Introduction to Parallel Programming Concepts

Summer 2025

Research Computing Services
IS & T

BOSTON UNIVERSITY

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.

- Usually, 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

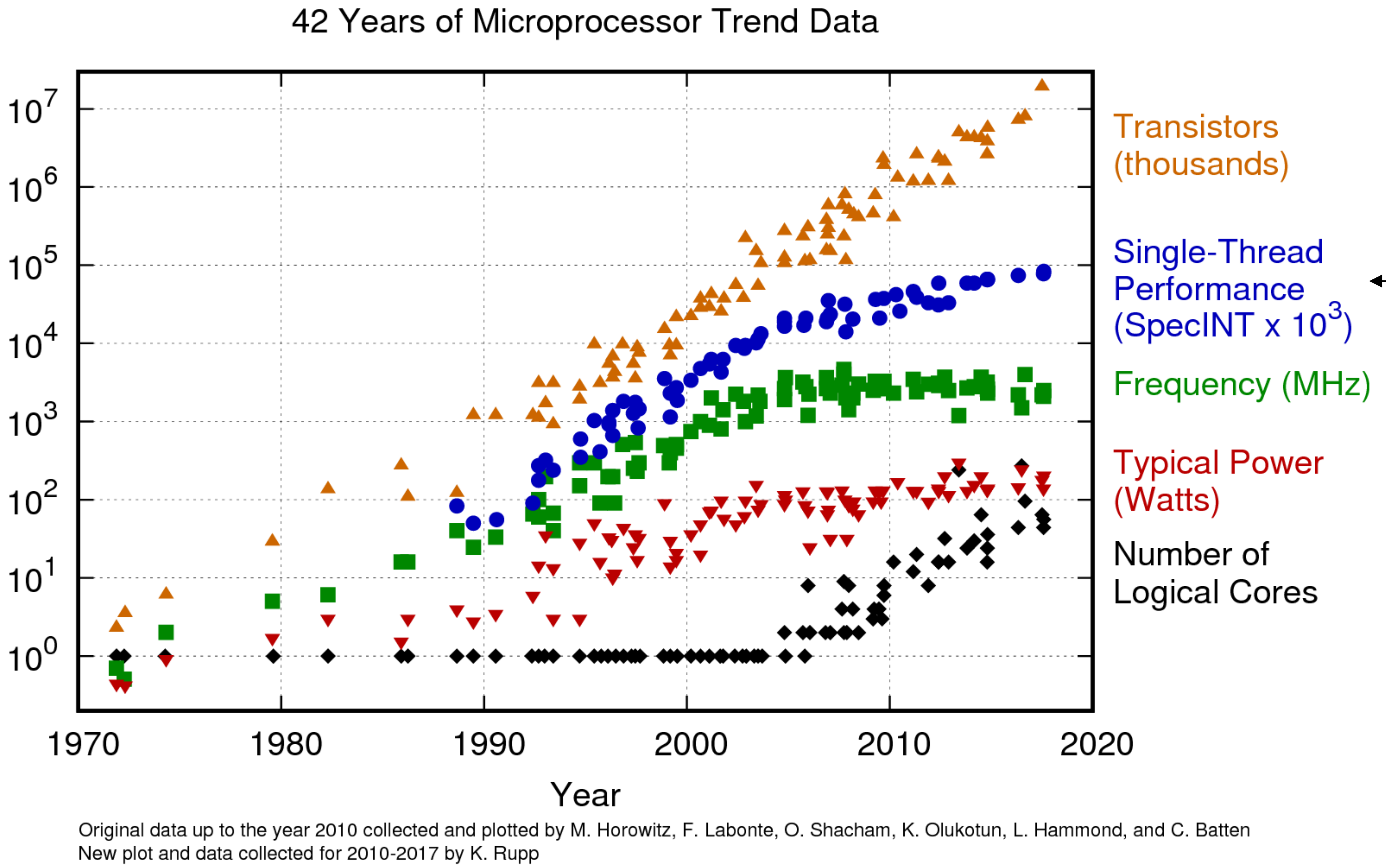
- **Input/Output** (I/O): The rate at which data can be read from a disk, a network file server, a remote server, a sensor, a user's physical inputs, etc. limits the performance of the program.

- **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.

BOSTON UNIVERSITY

Why Parallelize?



- Processors are not getting much faster these days

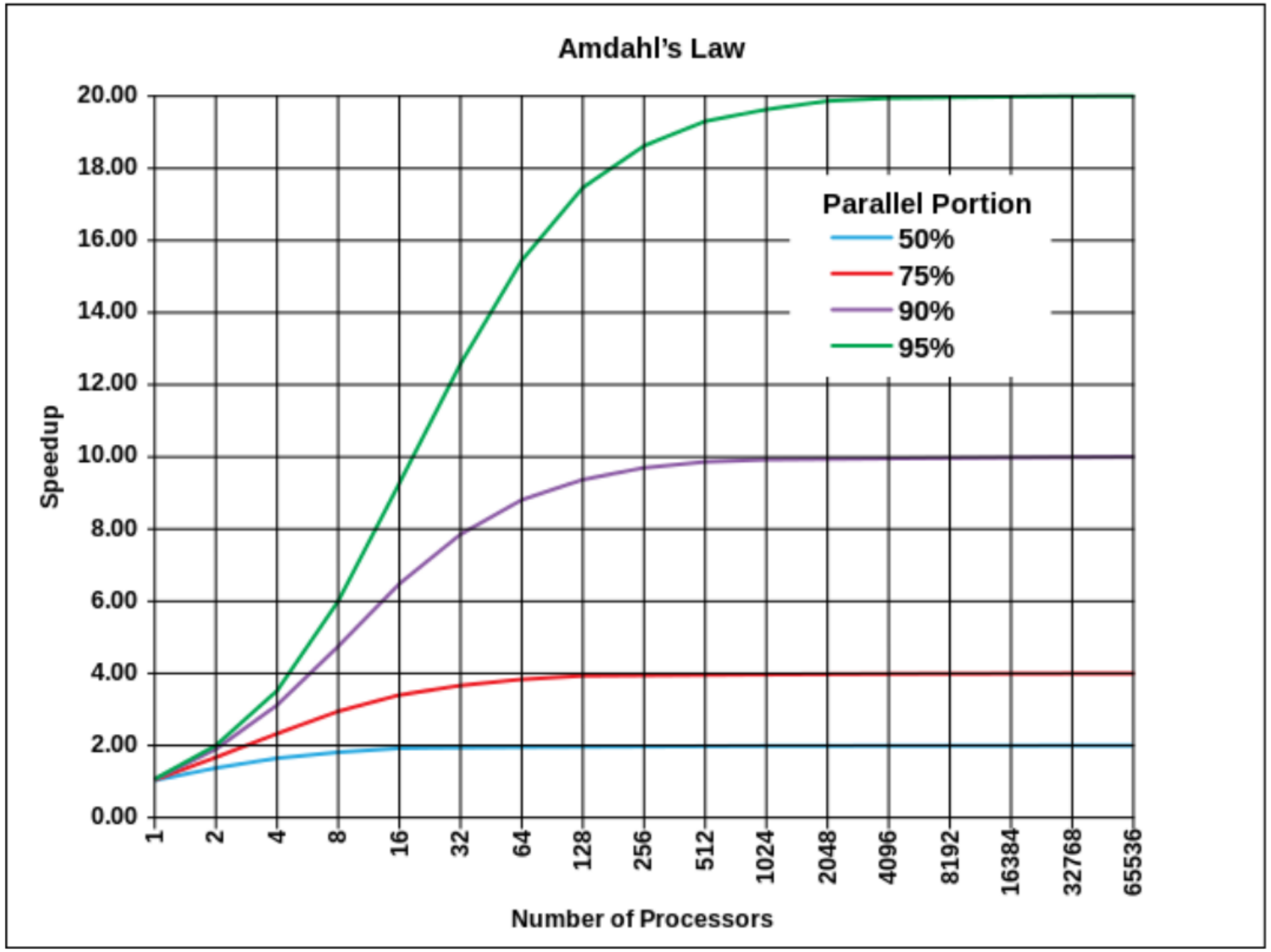
https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

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.

BOSTON UNIVERSITY

Strong and Weak Scaling

- Amdahl's Law describes the speedup for *strong* scaling:
  - For a fixed problem size, how fast can we solve it with more processors?

- Weak scaling:
  - How big of a problem size can we solve if we add more processors?
  - Example:
    - Simulate traffic flow in Boston – say this takes 1 hour on 1 processor.
    - If we increase the problem size by 8 (Boston and some surrounding cities) and use 8 processors, will it still run in 1 hour?
  - Gustafson's Law
    - *s* is the fraction of the program time that runs serially
    - *p* is the fraction that can run in parallel (s + p = 1)
    - New formula for the speedup S:

$$S = s + Np$$

BOSTON UNIVERSITY

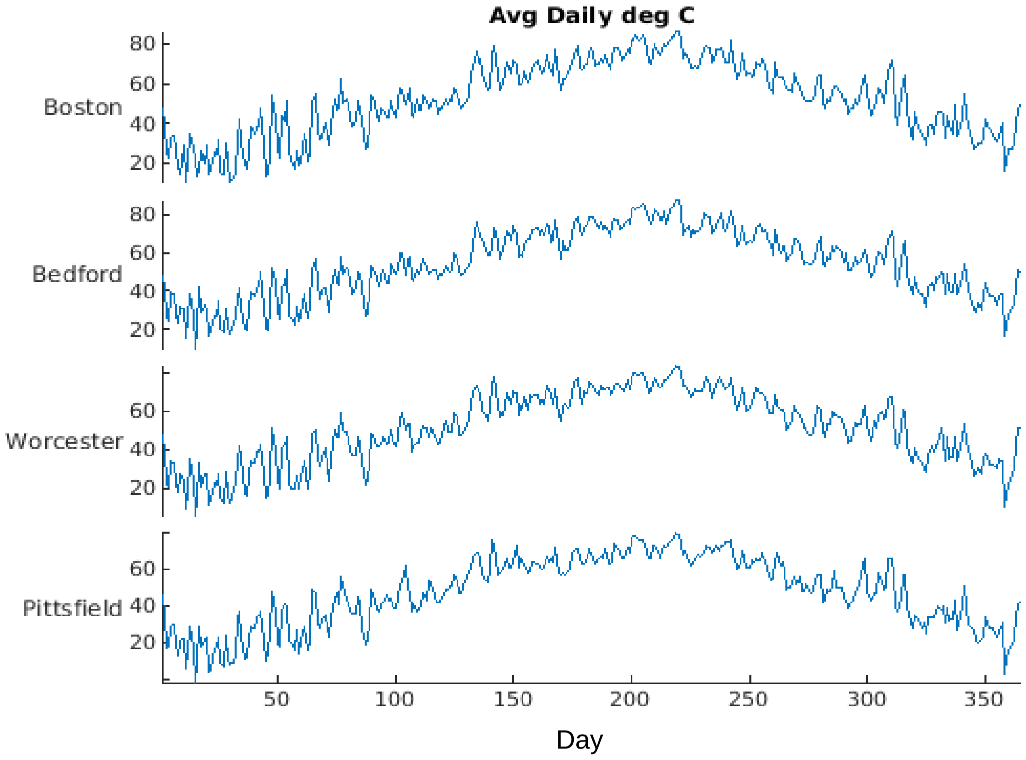
Some things can't be done in parallel.

- Gestation time for 1 female elephant to produce 1 calf: 18 months.

- 18 elephants cannot produce a calf in 1 month.

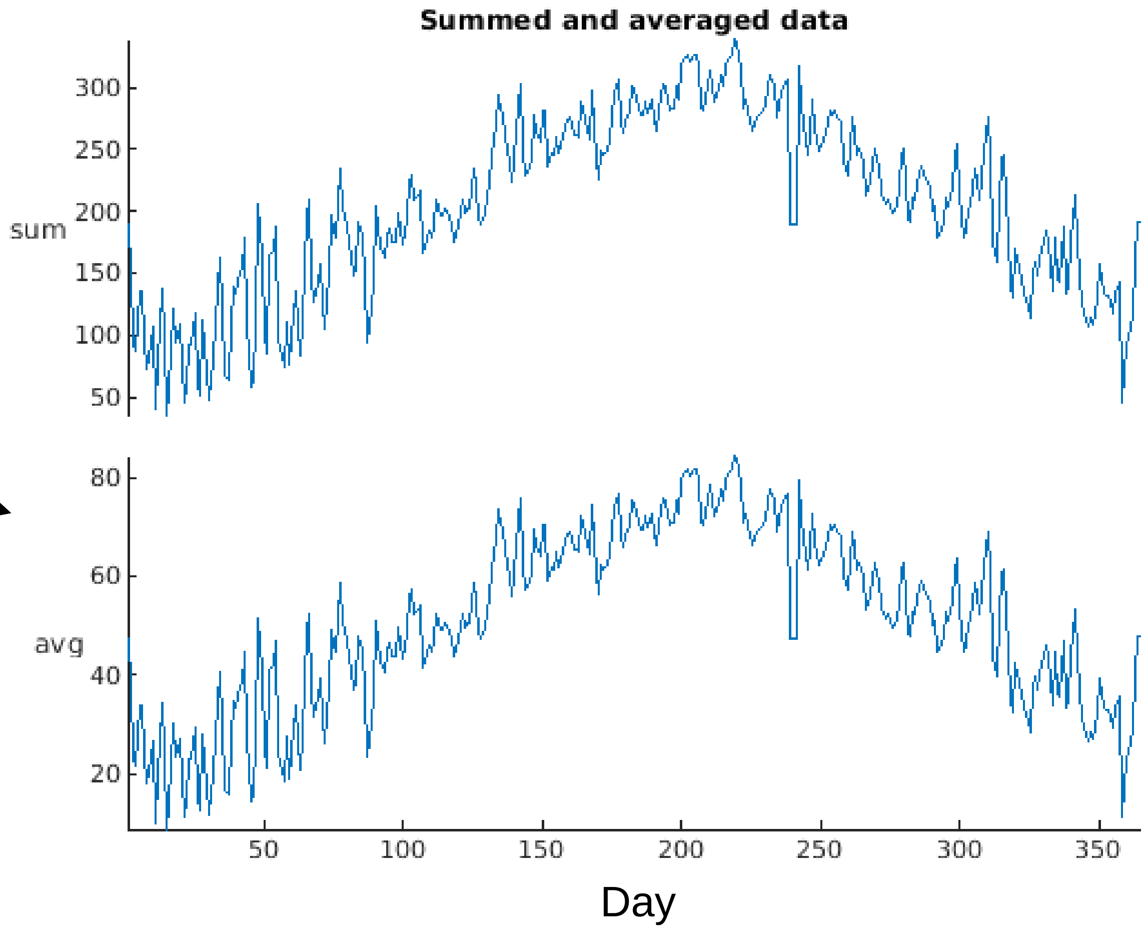
Example 1: Daily Average Temperatures

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



Example 1: Daily Average Temperatures

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



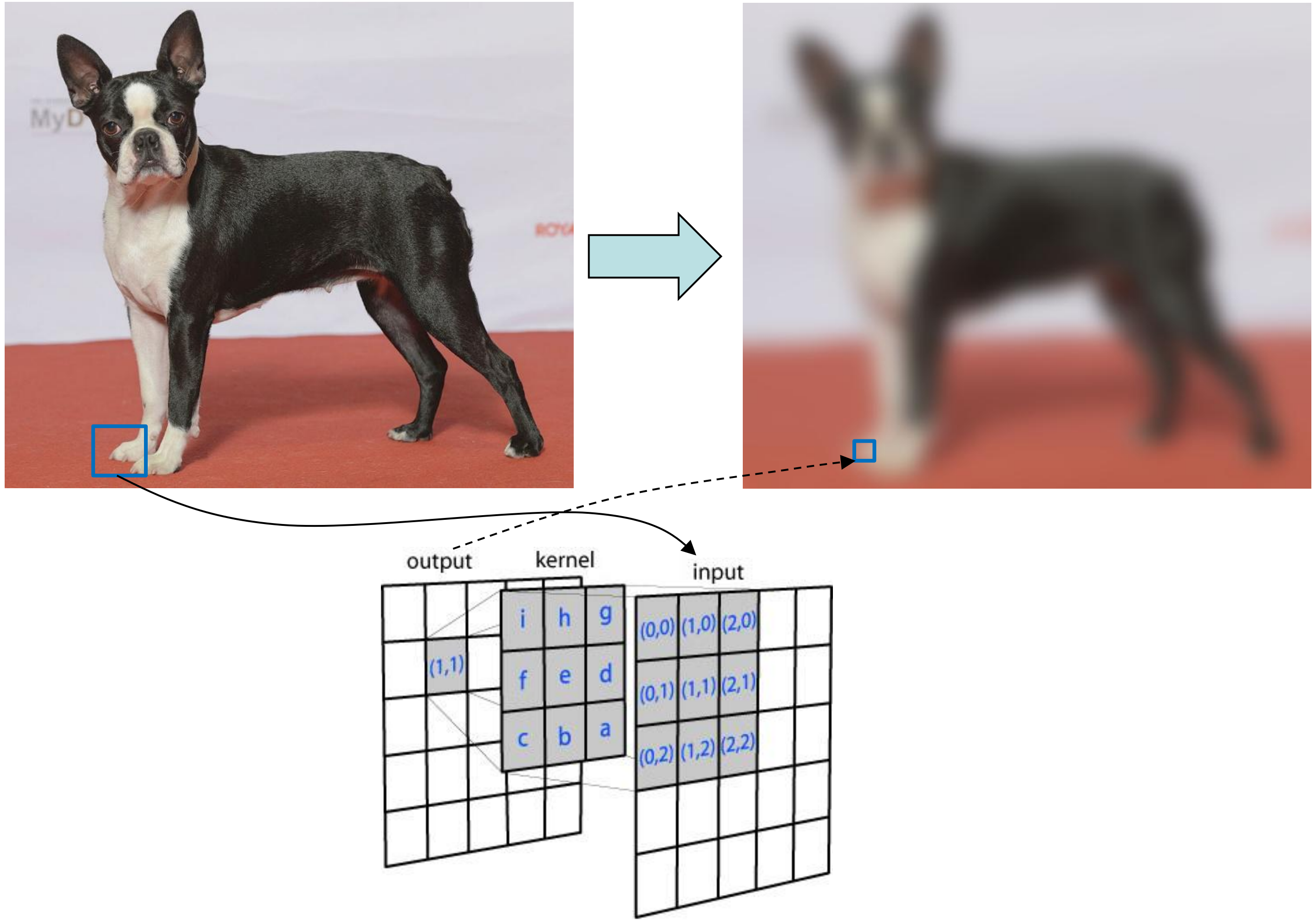
- Given 2 computer processors, how can we do this so that they do this computation in parallel?

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.

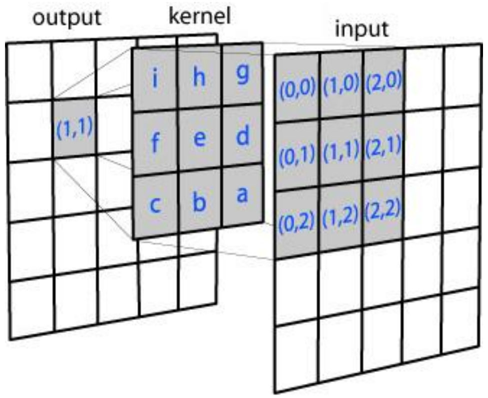


BOSTON UNIVERSITY

Example 2: Gaussian Image Blurring

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

- The image is 1033x882 pixels. The kernel is 9x9 pixels.



BOSTON UNIVERSITY

Example 3. Physical Modeling

- Simulate the air flow over a wind turbine.
  - Pressure, speed, direction at thousands of points in 3D space are determined by solving the Navier-Stokes equation.
  - Run the simulation for several wind speeds (say 5, 10, 15, and 25 m/s)
- A number of different algorithms are used to do the calculations.

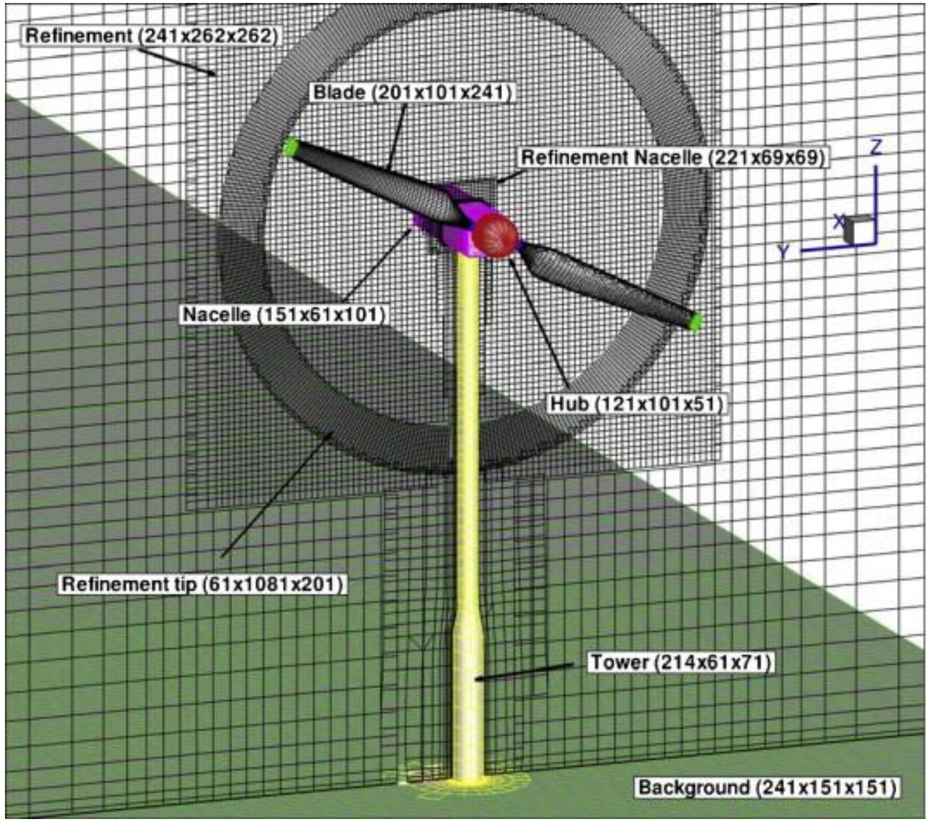
Vortices created from spinning turbine blades in a 10 m/s wind.

Yuwei Li, Kwang-Jun Paik, Tao Xing, Pablo M. Carrica, Dynamic overset CFD simulations of wind turbine aerodynamics, Renewable Energy, Volume 37, Issue 1, 2012, Pages 285-298.

BOSTON UNIVERSITY

Example 3. Physical Modeling

- The simulation is broken down spatially into a 3D grid of cells with varying cell sizes.
  - Higher cell density where there's a lot of things changing quickly, lower where there's less action.
- How might this computation be parallelized?
  - Let's use 8 processors.



BOSTON UNIVERSITY

Example 4: *k*-mer counting

- *k*-mers are repeated sets of nucleotides in genomic sequences. *k* is the length of the set.
  - Example: 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 *k*.

```
AGTCCCCGTCTTGCCGCGCGGGGGCGGGCGCGGGAAAAAAGCCGCGCGGGGGCGC
CCGCGGGAAGGCAGCCCCGCGGCGCGCGGGGGGAGGGGCGGCGCCCGCGGGGGAG
CGGCCGGCTCCGGGGGAGGGACGGGGAAGGGGGCGCGCGGGGCTGCCCTGCCGCC
CGCCCGCCGCCGCCGCCCGCCTTCGCGCCCCCCCCAAAAAACACCCCCCCCGGA
```

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

BOSTON UNIVERSITY

Example 4: *k*-mer counting

- 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

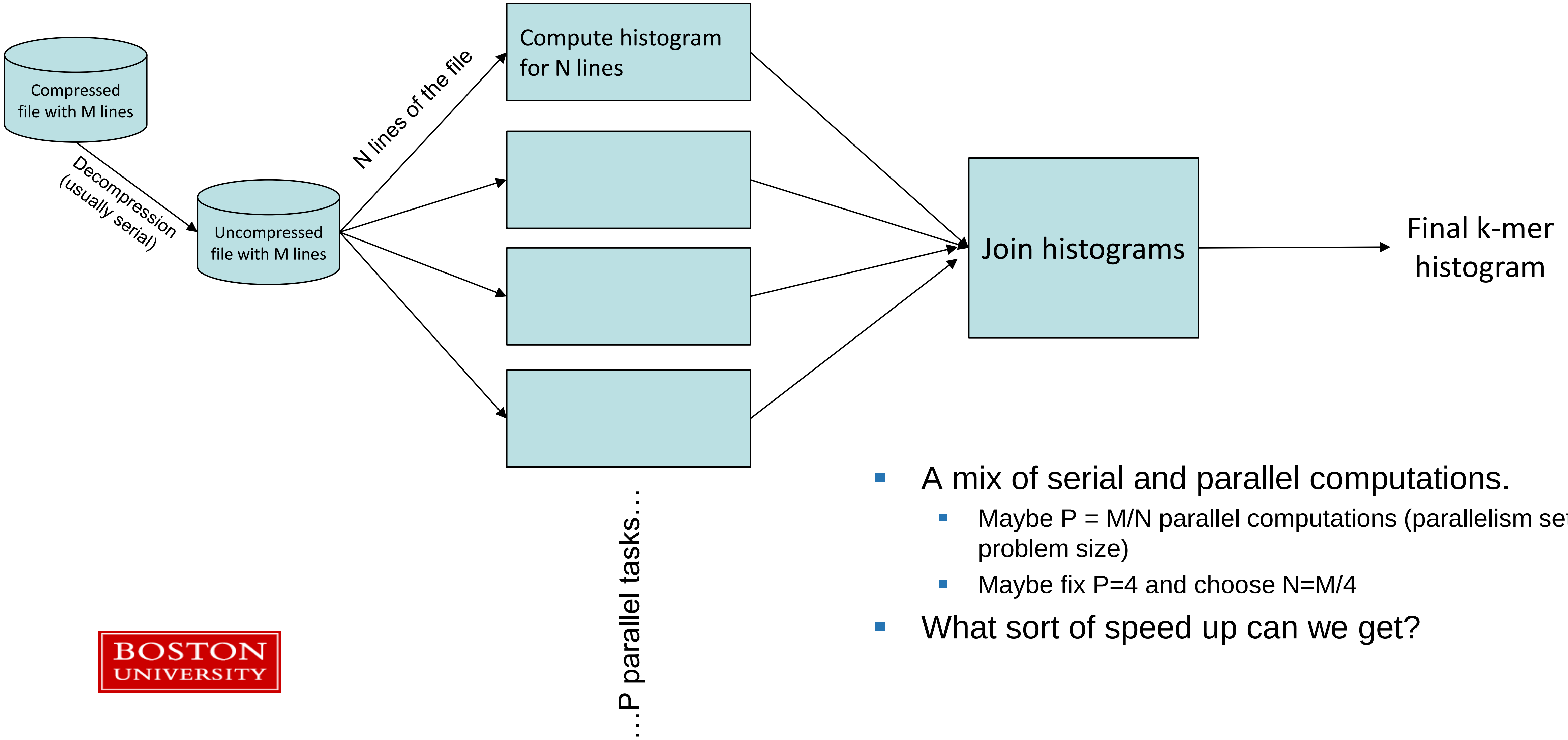
…etc…

How can we split this up into parallel computations?

Which steps can happen in parallel?

BOSTON UNIVERSITY

Example 4: *k*-mer counting



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

BOSTON UNIVERSITY

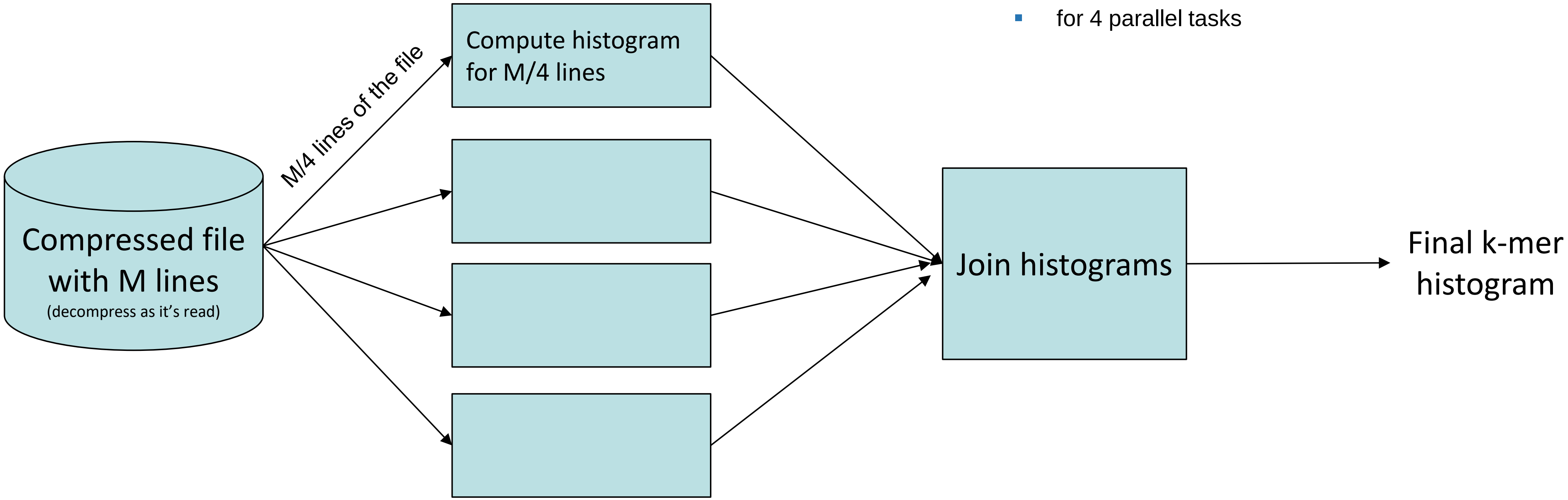
The basics of the Work-Depth Model

- 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 (**T_1**): total amount of tasks to complete for a single processor
- Depth (**T_∞**): 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 (**P**=4) as a concrete example.

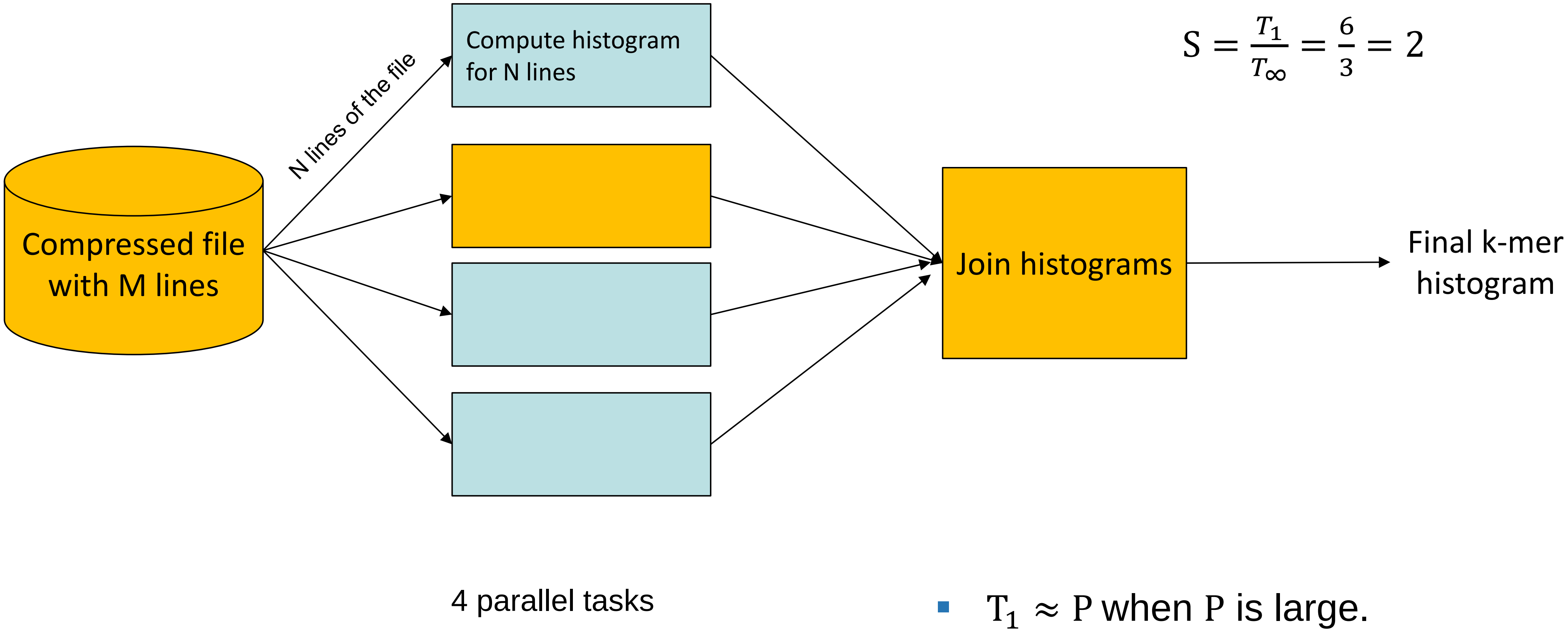
- $\mathbf{T_1} \approx \mathbf{P}$ for a large number of parallel tasks P if $\mathrm{P} \gg 2$

BOSTON UNIVERSITY

Work-Depth Model

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

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

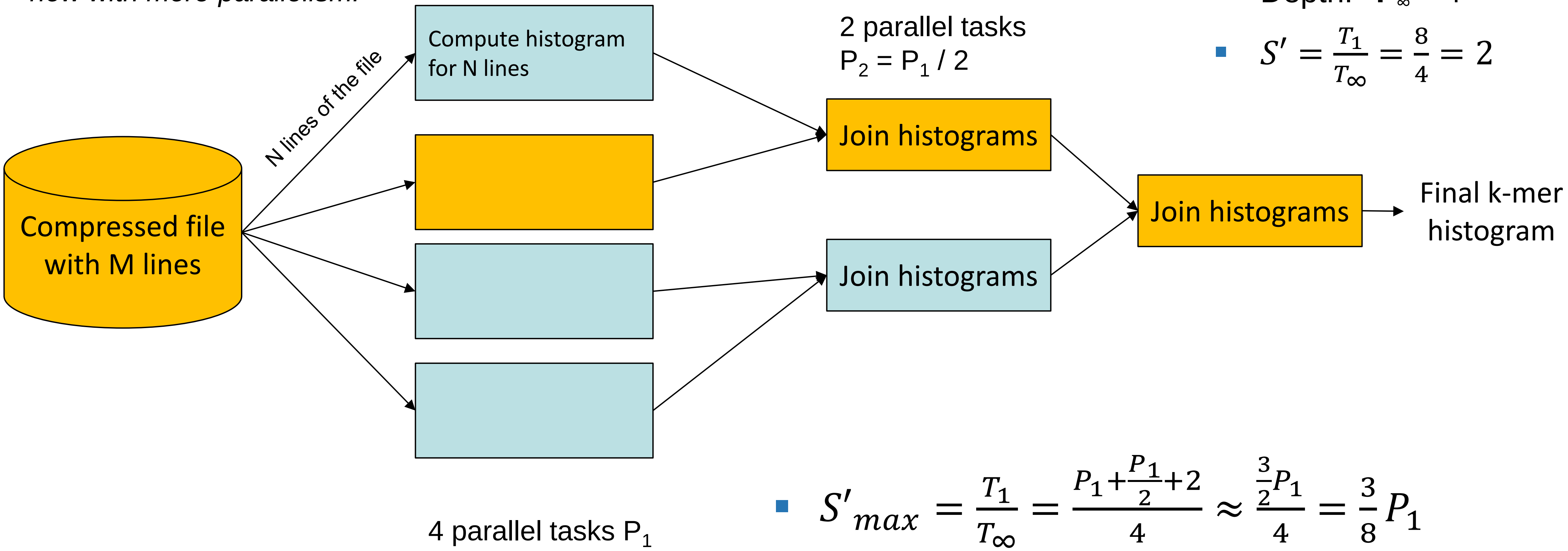


- $T_1 \approx P$ when P is large.
- $S_{max} = \frac{T_1}{T_\infty} = \frac{1}{3}P$

BOSTON UNIVERSITY

Work-Depth Model

now with more parallelism!



With P_1=4, P_2=2

- Work: $\mathbf{T_1} = 8$
- Depth: $\mathbf{T}_\infty = 4$
- $S' = \frac{T_1}{T_\infty} = \frac{8}{4} = 2$

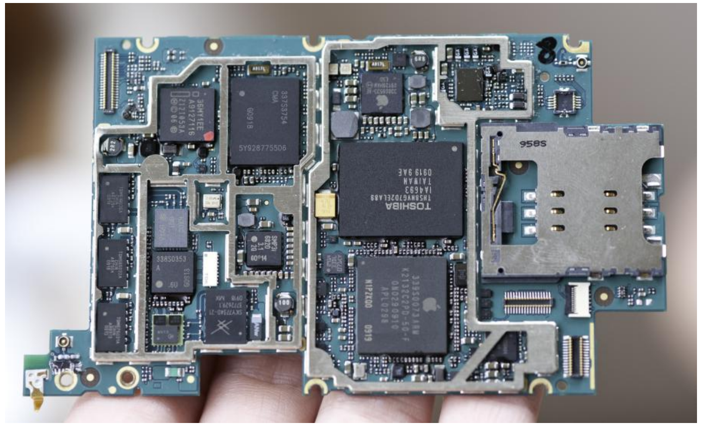
- $S'_{max} = \frac{T_1}{T_\infty} = \frac{P_1 + \frac{P_1}{2} + 2}{4} \approx \frac{\frac{3}{2}P_1}{4} = \frac{3}{8}P_1$
- Compare with the previous slide: this is a bit faster but is more complicated due to the extra parallel component.

BOSTON UNIVERSITY

Outline

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

Hardware for Parallel Computation



iPhone motherboard

Lenovo ThinkSystem HPC cluster

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

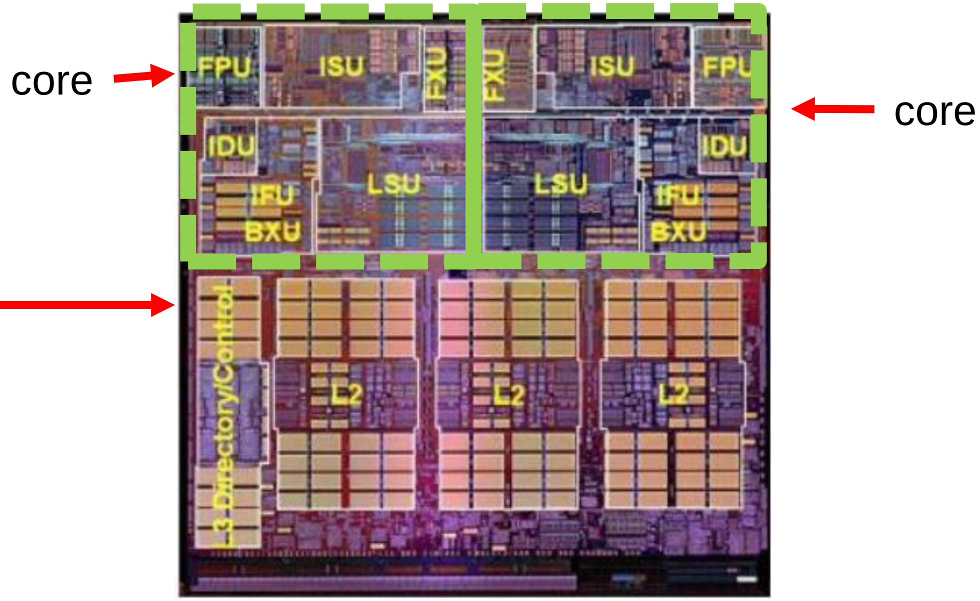
BOSTON UNIVERSITY

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

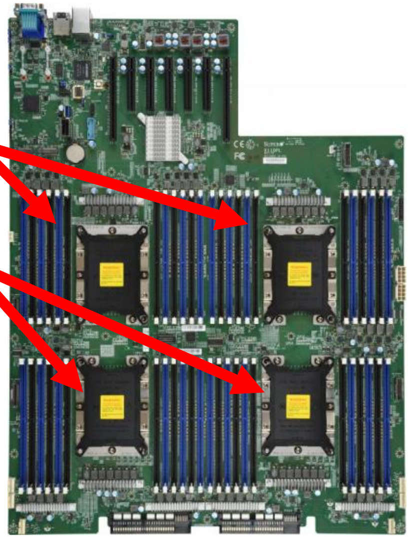


POWER4 circuit view

BOSTON UNIVERSITY

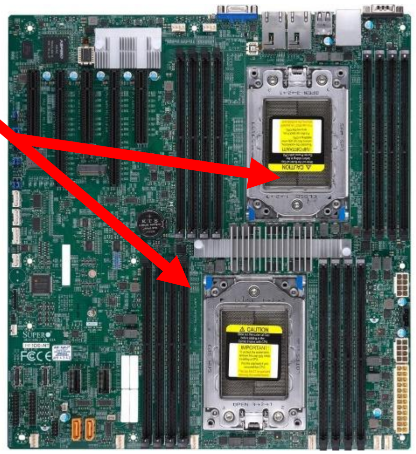
Modern configurations

- Quad Intel Xeon CPUs
- Up to 56 cores per CPU



High-end servers

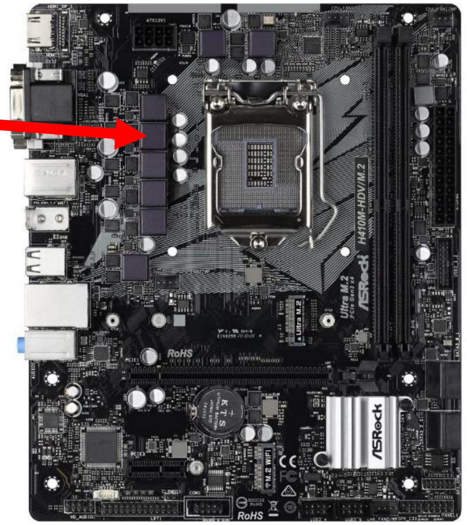
- Dual AMD Epyc CPUs
- Up to 128 cores per CPU



BOSTON UNIVERSITY

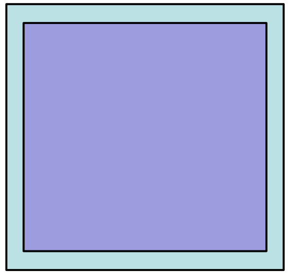
Common desktop

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

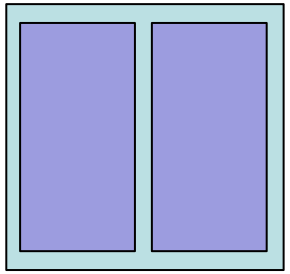


- For PC and server hardware the high end has very high core counts.
- Entry-level systems still have multiple cores.
- All SCC compute nodes are dual socket Intel-based systems.
  - 16-64 total cores per compute server

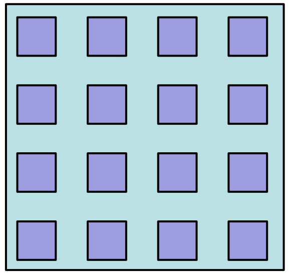
CPUs and cores



1 CPU, 1 core
1 program at a time



1 CPU, 2 cores
2 programs simultaneously

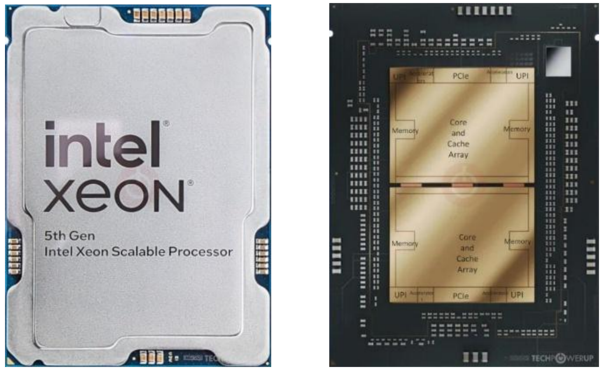


1 CPU, 16 cores
16 programs simultaneously

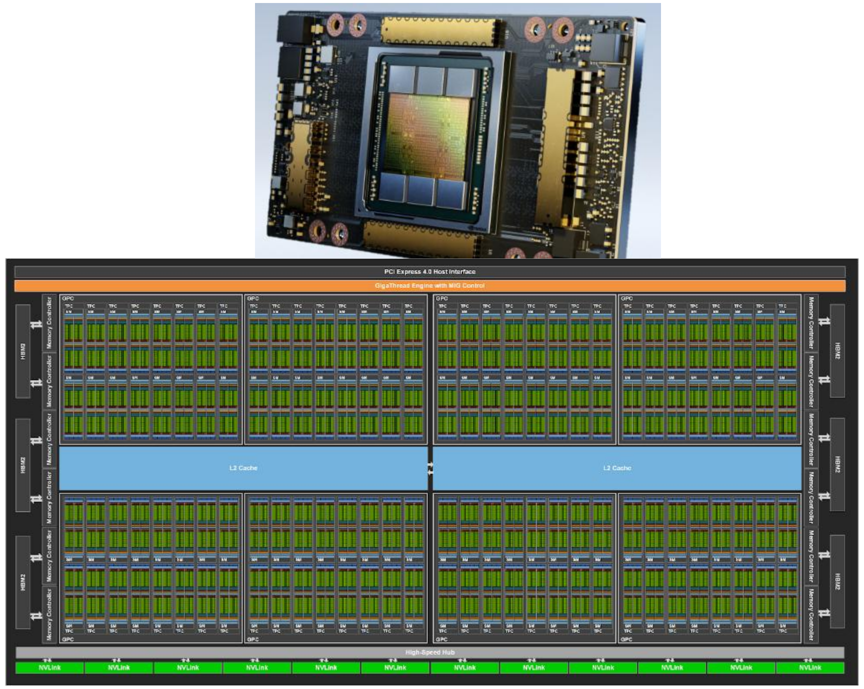
- “CPU” typically refers today to the physical packaging of multiple cores.
- CPU, processor, and core are sometimes used interchangeably to mean “core”.

BOSTON UNIVERSITY

GPU Hardware



SCC CPU



SCC GPU

Intel Xeon Gold 6526Y:

Clock speed: **2.8** GHz
8 instructions per cycle with AVX512
CPU - 32 cores

2.6 x 8 x 28 =
0.665 Teraflops double precision

NVIDIA Tesla A100:

Single instruction per cycle
6912 CUDA cores

9.7 Teraflops double precision

BOSTON UNIVERSITY

Double precision: 64-bit floating point

The Illusion of Parallelism

- All modern operating system will run more programs than there are available cores.
  - This is called *concurrency*.
- 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

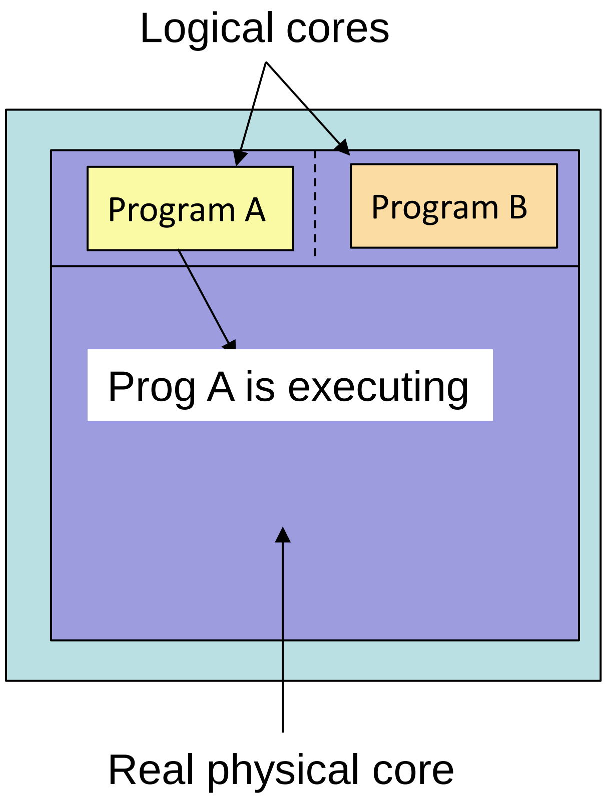
- If N programs are trying to compute things, then on a single core in a given timeframe each gets 1/N of the runtime.
  - Example: 4 programs, each running “for” loops and doing calculations.
  - On 1 core in 1 minute each will execute for ¼ of a minute (15 sec).

BOSTON UNIVERSITY

Logical Cores

aka “hyperthreading” or “hardware threads”

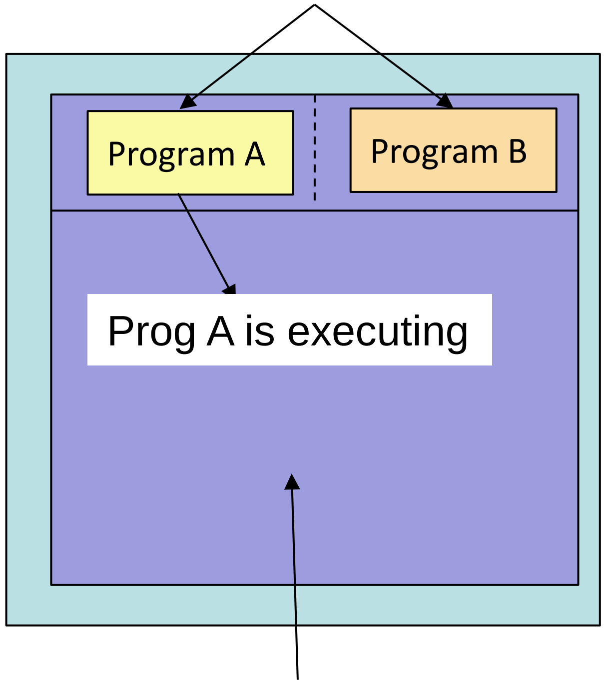
- CPUs with logical cores have:
  - additional hardware that lets a program (B) have its *execution state* pre-loaded onto a core while another program (A) is executing on that core.
  - The extra hardware allows the OS to switch the physical core to run the other program (from A to B) very quickly and 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)

Logical cores aka hands

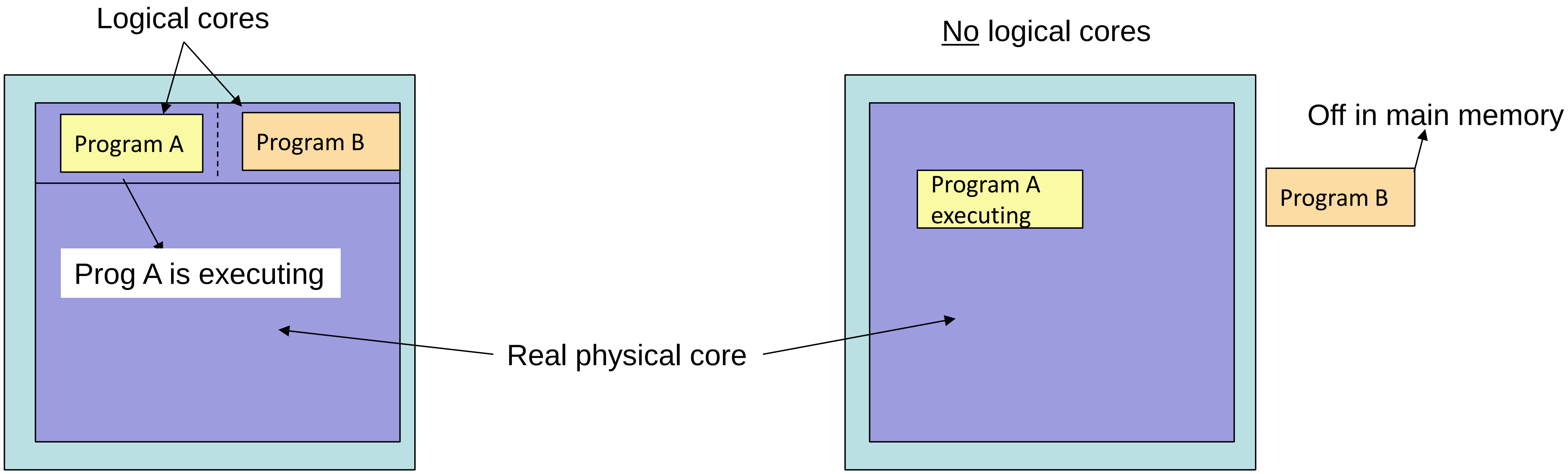


Real physical core
aka juggler

No logical cores
1 juggler

2 logical cores
Still 1 juggler

BOSTON UNIVERSITY

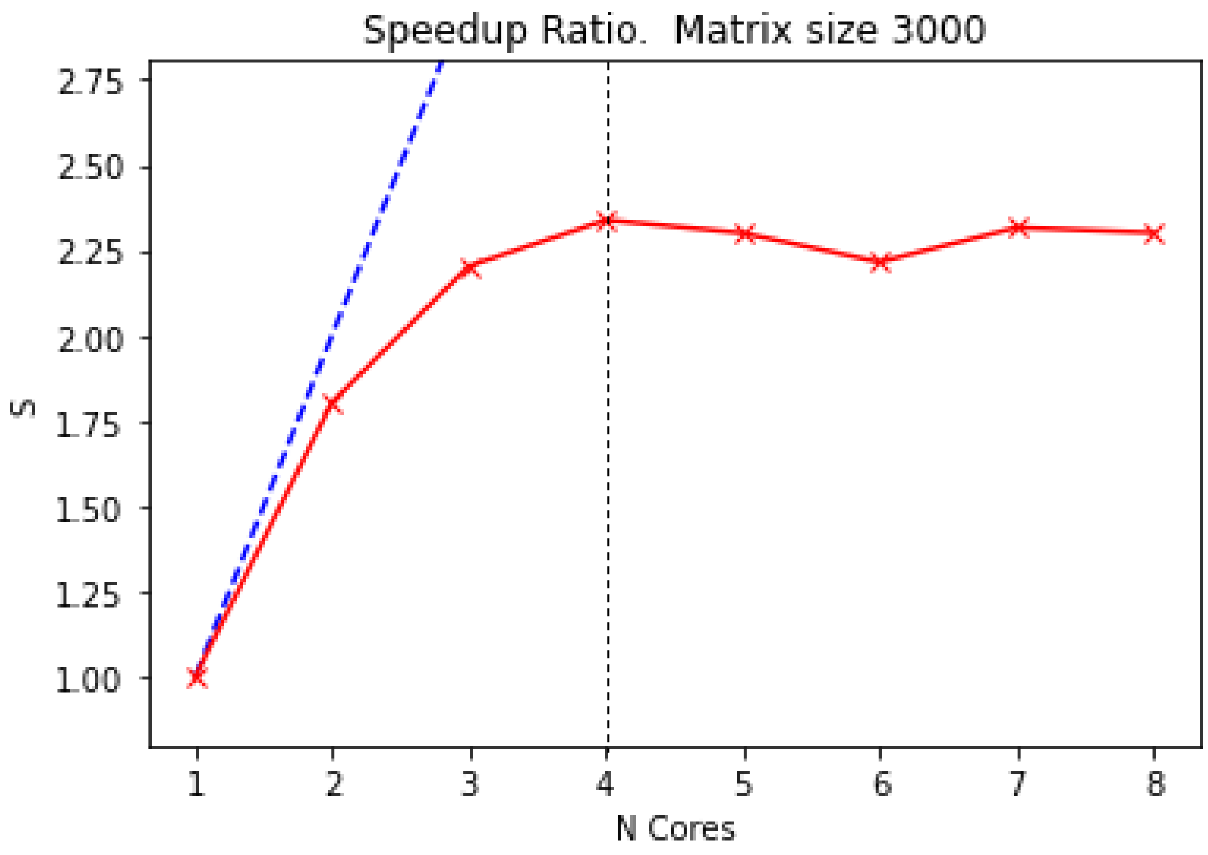


- *Without* logical cores the program switching is slower.
  - Physical core: maybe ~5-15 μs to switch the running program
  - Logical core: ~2-4x faster.

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

BOSTON UNIVERSITY

Logical cores in action



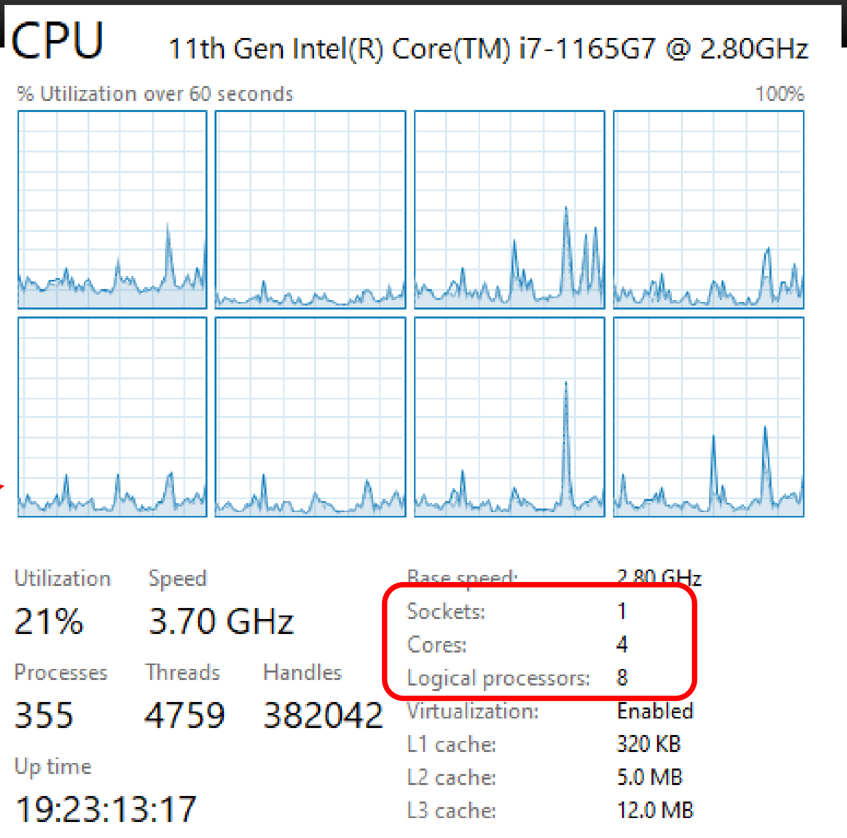
Intel i5-9300h CPU

- 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 Task Manager 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
```

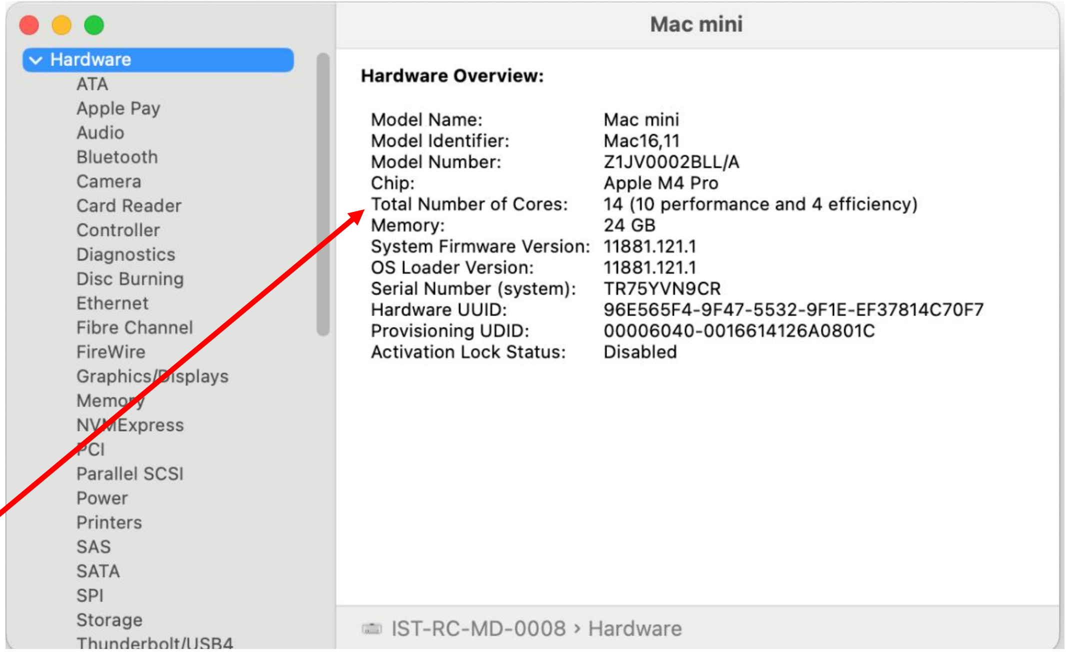
- Mac OSX (Intel CPUs):
  - Using the Terminal app

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

BOSTON UNIVERSITY

New CPUs: Performance and Efficiency Cores

- Apple M CPUs (M1, M2, etc) and recent Intel & AMD CPUs:
  - Performance cores: highest computing speed
  - Efficiency cores: slower, lower power consumption, for less important tasks
    - AMD calls these “c cores”
- How to check:
  - Use the Task Manager to get your CPU model number.
  - Intel: Look it up on the Intel Ark site.
  - AMD: same but use the AMD site.
  - Mac OSX: Use Spotlight to find the “System Report” utility.



BOSTON UNIVERSITY

Logical Cores and Your Program

- On your personal or lab computers, check to see if logical cores are present.

- If you’re CPU-bound, only use physical cores for your code.
  - Or “performance” cores.

- 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.**

BOSTON UNIVERSITY

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. 12.4

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

BOSTON UNIVERSITY

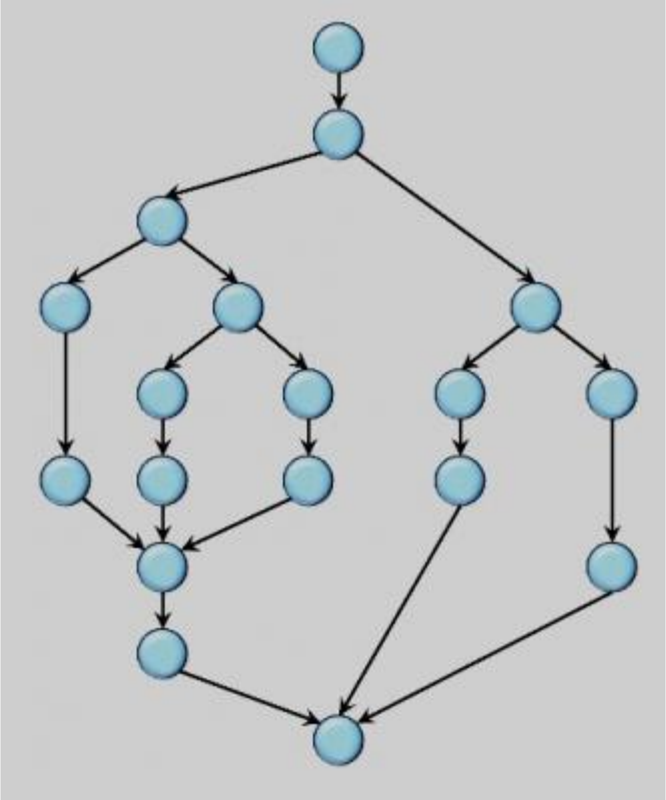
Outline

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

BOSTON UNIVERSITY

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.
- Here's a few examples. There are *lots* more than we have time for here!



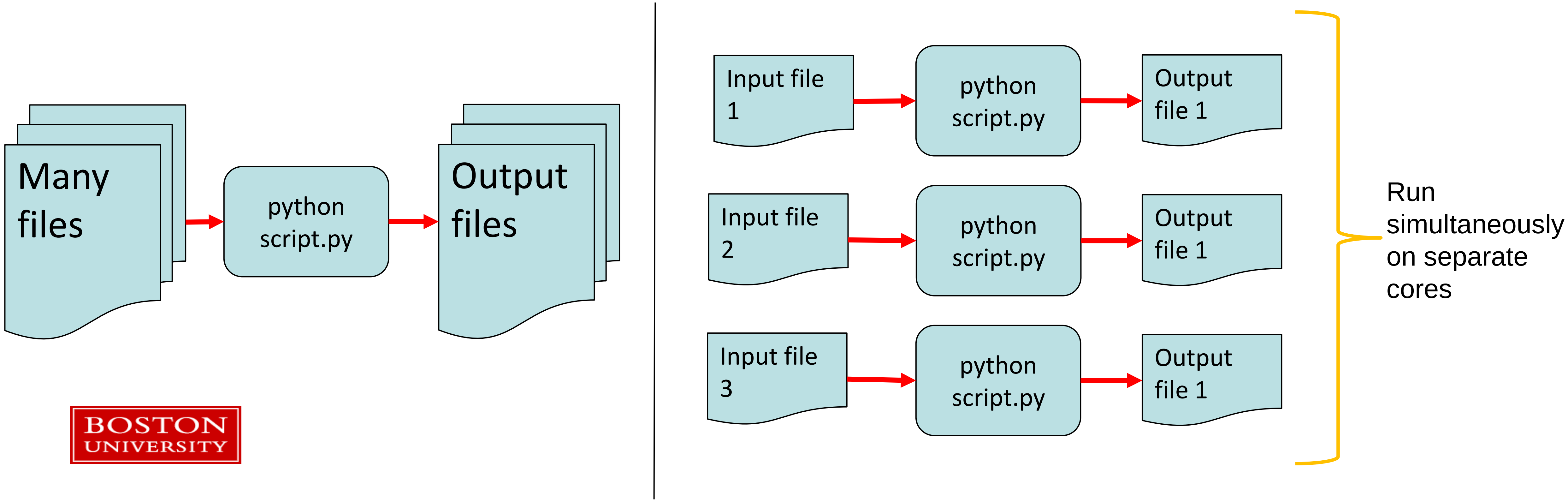
Embarrassingly Parallel

- 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: 1+2+3+4+5 + 6+7+8+9+10
  - Or break it down into even smaller computations.

BOSTON UNIVERSITY

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

Embarassingly Parallel

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

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

Serial Matlab code

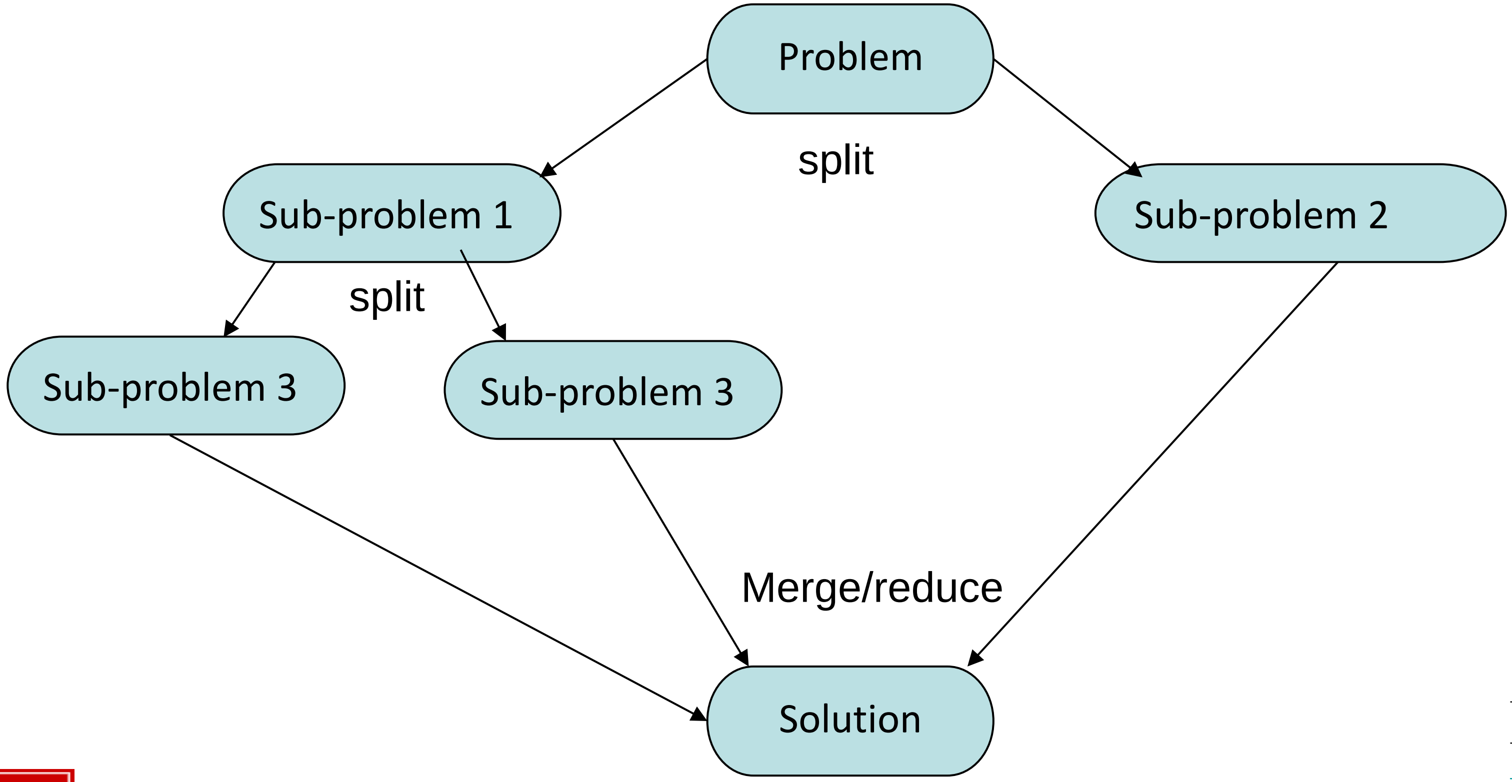
```
x = [1,2,3,4,5];

% Launch 5 Matlab processes to
% run in parallel
parpool(5) ;
parfor i = x
      y(i) = some_func(x(i));
```

Parallel Matlab code

BOSTON UNIVERSITY

Divide & Conquer

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



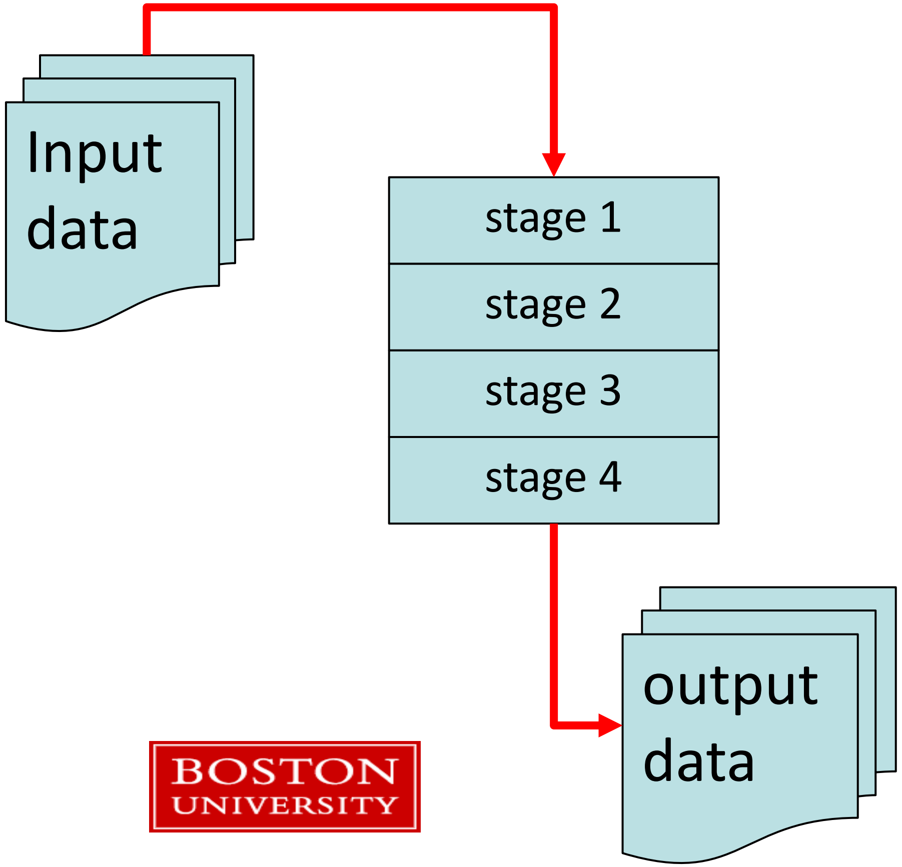
Sub-problems 1 and 2 can be executed in parallel.

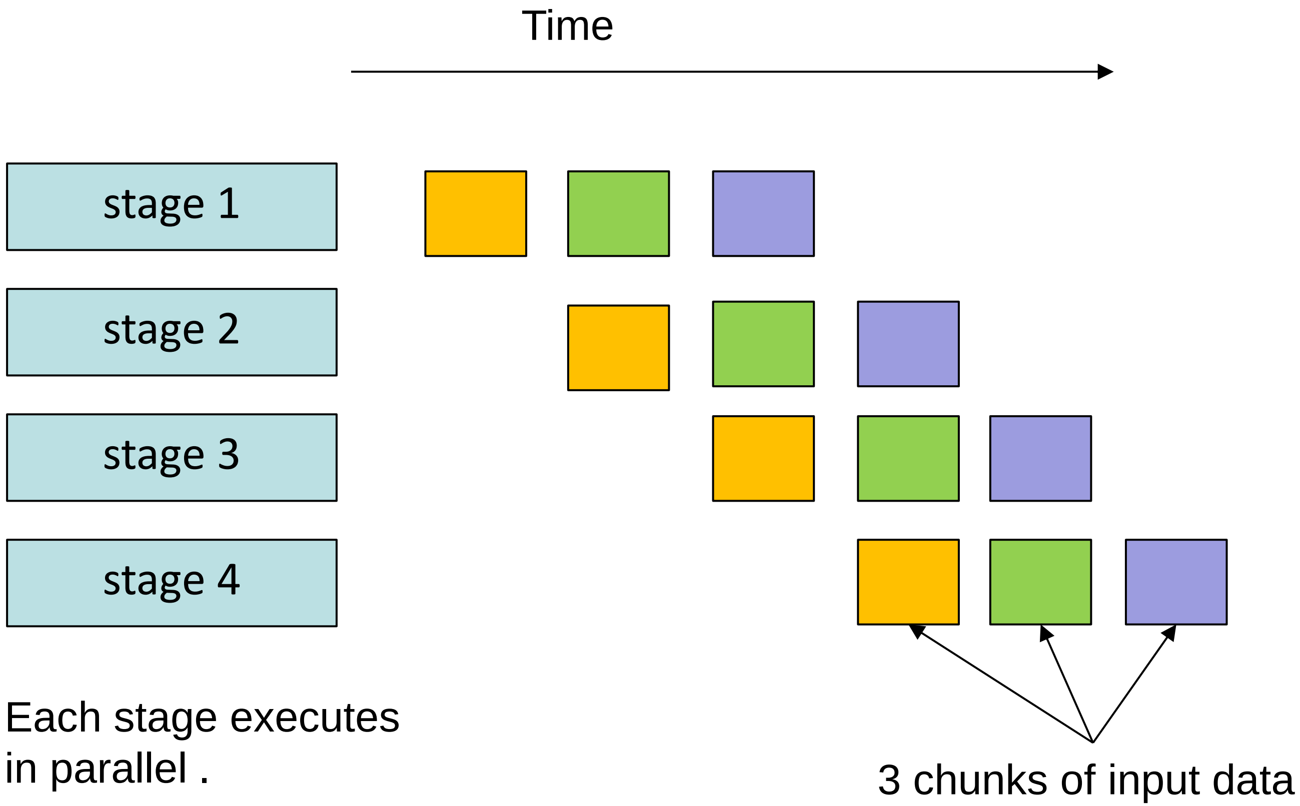
Or both 3's with 2.

Example: the famous MapReduce algorithm.

Pipeline

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

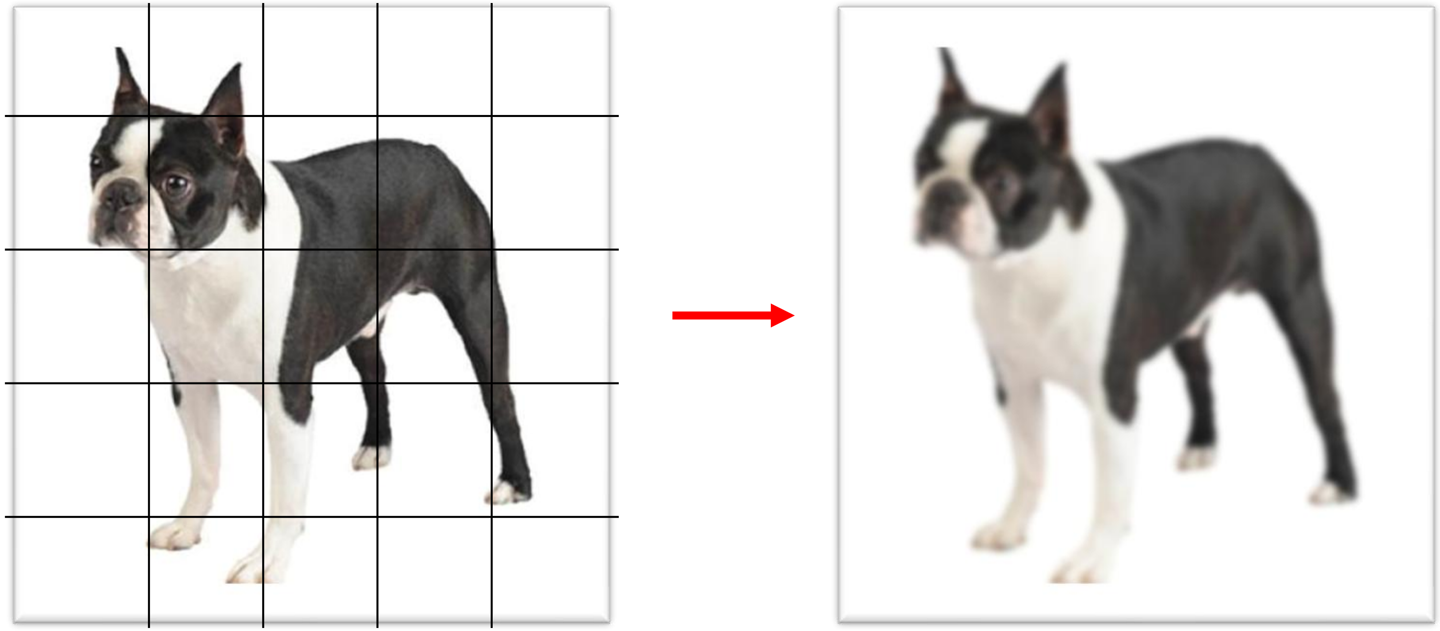




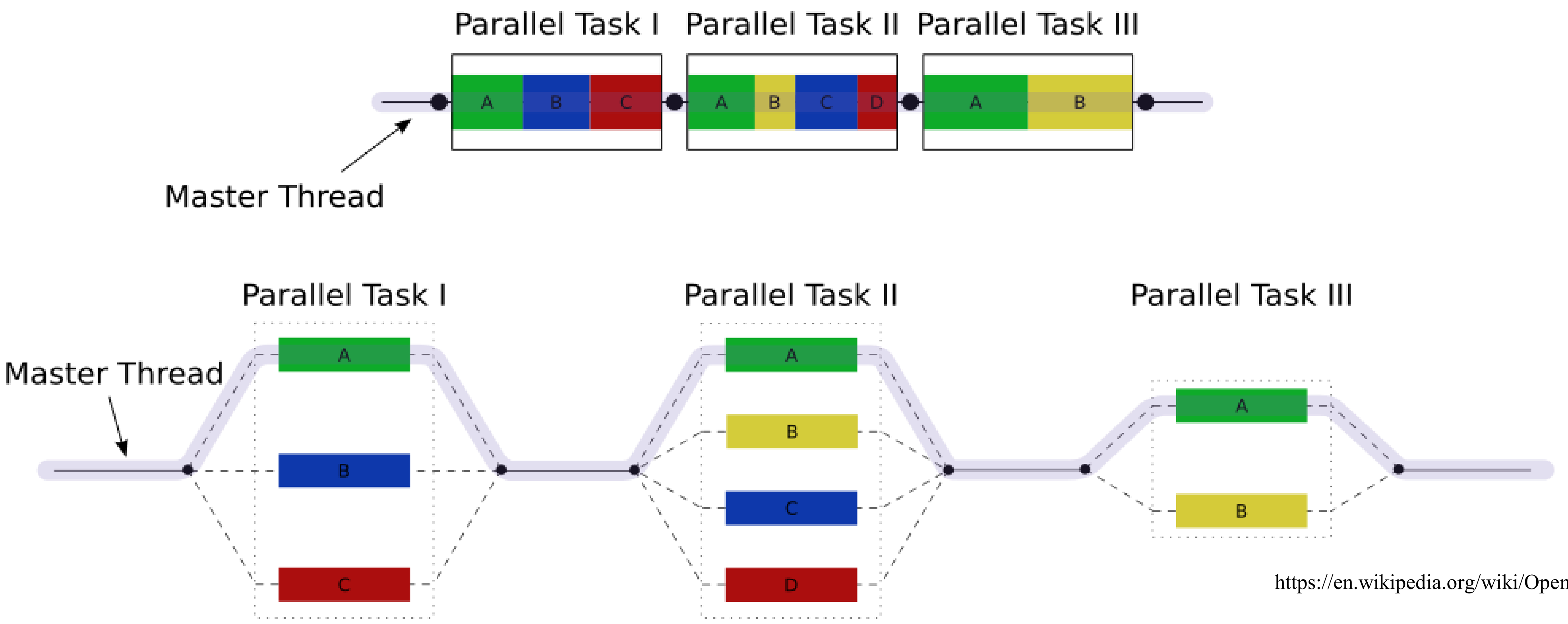
BOSTON UNIVERSITY

Geometric

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



BOSTON UNIVERSITY



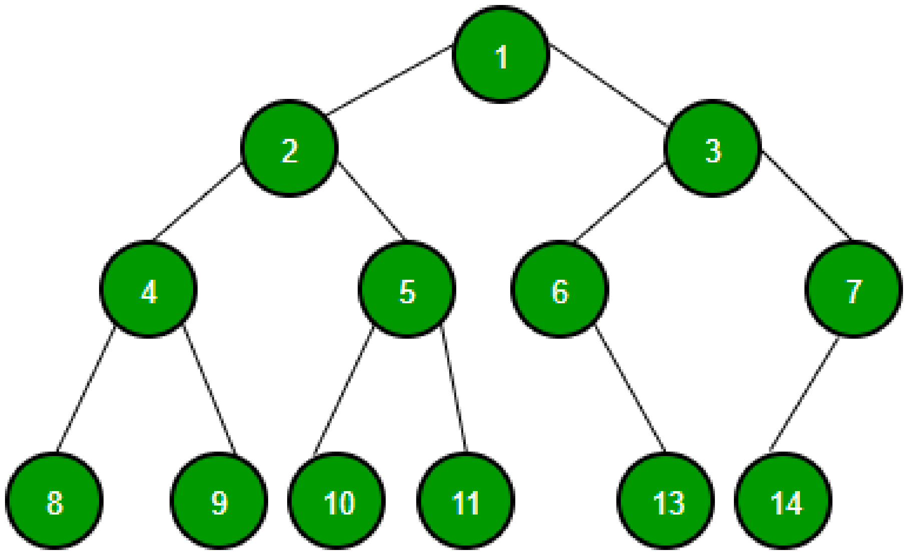
https://en.wikipedia.org/wiki/OpenMP

- 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.

BOSTON UNIVERSITY

Data Structure Driven

- The way your data is organized can influence the choice of algorithms.
- For common data structures do a literature search for parallel algorithms – you may get lucky.
- For example: sum the elements of this binary tree in parallel.



BOSTON UNIVERSITY

Outline

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

Monitoring on Linux 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+
```

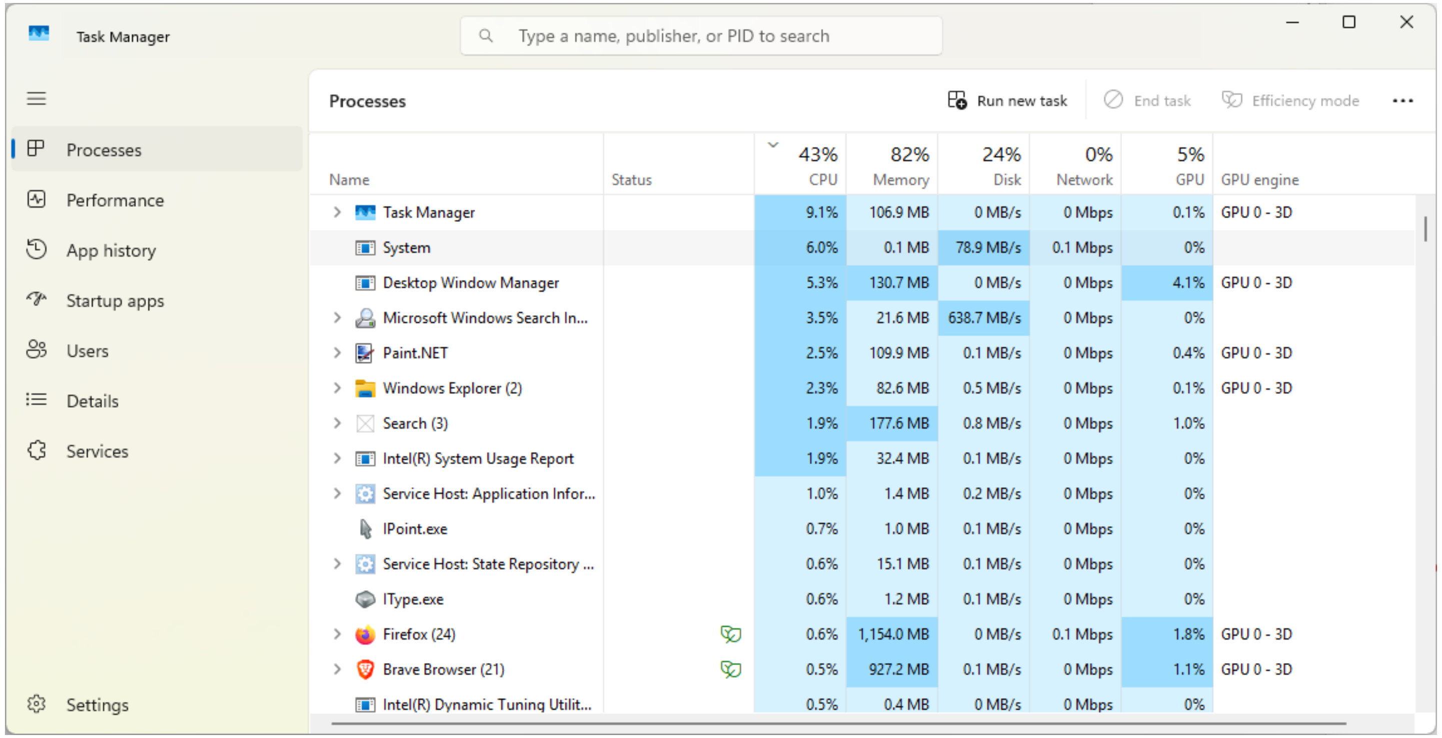
- On the SCC*, use *top*
- To see your processes only: top -u username
- To exit *top*: press ‘q’

- 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.

* Alternate tool: *htop*

Process Monitoring - Windows

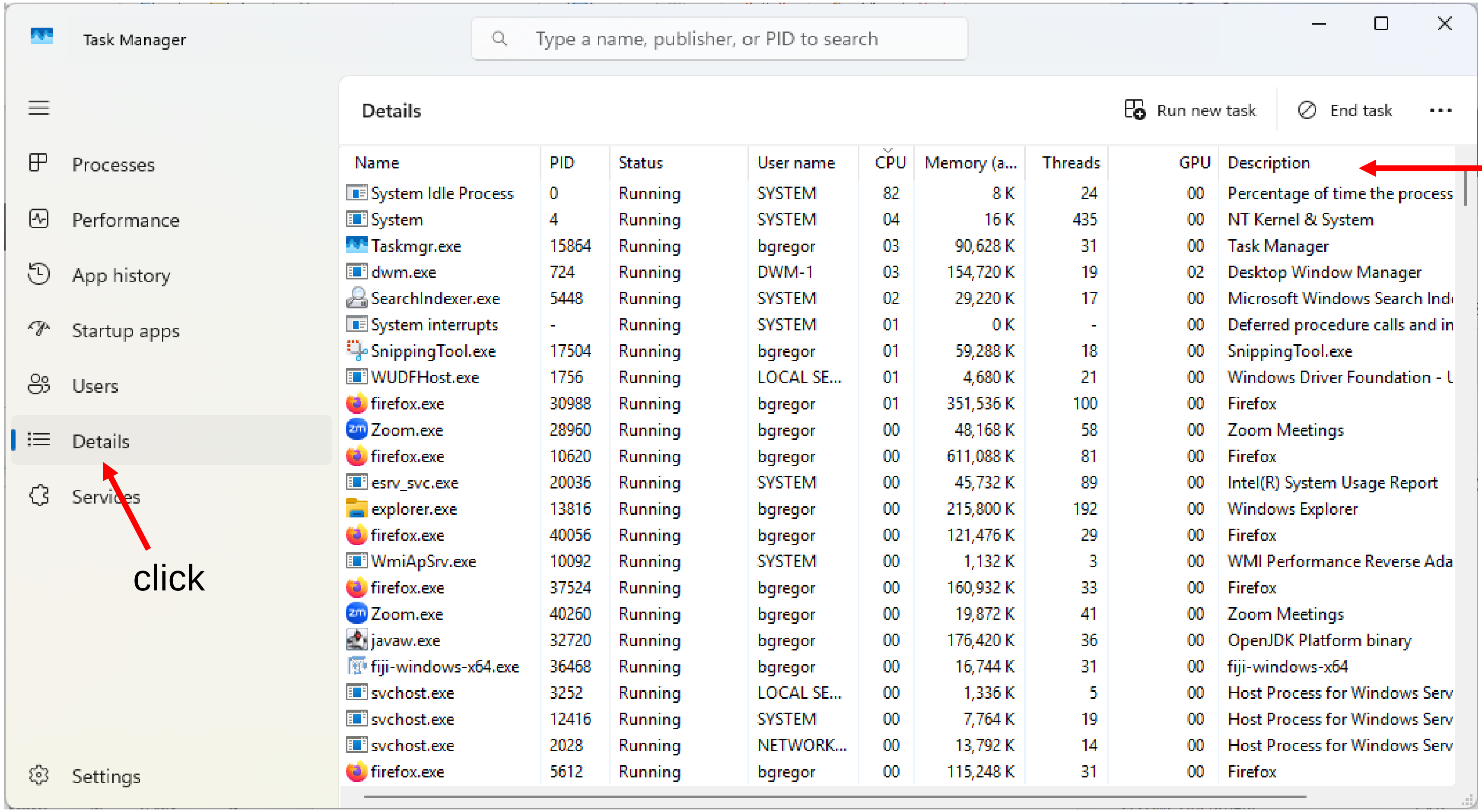
- Windows Task Manager
  - Right-click on task bar
- 100% of CPU means all cores are 100% utilized.
  - On a 4-core computer, if your program is running at 25% CPU then it's fully using 1 core.



Windows 11

BOSTON UNIVERSITY

Process Monitoring - Windows



Name	PID	Status	User name	CPU	Memory (a...	Threads	GPU	Description
System Idle Process	0	Running	SYSTEM	82	8 K	24	00	Percentage of time the process
System	4	Running	SYSTEM	04	16 K	435	00	NT Kernel & System
Taskmgr.exe	15864	Running	bgregor	03	90,628 K	31	00	Task Manager
dwm.exe	724	Running	DWM-1	03	154,720 K	19	02	Desktop Window Manager
SearchIndexer.exe	5448	Running	SYSTEM	02	29,220 K	17	00	Microsoft Windows Search Ind
System interrupts	-	Running	SYSTEM	01	0 K	-	00	Deferred procedure calls and in
SnippingTool.exe	17504	Running	bgregor	01	59,288 K	18	00	SnippingTool.exe
WUDFHost.exe	1756	Running	LOCAL SE...	01	4,680 K	21	00	Windows Driver Foundation - U
firefox.exe	30988	Running	bgregor	01	351,536 K	100	00	Firefox
Zoom.exe	28960	Running	bgregor	00	48,168 K	58	00	Zoom Meetings
firefox.exe	10620	Running	bgregor	00	611,088 K	81	00	Firefox
esrv_svc.exe	20036	Running	SYSTEM	00	45,732 K	89	00	Intel(R) System Usage Report
explorer.exe	13816	Running	bgregor	00	215,800 K	192	00	Windows Explorer
firefox.exe	40056	Running	bgregor	00	121,476 K	29	00	Firefox
WmiApSrv.exe	10092	Running	SYSTEM	00	1,132 K	3	00	WMI Performance Reverse Ada
firefox.exe	37524	Running	bgregor	00	160,932 K	33	00	Firefox
Zoom.exe	40260	Running	bgregor	00	19,872 K	41	00	Zoom Meetings
javaw.exe	32720	Running	bgregor	00	176,420 K	36	00	OpenJDK Platform binary
fiji-windows-x64.exe	36468	Running	bgregor	00	16,744 K	31	00	fiji-windows-x64
svchost.exe	3252	Running	LOCAL SE...	00	1,336 K	5	00	Host Process for Windows Serv
svchost.exe	12416	Running	SYSTEM	00	7,764 K	19	00	Host Process for Windows Serv
svchost.exe	2028	Running	NETWORK...	00	13,792 K	14	00	Host Process for Windows Serv
firefox.exe	5612	Running	bgregor	00	115,248 K	31	00	Firefox

click

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

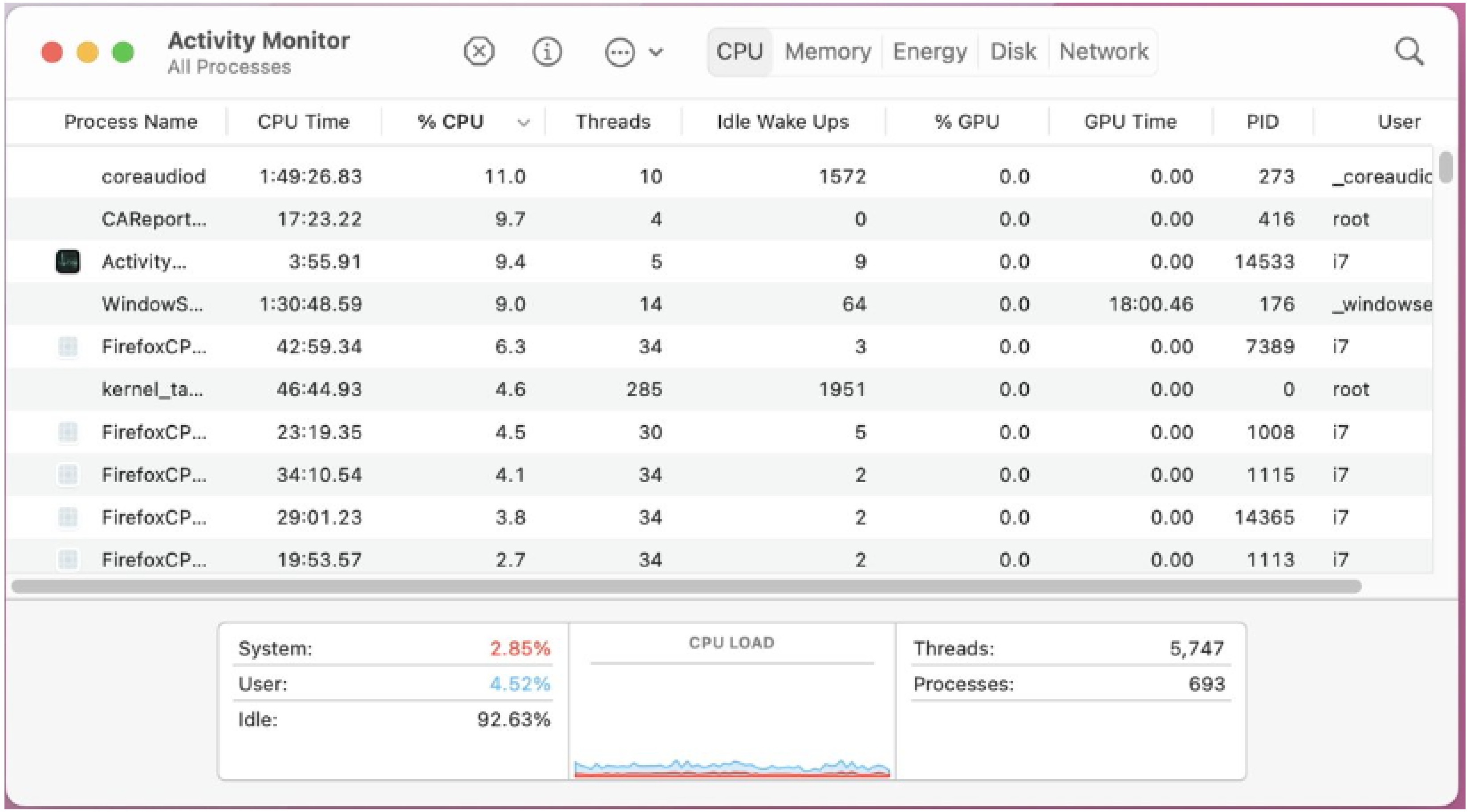
click

BOSTON UNIVERSITY

Process Monitoring – Mac OSX

- Use Spotlight (⌘ spacebar) to run the *Activity Monitor*

- 100% of CPU means 1 core is 100% occupied.
  - 200% means 2 cores are used, etc.



Process Name	CPU Time	% CPU	Threads	Idle Wake Ups	% GPU	GPU Time	PID	User
coreaudiod	1:49:26.83	11.0	10	1572	0.0	0.00	273	_coreaudic
CAReport...	17:23.22	9.7	4	0	0.0	0.00	416	root
Activity...	3:55.91	9.4	5	9	0.0	0.00	14533	i7
WindowS...	1:30:48.59	9.0	14	64	0.0	18:00.46	176	_windowse
FirefoxCP...	42:59.34	6.3	34	3	0.0	0.00	7389	i7
kernel_ta...	46:44.93	4.6	285	1951	0.0	0.00	0	root
FirefoxCP...	23:19.35	4.5	30	5	0.0	0.00	1008	i7
FirefoxCP...	34:10.54	4.1	34	2	0.0	0.00	1115	i7
FirefoxCP...	29:01.23	3.8	34	2	0.0	0.00	14365	i7
FirefoxCP...	19:53.57	2.7	34	2	0.0	0.00	1113	i7

System:	2.85%
User:	4.52%
Idle:	92.63%
Threads:	5,747
Processes:	693

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 →

BOSTON UNIVERSITY

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

Threads

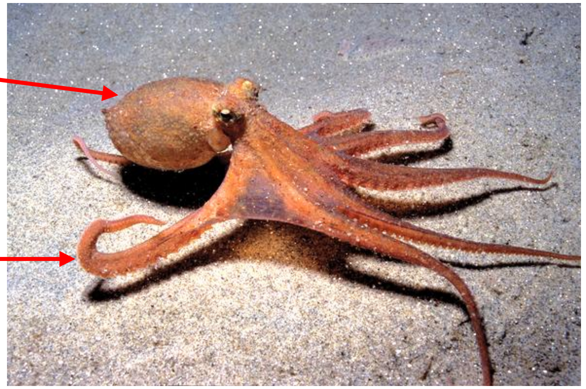
```
top - 10:45:13 up 45 days,  4:15, 109 users,  load average: 11.04, 5.48, 4.87
Tasks: 2753 total,   7 running, 2726 sleeping,   5 stopped,  15 zombie
%Cpu(s): 88.2 us,  2.4 sy,  0.0 ni,  8.3 id,  0.4 wa,  0.0 hi,  0.7 si,  0.0 st
KiB Mem : 26387792+total,  4700312 free, 76957904 used, 18221971+buff/cache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

- 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.

- Python running threads on 22 cores.
  - Note there is 1 Python process listed
  - 2186% means ~22 cores are busy.

One process

8 threads



BOSTON UNIVERSITY

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

Parallelize with Processes or Threads? Or both?

- Process Parallelism:
  - Can have a slow startup
    - Milliseconds to seconds to launch processes.
  - Data is usually be copied between processes.
    - Memory usage can be higher
  - Simpler to implement.
    - A serial function run in parallel via processes can often be used with few changes.
  - Can potentially execute on multiple computers and communicate via a network.
  - Avoids issues with libraries that are not compatible with threads.

- Thread-based:
  - Threads start up very fast (~dozen μs)
  - All the program memory is accessible by all threads.
    - Avoids the need to copy memory.
    - Lower system memory usage
    - Fast communication between threads by shared memory.
  - More complicated parallelization patterns can be implemented with less work.

You can add parallelism to your program through changing your source code or by calling libraries that implement parallel algorithms.

BOSTON UNIVERSITY

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 is an example of weak scaling.
  - 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

Language / Library	Parallelization	Notes
Python *multiprocessing* *joblib* *numba*	Processes Processes Threads	Standard language library popular library Python function → native code compiler
Matlab *parpool* *Implicit* parallelism	Processes Threads	Standard language library. Some operations will automatically multi-thread.
R *parallel* *foreach / doParallel* *future*	Threads Processes Both	Standard language libaries.
C++ *STL parallel* *TBB* *OpenMP*	Threads Threads Threads	C++17 standard and newer The Intel Thread Building Blocks library Standard threading library, comes with the compiler
Java *Thread* class *ForkJoinPool*	Threads Threads	Standard Java libraries.

BOSTON UNIVERSITY

Common Parallel Libraries

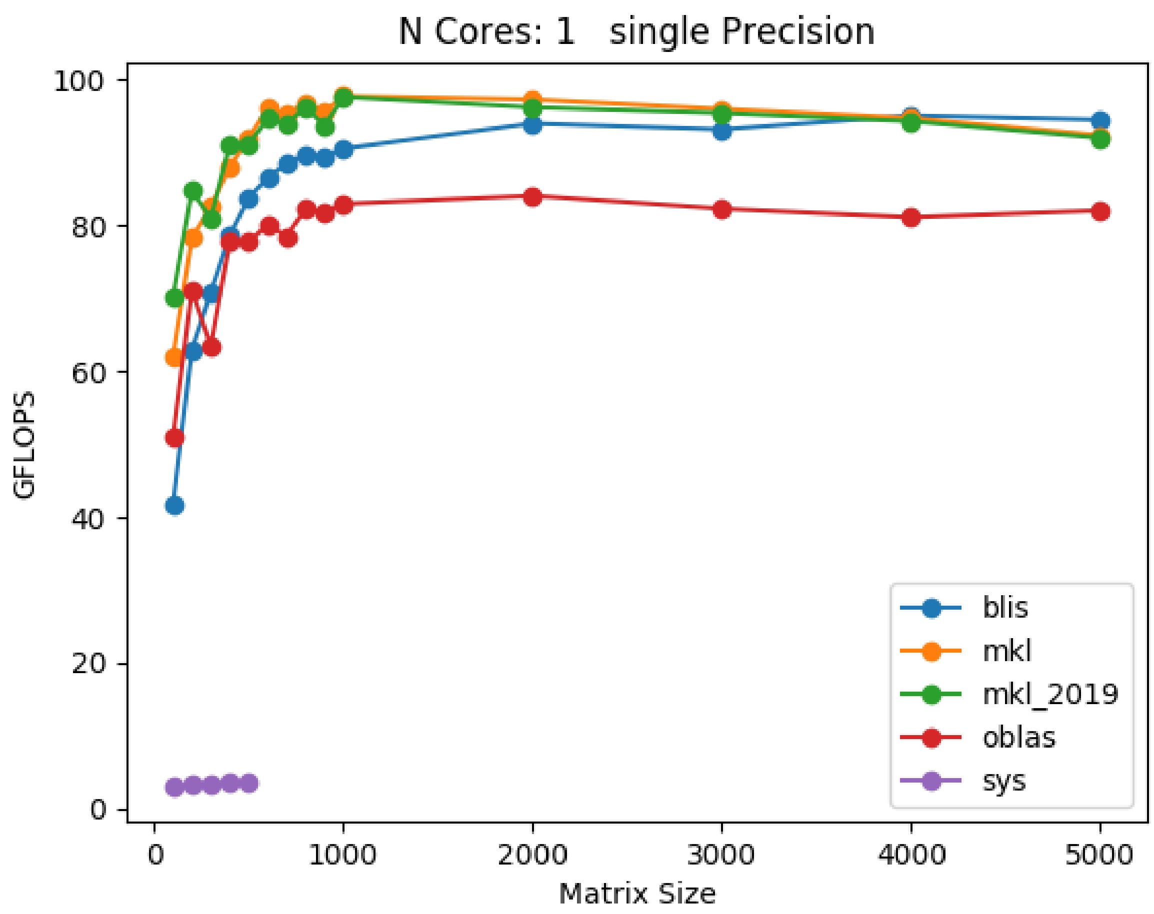
Library	Parallelization	Notes
BLAS & LAPACK (SCC: *blis* or *openblas* modules, MKL library in the *intel* module)	Threads	Linear algebra. **Widely** used, for example by R, Python, and Matlab.
FFTW	Threads	Fast Fourier Transforms.
OpenCV	Threads	Image processing.
PyTorch	Threads (CPU) or GPU	Machine 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.

BOSTON UNIVERSITY

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 OpenMP Threading Libraries on the SCC

- The most common multi-threading library in SCC modules is OpenMP.
  - Including the various BLAS libraries.
- 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 all jobs.

```
#$/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 assigned cores.

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

BOSTON UNIVERSITY

Don’t use more OpenMP threads than you have cores – performance will drop **drastically**.

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

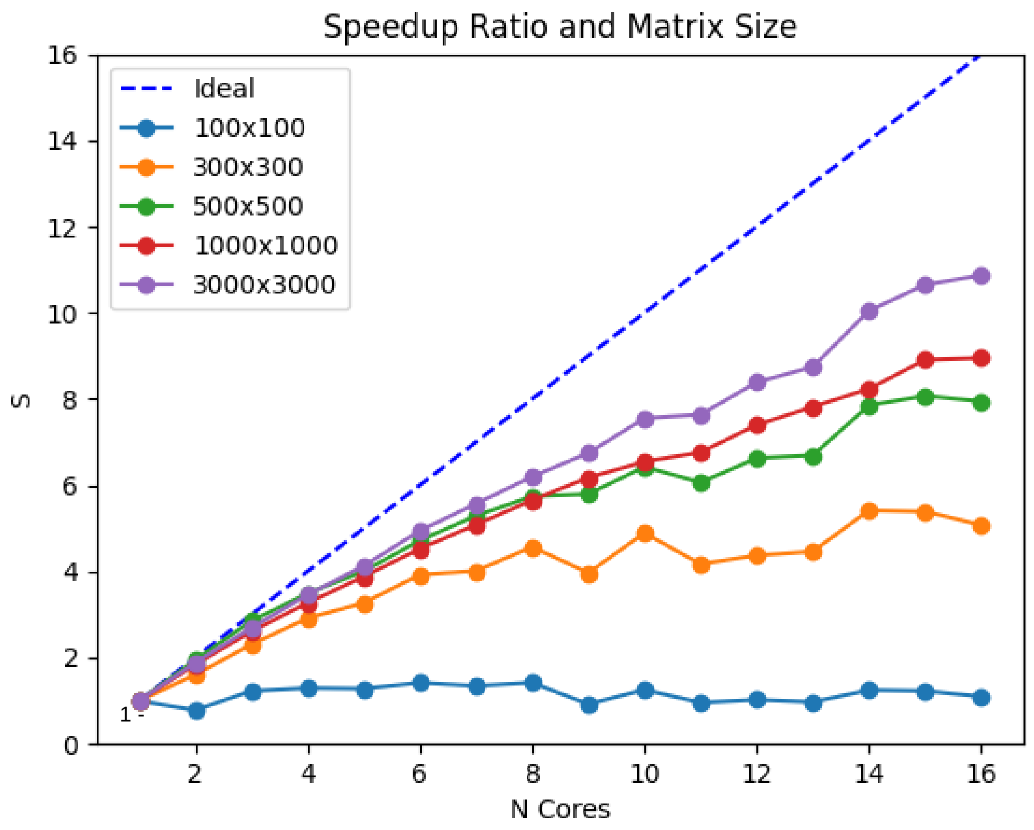
Enable OpenMP on non-SCC computers

- Environment variables can be set in various ways on different operating systems. Here is a guide for Windows, Linux, and Mac OSX.
- The OpenMP library looks for OMP_NUM_THREADS regardless of the operating system.
- Mac users:
  - The BLAS library used by R, Python, etc. is likely to be the *Apple Accelerate* library.
  - Try setting the variable **VECLIB_MAXIMUM_THREADS** along with OMP_NUM_THREADS.

Know your software

- OpenMP is hardly the last word in multithreading.
- Different software may have different mechanisms for enabling threaded or multiprocess calculations such as configuration options or command line flags.
- Read the documentation!

Strong scaling: Speedup Depends on the Problem

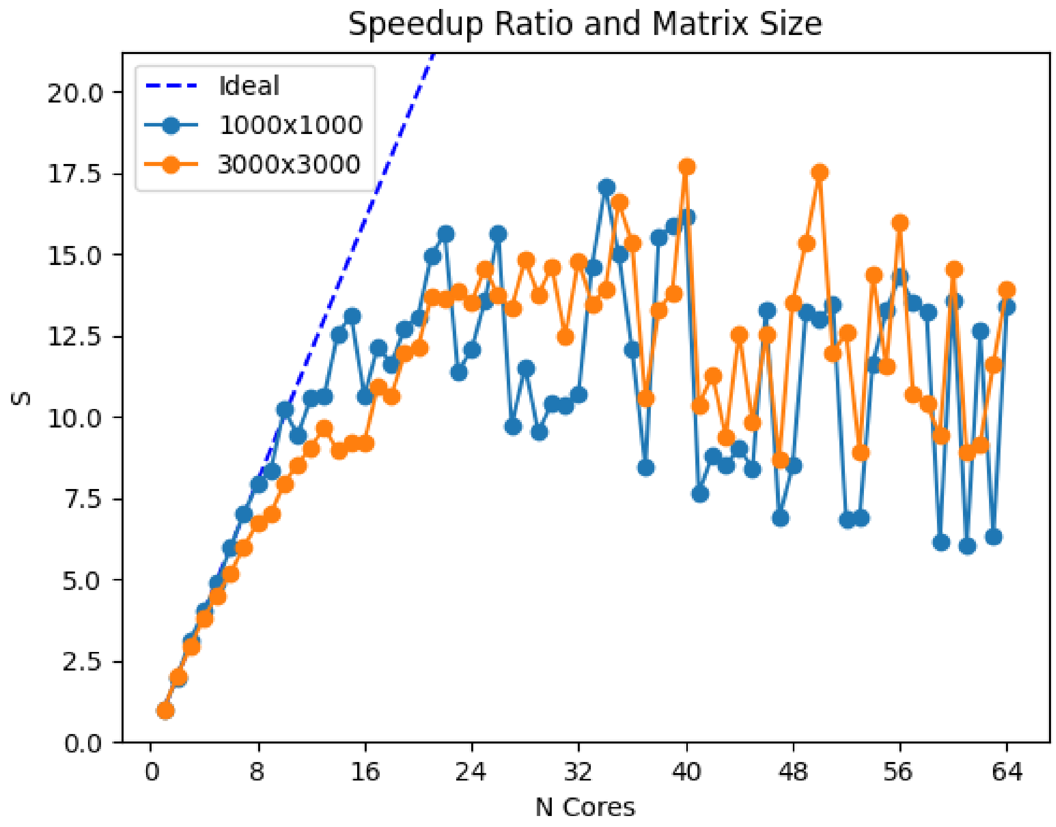


Intel Xeon CPU E5-2650 v2 @ 2.60GHz. 16 physical cores (scc-pi2)

- 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/processes doesn't help.
  - You have to test!

BOSTON UNIVERSITY

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



- Running on a 64-core system the computation actually gets slower with too many threads.
- It may be that some parts of your code benefit from more threads than others – try to pick a sensible number.
- The ideal thread number may change if you change the CPU manufacturer, CPU model, BLAS library, and so on.
  - Test your code!

BOSTON UNIVERSITY

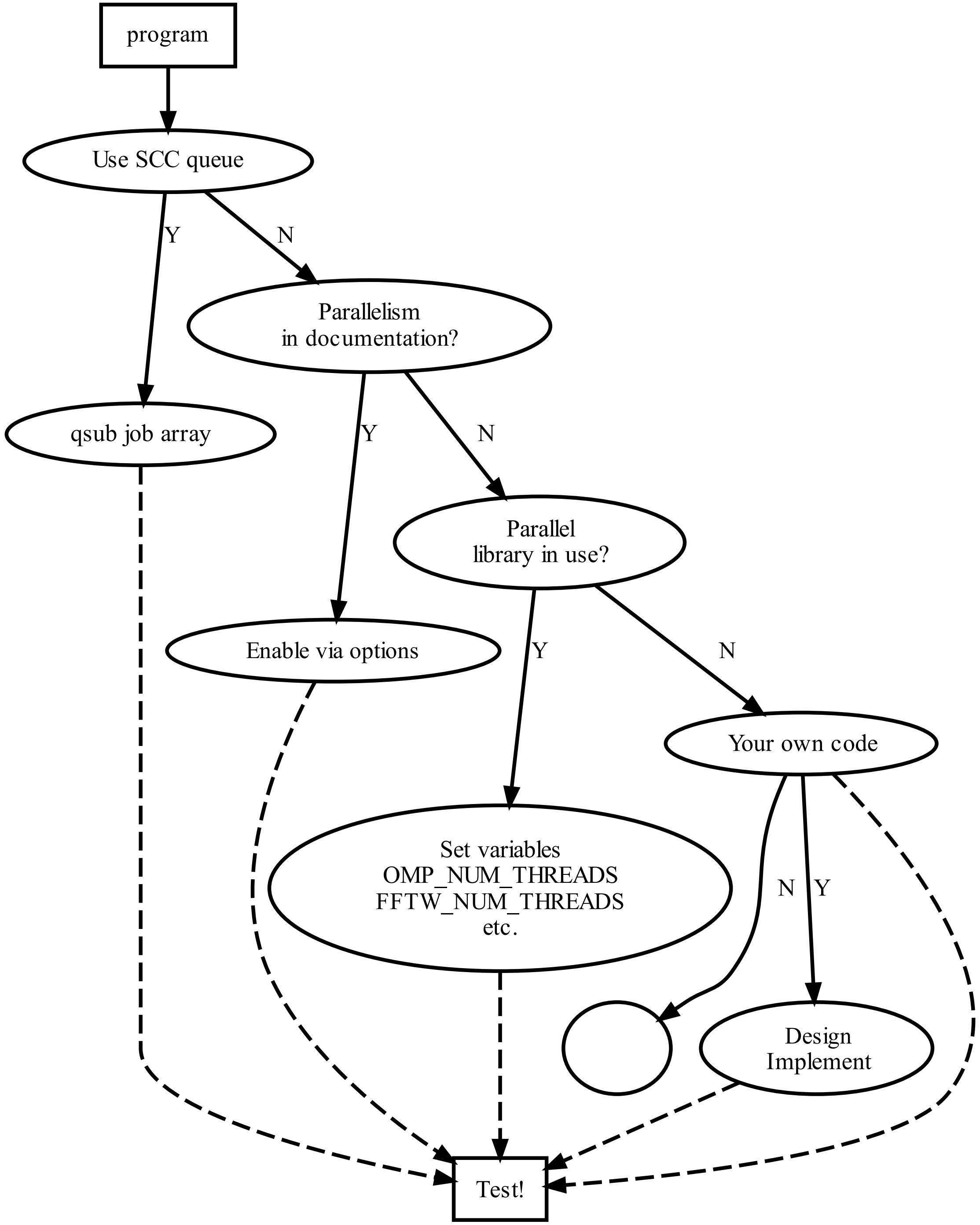
Outline

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

Parallelizing your own code

- Here's a chart to help you bring all this together in the context of your code.
- Is it your own program (you have the source code) or are you using someone else's program?
  - Always: read the documentation.

.



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/2021.1` module)

BOSTON UNIVERSITY

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 itself.
  - Try `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 parallelization.

BOSTON UNIVERSITY

Use the source

- If you have the source code you have much more control

- Look for language options for implicit/automatic multithreading:
  - Matlab: `maxNumCompThreads(N)`
  - R (for Rcpp code): `setNumThreads(N)`
  - Mathematica: `LaunchKernels[N]`

- Incorporate parallel libraries
  - Ex. Python: switch from Pandas DataFrames to Dask DataFrames or Polars

BOSTON UNIVERSITY

Modify your code

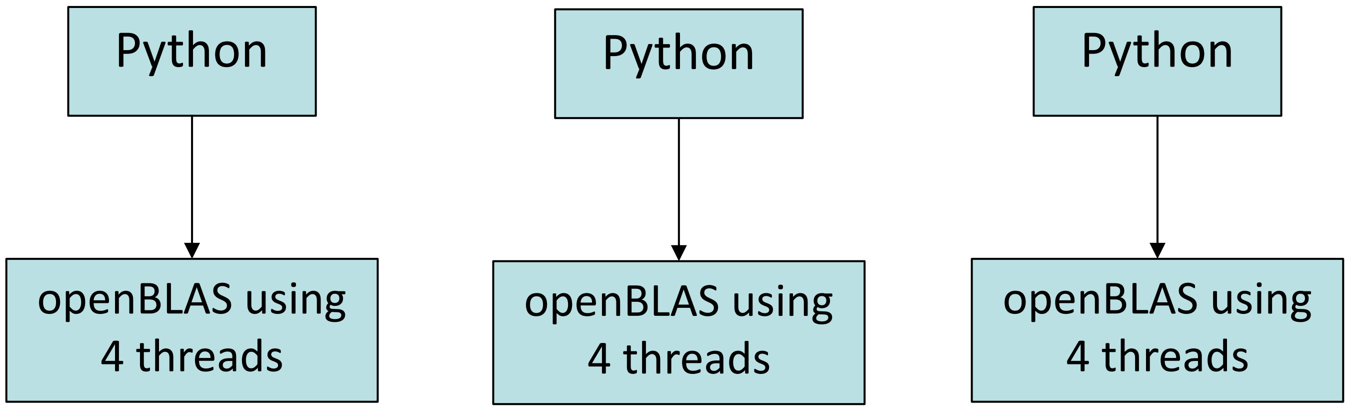
- Make use of parallel capabilities built into the language when you can:
  - Matlab: `parfor`
  - R: `parallel:mclappy, foreach, doParallel`
  - Python: `multiprocessing.Pool, joblib`
  - C++ (C++17 standard and up) parallel `STL` algorithms

- More extensive or elaborate parallelization might require using an additional programming language with libraries like OpenMP
  - R → Rcpp (C++)
  - Python → use Numba, or Cython, or call out to C, C++, or 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

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 using *numpy*
- What's the most number of cores that get used at the same time?
- 3 processes * 4 threads per process = 12

Parallelization Difficulties

- Some code cannot be parallelized – it must be computed in order.
  - Ex.: random number generation can be tricky
- 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.

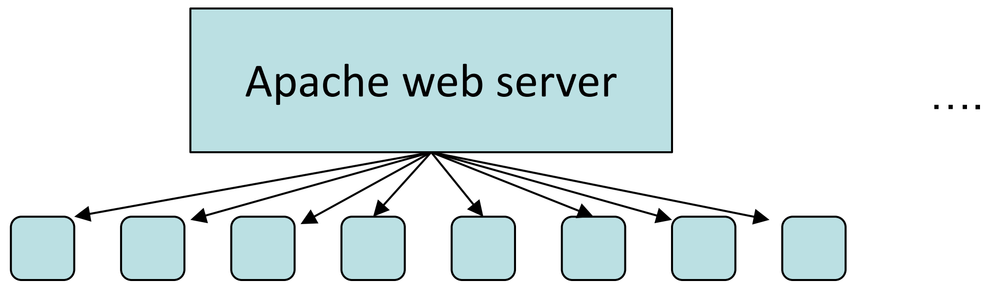
- Multiple process parallelization can consume large amounts of memory.

How Many Workers*?

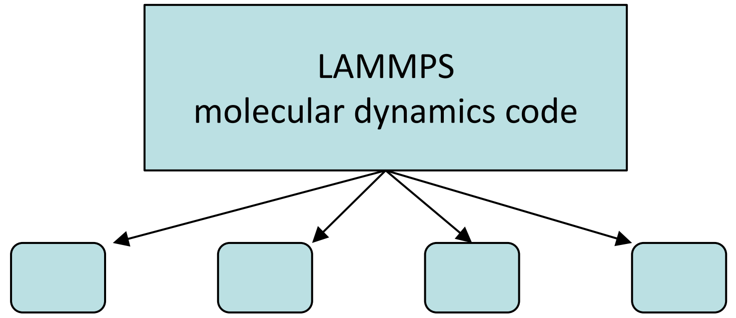
- I/O-bound programs may run hundreds or thousands of workers
  - These spend a lot of time **waiting** for data from the network, the disk, the user, etc.

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

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



Hundreds of copies of itself handle incoming web traffic



4 cores – 4 workers

BOSTON UNIVERSITY

* worker: process or thread

What happens with too many workers?

- For CPU-bound problems, use no more than 1 worker per physical core.
- 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.

Appendix

- Some extra slides

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`**

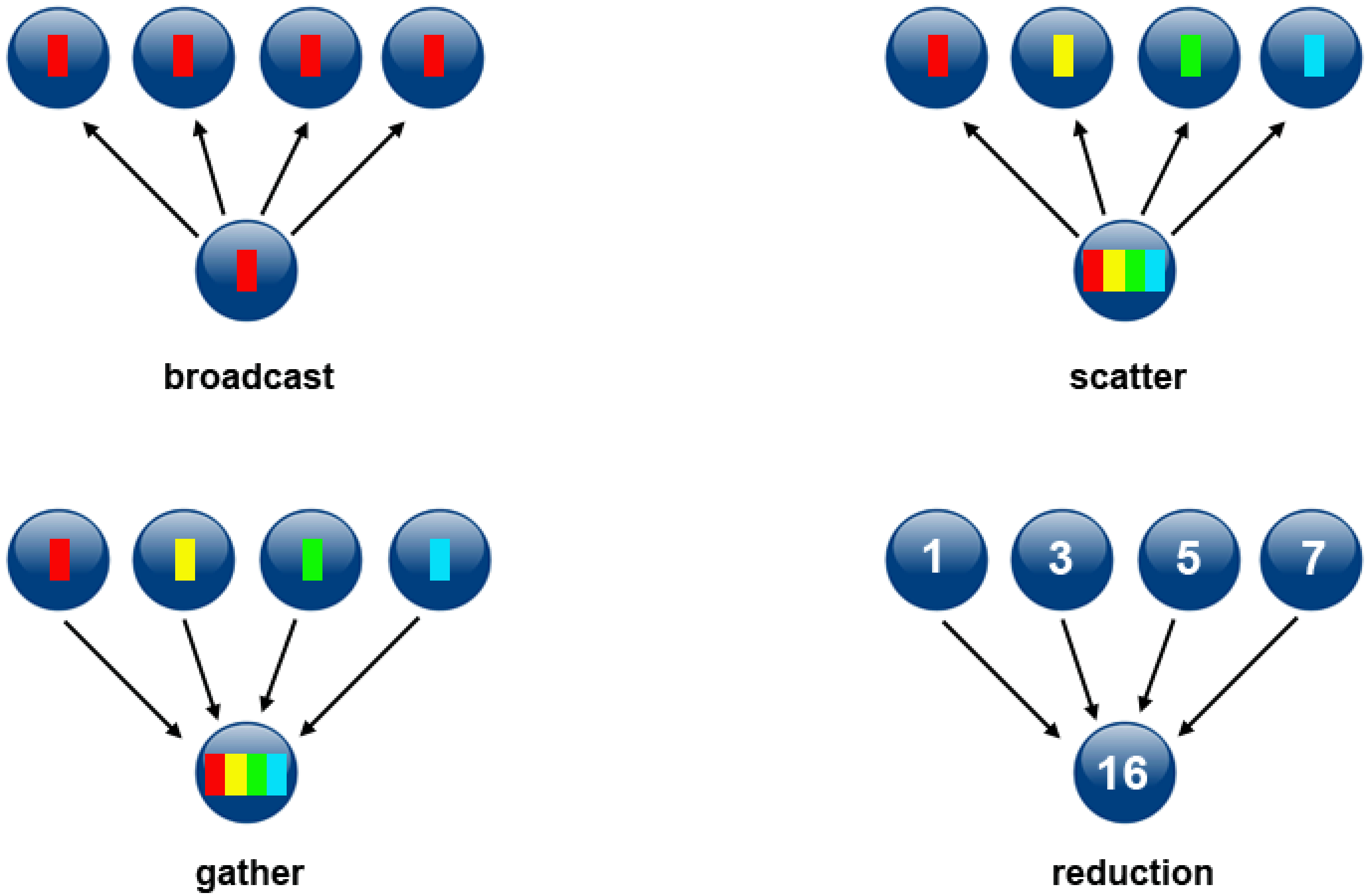
- Used on the world's largest clusters with thousands of cores over hundreds of compute nodes for single programs.

BOSTON UNIVERSITY

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.



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.

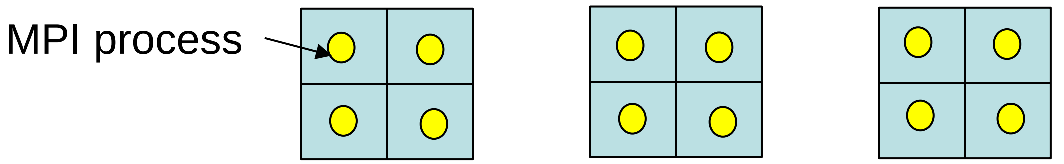
Language	Library
Python	mpi4py
R	Rmpi
Julia	MPI.jl
C#	MPI.NET

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

mpirun

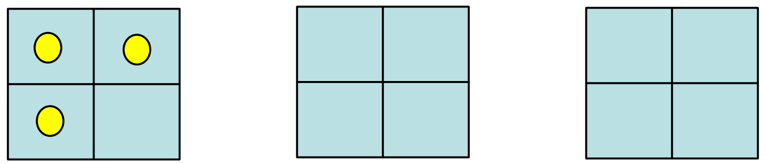
- MPI programs have a special program to launch them, *mpirun*
- OpenMPI's *mpirun* has many options that control how MPI processes are started and where they run.
  - Try *module help modulename* on the SCC for MPI-based modules
- On the SCC the configuration of compute nodes for *mpirun* is handled by the queue.

3 compute nodes, 4 cores each.



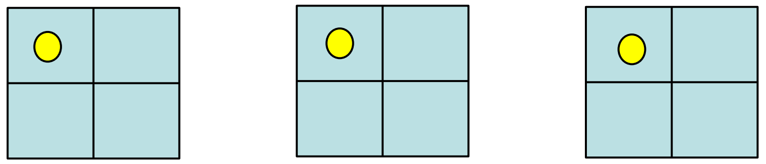

```
mpirun -np 12 my_mpi_prog
```

1 MPI process per compute node will run.



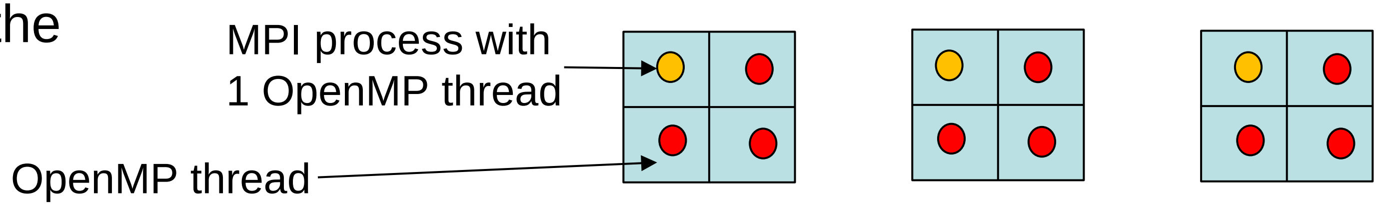
```
mpirun -np 3 my_mpi_prog
```

3 MPI processes will run…all on node 0.



```
mpirun -np 3 --map-by ppr:1:node my_mpi_prog
```

3 MPI processes will run, one per node




```
export OMP_NUM_THREADS=4
mpirun -np 3 --map-by ppr:1:node my_mpi_prog
```

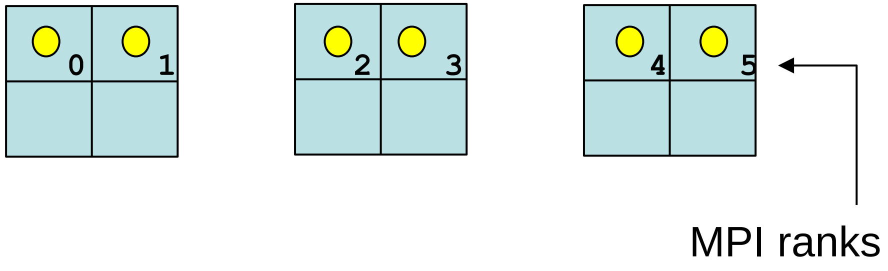
3 MPI processes will run, one per node, with 4 threads

BOSTON UNIVERSITY

3 compute nodes, 4 cores each.

mpirun process assignment

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



- 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.

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

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

- Request MPI-specific nodes on the SCC with the qsub option:
  - -pe mpi_16_tasks_per_node N
    - Where N is a multiple of 16
    - N=48 → 4 16-core nodes
    - NSLOTS → 48
  - -pe mpi_28_tasks_per_node M
    - Where M is a multiple of 28
- The only way to use multiple compute nodes for a job on the SCC is to use the MPI queues.

- These jobs run on dedicated compute nodes connected with an *Infiniband* network.
  - See above for SCC versions
- Latency is how quickly a data transfer can be initiated. For MPI computations this is often the limit, not the bandwidth.

BOSTON UNIVERSITY