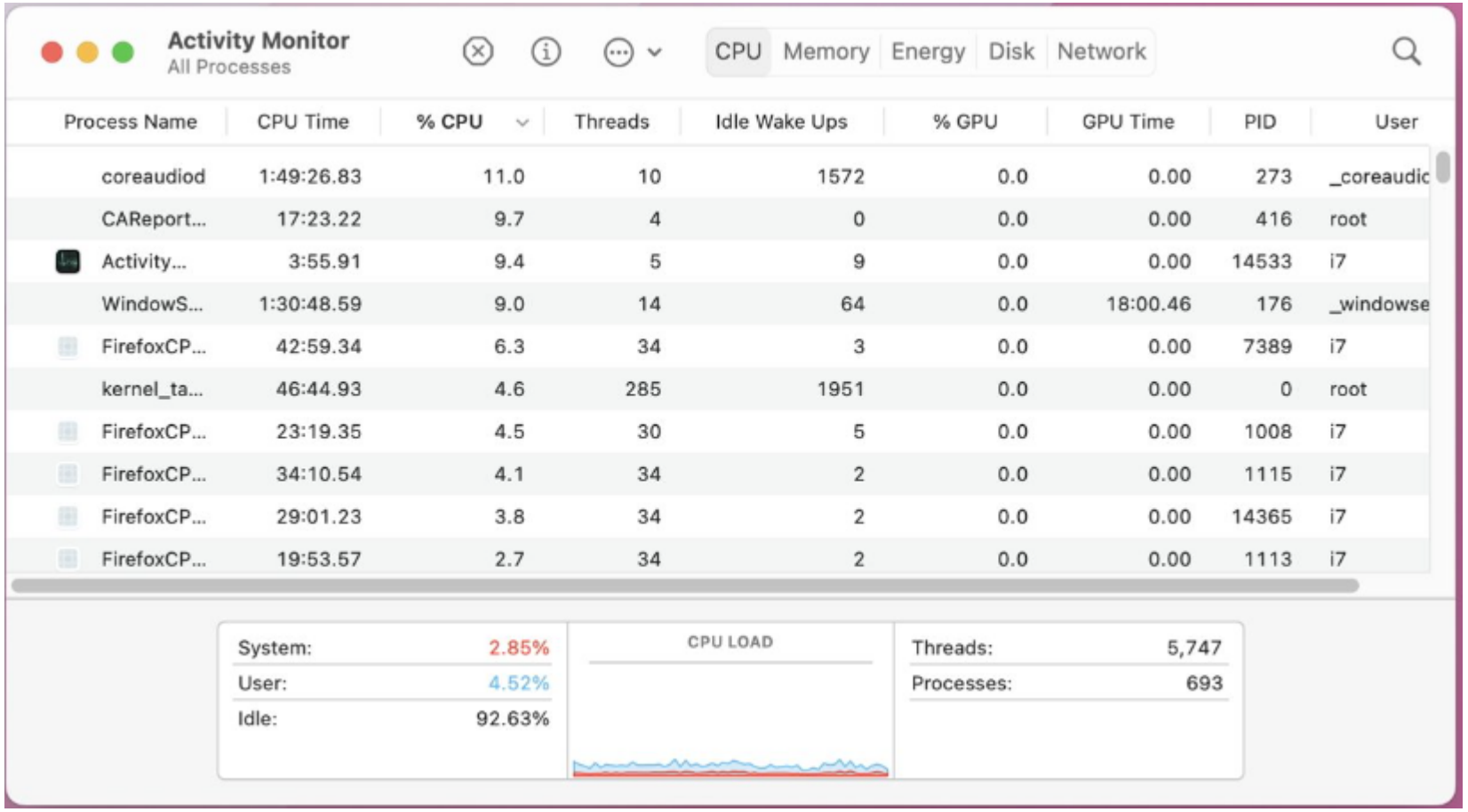


BOSTON UNIVERSITY

Process Monitoring – Mac OSX

- Use the Finder to run the *Activity Monitor*

- 100% of CPU for a process means 1 core is completely utilized.
- 200% means 2 cores, 350% means 3.5 cores (averaged), etc.



BOSTON UNIVERSITY

Process

- A program running on a computer.
- Processes can start other processes.
- Properties:
  - A private (non-shared) memory space
  - A process ID
  - Can exchange data with other processes via files, pipes, network connections, system shared memory, etc.

```
top - 17:14:03 up 29 days, 10:44, 115 users,  load average: 1.85, 1.40, 1.67
Tasks: 2855 total,   9 running, 2831 sleeping,  12 stopped,   3 zombie
%Cpu(s): 33.7 us,  1.9 sy,  0.0 ni, 64.2 id,  0.1 wa,  0.0 hi,  0.1 si,  0.0 st
KiB Mem : 26387792+total,  7611380 free, 17886752+used, 77399024 buff/cache
KiB Swap:  8388604 total,     8112 free,  8380492 used. 78756720 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
12271 bgregor   20   0  206752   8156   1248 R  99.7  0.0   0:04.51 python3
12272 bgregor   20   0  206752   8156   1248 R  99.7  0.0   0:04.53 python3
12277 bgregor   20   0  206752   8168   1248 R  99.7  0.0   0:04.51 python3
12268 bgregor   20   0  206752   8172   1268 R  99.0  0.0   0:04.50 python3
12270 bgregor   20   0  206752   8168   1260 R  98.7  0.0   0:04.46 python3
12274 bgregor   20   0  206752   8164   1248 R  98.7  0.0   0:04.49 python3
12276 bgregor   20   0  206752   8164   1248 R  98.7  0.0   0:04.49 python3
12269 bgregor   20   0  206752   8168   1260 R  98.4  0.0   0:04.48 python3
```

- Multiple Python processes running

- The operating system schedules the process so that it shares computational time with other processes.

Multiple processes →

https://en.wikipedia.org/wiki/Process_(computing)

BOSTON UNIVERSITY

Threads

- A part of a process that can be scheduled to run **independently** of the rest of the process.
- Are created, run, and destroyed by a process.
- Properties:
  - **Shares memory** with other threads and the original process.
  - Does not have a separate process ID.
  - Can exchange data with other threads or with other processes.

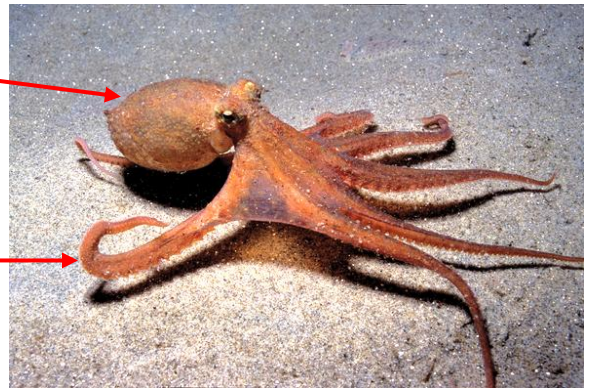
```
top - 10:45:13 up 45 days,  4:15, 109 users,  load average: 11.04, 5.48, 4.87
Tasks: 2753 total,   7 running, 2726 sleeping,   5 stopped,  15 zombie
%Cpu(s): 88.2 us,  2.4 sy,  0.0 ni,  8.3 id,  0.4 wa,  0.0 hi,  0.7 si,  0.0 st
KiB Mem : 26387792+total,  4700312 free, 76957904 used, 18221971+buff/cache
KiB Swap:  8388604 total,   444048 free,  7944556 used. 18075526+avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
20092 bgregor   20   0 3240428  99356  30496 R  2186  0.0   5:48.43 python
 7401 bgregor   20   0  622700 326404   3840 S   4.9  0.1 658:38.82 Xvnc
23940 bgregor   20   0 3065384 218048  72748 S   1.9  0.1  39:47.71 Web Content
16998 bgregor   20   0   59492   4948   1516 R   1.3  0.0   0:00.65 top
 7572 bgregor   20   0  359472  14244   7556 S   0.3  0.0  14:14.01 xfwm4
```

- Python running threads on 22 cores.
  - Note there is 1 Python process listed
  - In the *top* program 100% of CPU means 1 core is 100% busy.
  - 2186% means ~22 cores are busy.

One process

8 threads



https://en.wikipedia.org/wiki/Light-weight_process

BOSTON UNIVERSITY

Parallelize with Processes or Threads? Or both?

- You can add parallelism to your program through changing your source code or by calling libraries that implement parallel algorithms.
- Process-based parallelism:
  - Keeps memory separated.
  - Can potentially execute on multiple computers and communicate via a network.
  - Avoids issues with non-thread-safe code.
- Thread-based:
  - All the program memory is accessible by all threads.
    - Threads can use program memory for communication – much faster than processes exchanging data.
  - More complicated parallelization patterns can be implemented with less work.
    - Easy to start & stop threads.

Outline

- Parallel Examples
- Hardware
- Parallel Strategies
- Processes and threads
- Libraries
- Parallelizing your code
- Parallelization pitfalls

Types of Parallelization

- On the SCC: queue parallelization.
  - You have N files to process. Submit N jobs.
  - Or, one *job array* that launches N jobs.
  - This often requires little to no changes to your code.
- Parallel Libraries
  - Use a library that internally implements some kind of parallelization.
- Multiple Processes
  - Your program launches several copies of itself (or other programs) to solve the computational problem.
    - On one computer or many.
- Multiple Threads
  - Your program creates *threads*, which are parts of the **same** program that can execute independently of each other.

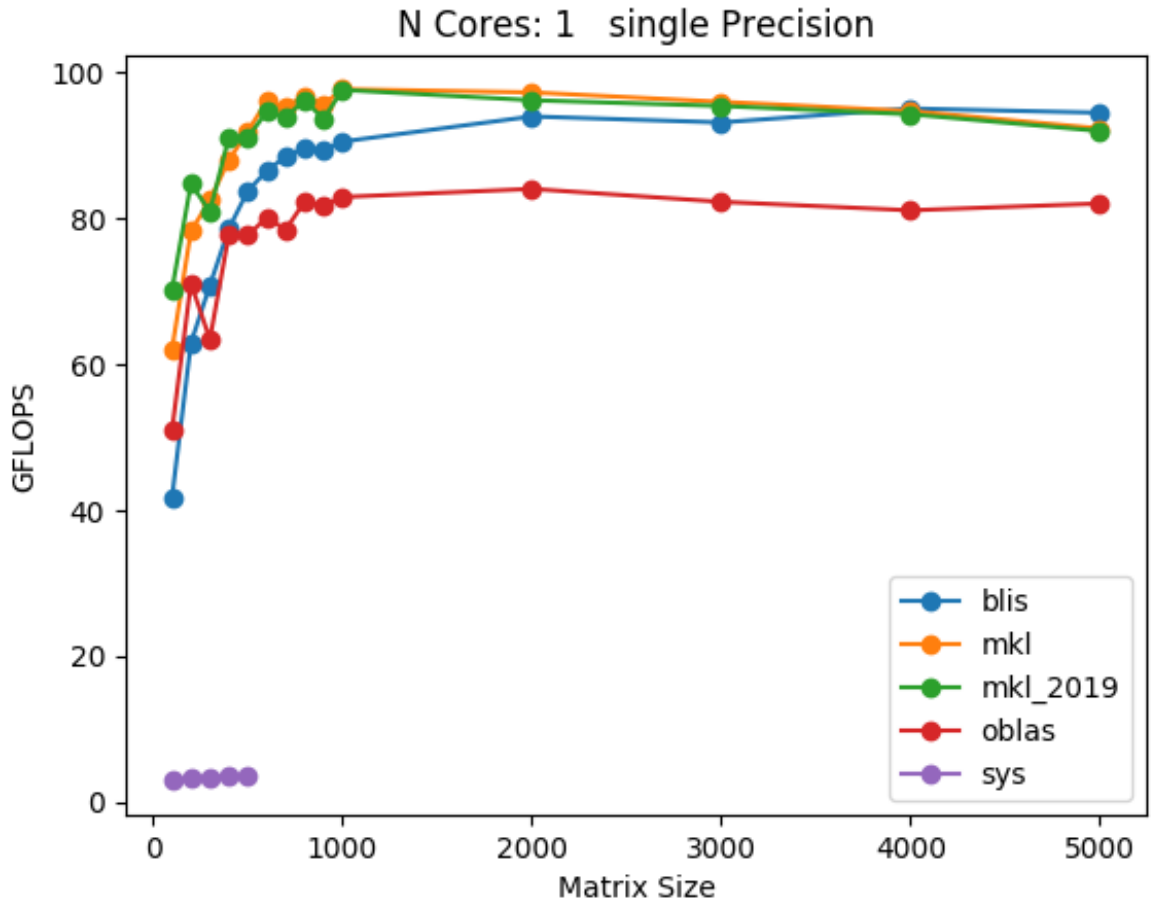
Common Parallel Libraries

Library	Parallelization	Notes
Python *multiprocessing* *joblib*	Processes	Standard language library (multi-p) and a popular one (joblib)
Python *numba*	Threads	Python function → native code compiler
Matlab *parpool* *Implicit* parallelism	Processes Threads	Standard language library. Some operations will automatically multi-thread.
R *parallel* *foreach*	Threads Processes	Standard language libaries.
BLAS (SCC: *blis* or *openblas* modules, MKL library in the *intel* module)	Threads	Linear algebra. Widely used, for example by R, Python, and Matlab.
FFTW	Threads	Fast Fourier Transforms.
OpenCV	Threads	Image processing.
Tensorflow, PyTorch	Threads (CPU) or GPU	Machine learning.
PETSc	Processes and threads	Partial differential equation solver, multi-compute node.
MPI	Processes	Low-level library for multi-node communication.
OpenMP	Threads	Low-level library (C/C++/Fortran) for multi-threading.

Example: BLAS

- The **B**asic **L**inear **A**lgebra **S**ubprograms library provides a variety of functions for linear algebra type calculations.
  - This underlies a staggering number of algorithms and computations in every area of computing.
  - Are you computing eigenvalues, doing singular value decomposition, solving least-squares, computing covariant matrices?

- High performance threaded BLAS libraries continue to be an active area of computer science research.



- SCC benchmark.

BOSTON UNIVERSITY

Enable Threading Libraries on the SCC

- The most common multi-threading library in SCC modules is OpenMP.
- The number of threads that will be used by your program can be set using the environment variable *OMP_NUM_THREADS*
- The SCC sets *OMP_NUM_THREADS=1* by default.
- For other variables see the next slide…

```
#$/bin/bash -l

# Request 8 cores for this job
# The queue will set the variable
# NSLOTS to 8
#$ -pe omp 8

# We know a priori that this multithreads
# with OpenMP
module load abc/1.0

# Allow for OpenMP threading.
export OMP_NUM_THREADS=$NSLOTS

# Using NSLOTS means we will never ask
# for more threads than cores.

# Now run the program...is it faster?
abc ...etc...
```

NEVER try to use more threads than $NSLOTS…the process reaper will kill your job.

BOSTON UNIVERSITY

Common Threading Environment Variables

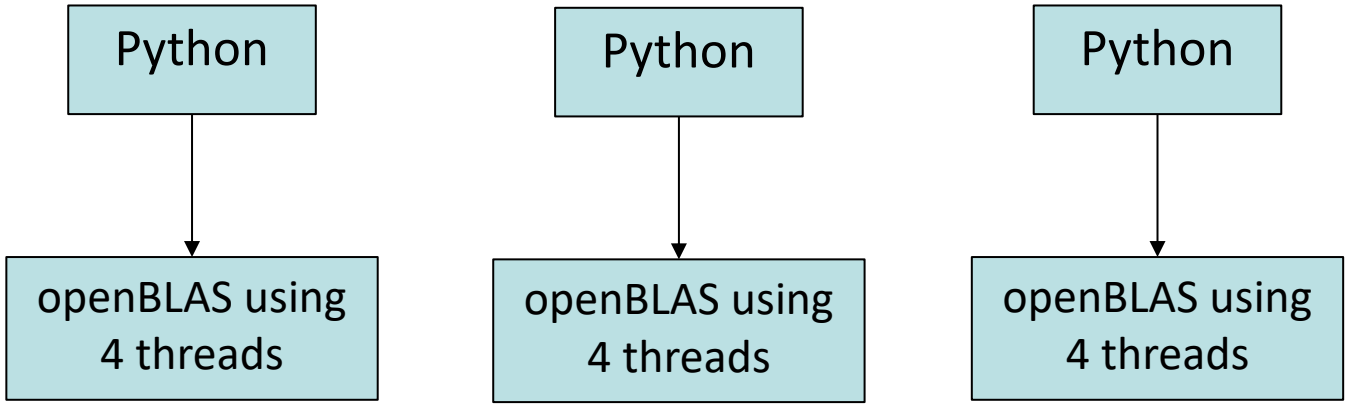
Environment Variable	Common Usage
OMP_NUM_THREADS	• OpenMP library. • SCC python3 and R modules, plus many more. • FFTW library for FFT's. • OpenMP has a lot of variables that can be set!
OPENBLAS_NUM_THREADS	• OpenBLAS library (when not built to use OpenMP). • Python+numpy when numpy was installed via pip.
MKL_NUM_THREADS	• Intel MKL library • Various SCC modules. However, MKL will also look for the OMP_* vars if MKL ones aren't found.
VECLIB_MAXIMUM_THREADS	• Mac OSX. A library called *Accelerate* replaces BLAS/LAPACK
Application/library specific	• Some software will look for their own variables. Read the documentation!

- Environment variables can be set in various ways on different operating systems. Here is a guide for Windows, Linux, and Mac OSX.
- Software can use more than one, for example it looks for its own defined environment variable that sets the number of processes to *and* also uses OpenMP threading and OMP_NUM_THREADS.
  - RTFM: "Read the Friendly Manual".

BOSTON UNIVERSITY

Feel free to email us at help@scc.bu.edu for figuring out what works for your software.

Watch Your Core Usage



OMP_NUM_THREADS=4

- Example: a Python program uses the *multiprocessing* library to launch 3 Python processes.
- Each process calls a function that eventually calls out to the openBLAS library (say using *numpy)*.
- What's the most number of cores that get used at the same time?
  - 3 processes * 4 threads per process = 12

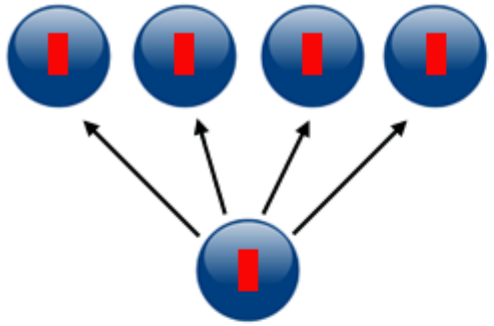
BOSTON UNIVERSITY

The Message Passing Interface (MPI)

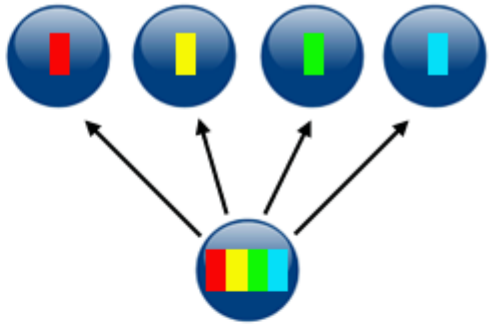
- With the right software tools processes can be run on multiple computers simultaneously and communicate with each other across a network.
- The MPI library is the most successful system for this in high performance computing.
  - On the SCC we standardized on the OpenMPI implementation: **`module avail openmpi`**
- Used on the world's largest clusters with thousands of cores over hundreds of compute nodes for single programs.

MPI

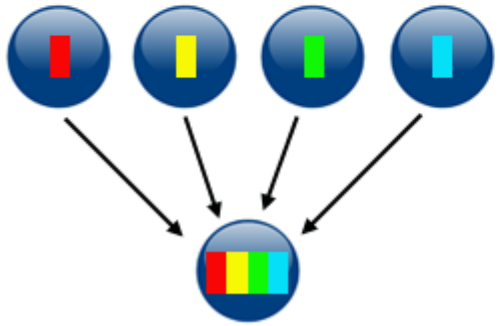
- Since MPI uses separate processes, the programmer has to decide how and when data is shared between them.
- MPI provides routines for communication, parallel file I/O, gathering and reducing data from processes, and many more.



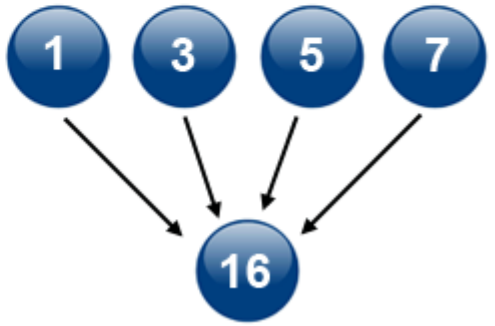
broadcast



scatter



gather



reduction

BOSTON UNIVERSITY

Using MPI in your software

- OpenMPI libraries are typically available for C, C++, Fortran, and Java.
- Wrappers libraries for MPI are readily available. These will typically work with whichever MPI implementation is available
  - OpenMPI, MVAPICH, Intel MPI, etc.

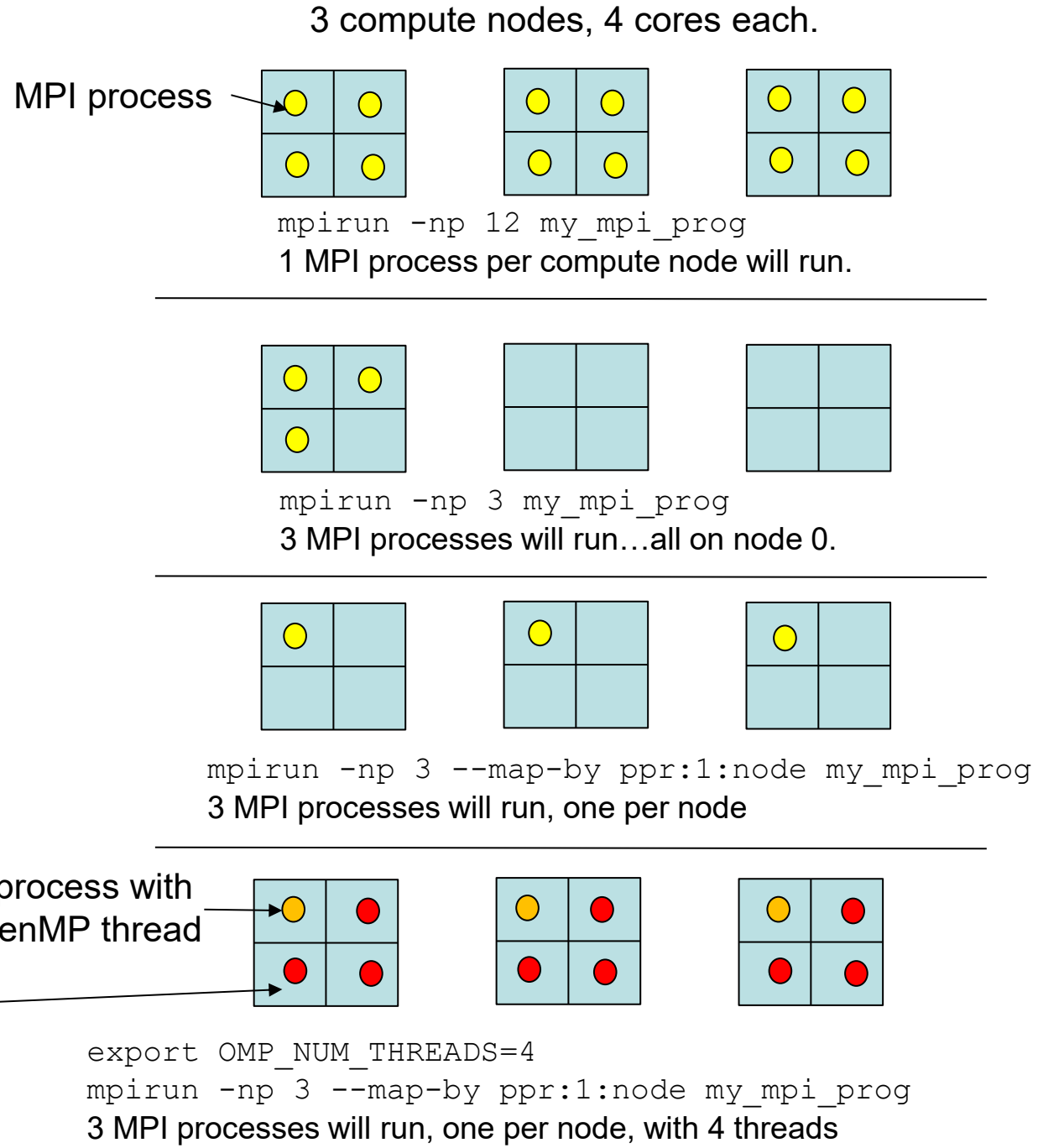
Language	Library
Python	mpi4py
R	Rmpi
Julia	MPI.jl
C#	MPI.NET

- MPI programming is an advanced programming skill. RCS is happy to help – email us!

mpirun

- MPI programs have a special program to launch them, *mpirun*
- OpenMPI's *mpirun* has many options that control how MPI processes are started and where they run.
  - Try *module help modulename* on the SCC

  for MPI-based modules
- On the SCC the configuration of compute nodes for *mpirun* is handled by the queue.

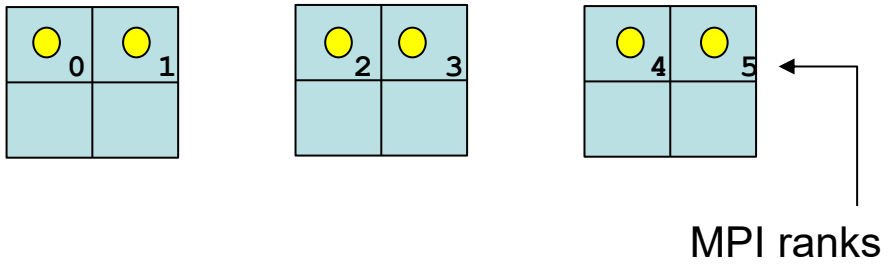


mpirun process assignment

- OpenMPI's *mpirun* can spread processes across the nodes in multiple ways
  - Recommended on the SCC:
  - --map-by ppr:N:resource Launches N processes per *resource*
  - Resource: socket, node, numa, etc.
  - You can also control how processes can be migrated between sockets, memory controllers, etc, along with any threads they launch.
  - Ask RCS for assistance.

3 compute nodes, 4 cores each.

`mpirun -np 6 **--map-by ppr:2:node** my_mpi_prog`



BOSTON UNIVERSITY

mpirun

- To experiment with various OpenMPI *mpirun* options use the `xthi` module
- This is a utility that prints out MPI process and OpenMP threads and where they were launched using *mpirun*.

```
# get yourself an MPI session
qrsh -pe mpi_16_tasks_per_node 32

# load xthi
module load openmpi/3.1.4
module load xthi/1.0

module help xthi
man mpirun

export OMP_NUM_THREADS=4
mpirun --map-by ppr:1:socket xthi
```

BOSTON UNIVERSITY

SCC MPI Nodes

- Request MPI-specific nodes on the SCC with the qsub option:
  - `-pe mpi_28_tasks_per_node N`
    - Where N is a multiple of 28
    - N=112 → 4 128-core nodes
    - NSLOTS → 112
  - `-pe mpi_64_tasks_per_node M`
    - Where M is a multiple of 64
    - (min 2 nodes)

- The only way to use multiple compute nodes for a job on the SCC is to use the MPI queues.

BOSTON UNIVERSITY

Network Type	Bandwidth (Gbit/sec)	Latency (μs)
10gig Ethernet	10	12.5
QDR Infiniband	40	1.3
FDR Infiniband	56	0.7
EDR Infiniband	100	0.5
HDR Infiniband	200	0.6

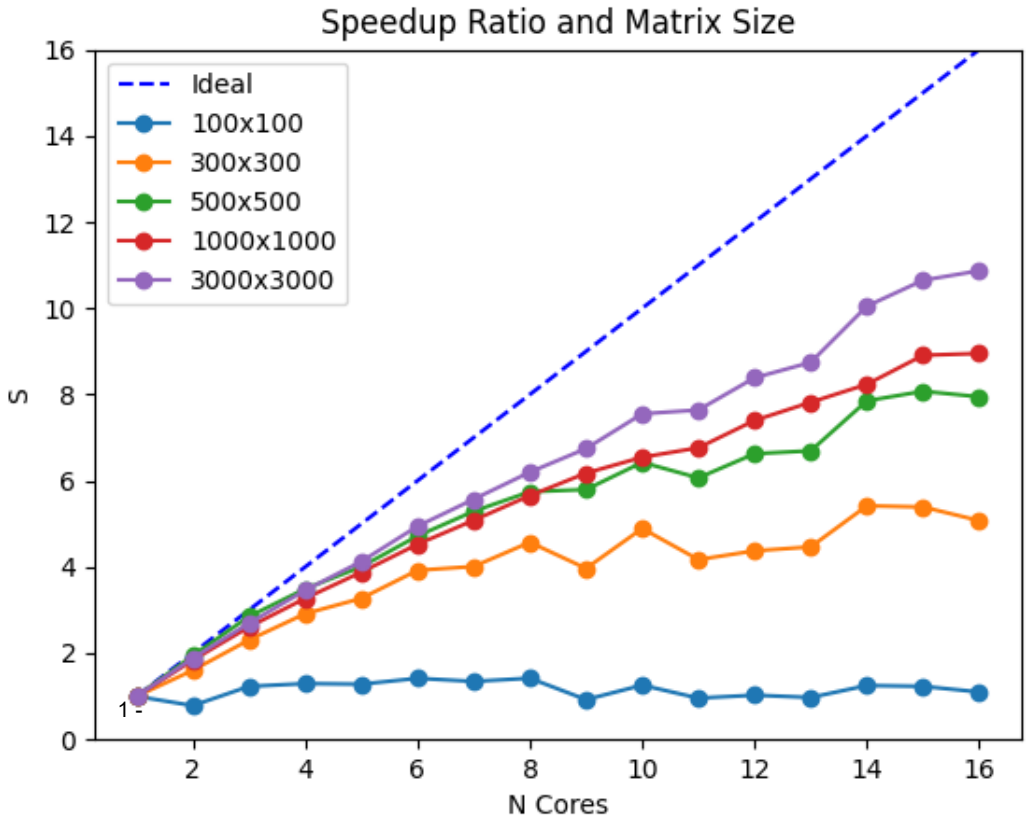
- These jobs run on dedicated compute nodes connected with an *Infiniband* network.
  - See above for SCC versions
  - The version used depends on the age of the compute node.

- Latency is how quickly a data transfer can be initiated. For MPI computations this is often the limit, not the bandwidth.

Parallel Speedup

- There are many ways to parallelize code.
- …is it worth the effort and how much will it benefit you?

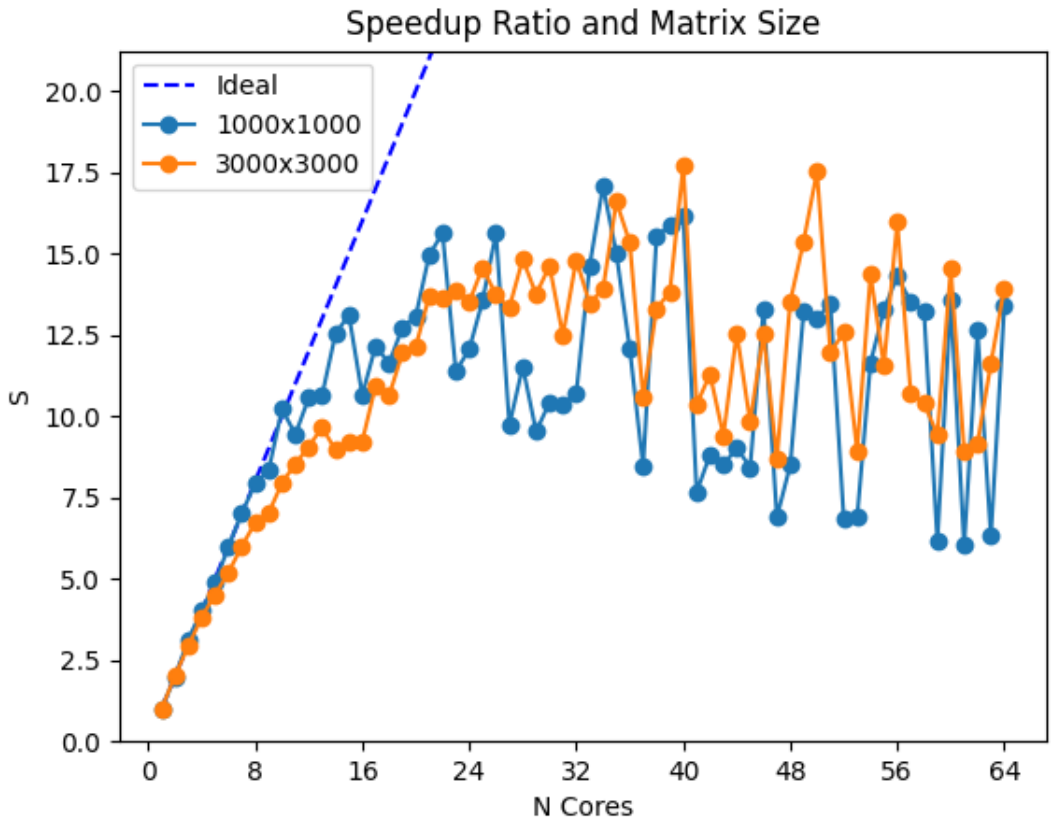
Intel Xeon CPU E5-2650 v2 @ 2.60GHz. 16 physical cores, 2 sockets (scc-pi2)



Matrix-matrix multiplication using a BLAS library

- For small matrix sizes, using any number of threads >1 is **slower**.
  - Thread coordination takes longer than the parallel speedup.
- Larger matrices have diminishing returns for higher numbers of threads.
- For any given code you'll likely find a range above which more threads doesn't help.
  - You must **test your code**!
- SCC suggestion – try 4 threads/cores.

AMD EPYC 7702 CPU @2 GHz. 64 physical cores, 1 socket



- Computation can be slower with too many threads/processes.
- The ideal thread number may change if you change the CPU manufacturer, CPU model, BLAS library, and so on.
  - …you got it..test your code!
  - The SCC is a mixed-architecture cluster, so the “ideal” settings can vary from compute node to compute node.
  - Contact RCS for help tuning your code.

BOSTON UNIVERSITY

Outline

- Parallel Examples
- Hardware
- Parallel Strategies
- Processes and threads
- Libraries
- Parallelizing your code
- Parallelization pitfalls

Parallelizing your own code

- Exactly how you parallelize your code is highly dependent on what you're doing.

- Is it your own program (you have the source code) or are you using someone else's program?
  - Always: read the documentation.

- Can it be parallelized via the SCC queue?
  - Launching many jobs *is* parallel computing.
  - This is ideal when it works – Use an array job.
  - You can have up to 1,040 cores running in jobs concurrently on the SCC.

BOSTON UNIVERSITY

Code Profiling

- For programs you've written, do you know where the program spends its time?
- Is it CPU, I/O, or memory bound?
  - And this can vary throughout a program's execution.

- Profile before you parallelize (or optimize) – we're all bad at guessing what's fast or slow in our software.
  - Using Rstudio for R code: profvis
  - Matlab: use the built-in profiler
  - Python: use the available libraries
  - C/C++/Fortran: try the Intel Advisor and/or Vtune profilers (in the `intel/2024.0` module)

Take the path of least resistance

- Parallel coding takes practice and the development of expertise.
- If your code is numerically intensive (eigenvalues, correlations, SVD, FFT's etc.) your program is likely to be using a BLAS (or FFT) library which multithreads internally.
  - Try an appropriate environment variable: `export OMP_NUM_THREADS=4`
  - If that gives you a good speedup in your code, declare victory and focus on other parts of your code or problem.
- For other people's code, check for options that enable multiple threads or processes.

Use the source

- If you have the source code (or it's your own program) you have much more control
- Examine time-intensive loops or functions.
- Look for language options for implicit/automatic multithreading:
  - Ex. Matlab: `maxNumCompThreads(N)`
- Incorporate parallel libraries
  - Can you parallelize using language-level libraries?

- Do parallel algorithms already exist?
  - For a particular function/calculation, do a literature search – lots of solutions get published.
  - There's a lot of parallel design patterns, if one maps to your problem you can use it as a guide.

Modify your code

- Make use of parallel capabilities built into the language when you can:
  - Matlab: `parfor`
  - R: `parallel:mclappy, foreach, doParallel`
  - Python: `multiprocessing.Pool, joblib`
  - C++ (C++17 standard and up) parallel standard template library (STL) algorithms

- More extensive or elaborate parallelization might require using an additional programming language with libraries like OpenMP or Intel Thread Building Blocks.
  - Lower level languages (C++, Fortran, etc) allow for more control over how algorithms are executed.
  - R → Rcpp (C++)
  - Python → use Numba, or use Numba to call out to C, C++, or use Fortran (via f2py)
  - Matlab → C or C++ using the mex tool

Outline

- Parallel Examples
- Hardware
- Parallel Strategies
- Processes and threads
- Libraries
- Parallelizing your code
- Parallelization pitfalls

Parallelization Difficulties

- Some code cannot be parallelized – it must be computed in order.
- Some loops or function calls can have dependencies on other loop iterations that make it impossible, difficult, or inefficient to parallelize.
- Choose your battles wisely
- Use profiling to identify code that is worth improving.

Parallelization Difficulties

- Random number generation is not straightforward. RNG algorithms cannot be called from multiple threads.
- **Do not improvise this**, read documentation!
- Computing RNG's in parallel requires different random seeds for each worker*.
  - Suggestion: seed your RNG in the main process. When spawning workers, provide each a different random number to use as a seed for a private RNG for that worker.

Notes for Python, Matlab, and R.

BOSTON UNIVERSITY

* worker: process or thread

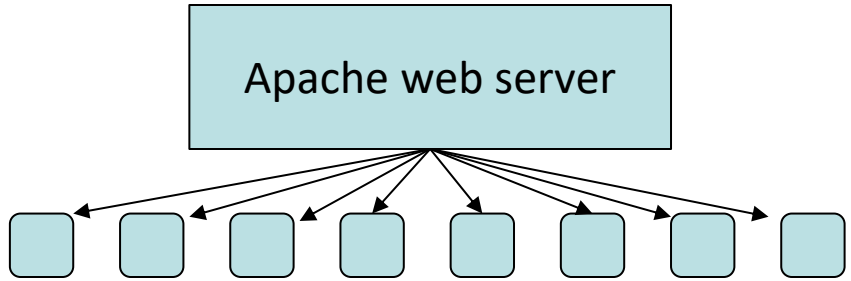
Parallelization Difficulties

- Be careful about the amount of I/O your workers are performing.
- Disks, networks, etc. have bandwidth limits.
- Excess workers can overload resources, turning the problem from CPU-bound to I/O bound.

- Multiple process parallelization can consume large amounts of memory.

How Many Workers*?

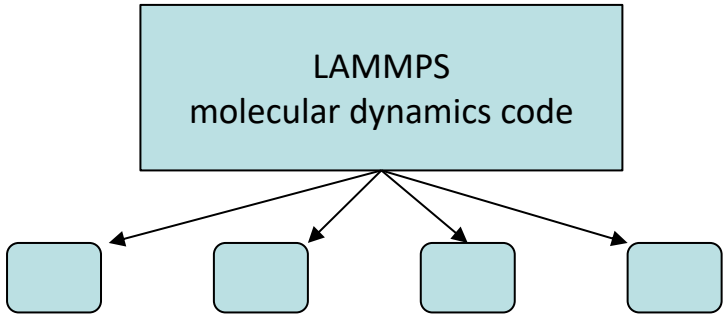
- I/O-bound programs may run hundreds or thousands of workers
  - These spend a lot of time **waiting** for data from the network, the disk, the user, etc.



....

Hundreds of copies of itself handle incoming web traffic

- CPU-bound programs should run one worker per physical core.



4 cores – 4 workers

- Memory-bound programs often use fewer workers than cores.

BOSTON UNIVERSITY

What happens with too many CPU-bound workers?

- More than 1 results in workers competing for access to the cores and memory bandwidth.
- Performance will suffer **significantly** with excess workers.
- Watch for mixing multiple processes and multithreading (like MPI with OpenMP): each process can end up launching many threads, overloading the cores.