Introduction to Parallel Computing

0.1

Spring 2021

Research Computing Services

IS & T

BOSTON UNIVERSITY

Outline

- Parallel Examples
- Parallel Strategies
- Hardware
- Processes and threads
- Libraries & your own code
- Parallelization 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.
  - Either directly by implementing parallel code
  - Or indirectly by using libraries that perform parallel calculations.

- First, let's look at an example of some problems that could be solved with parallel computations.

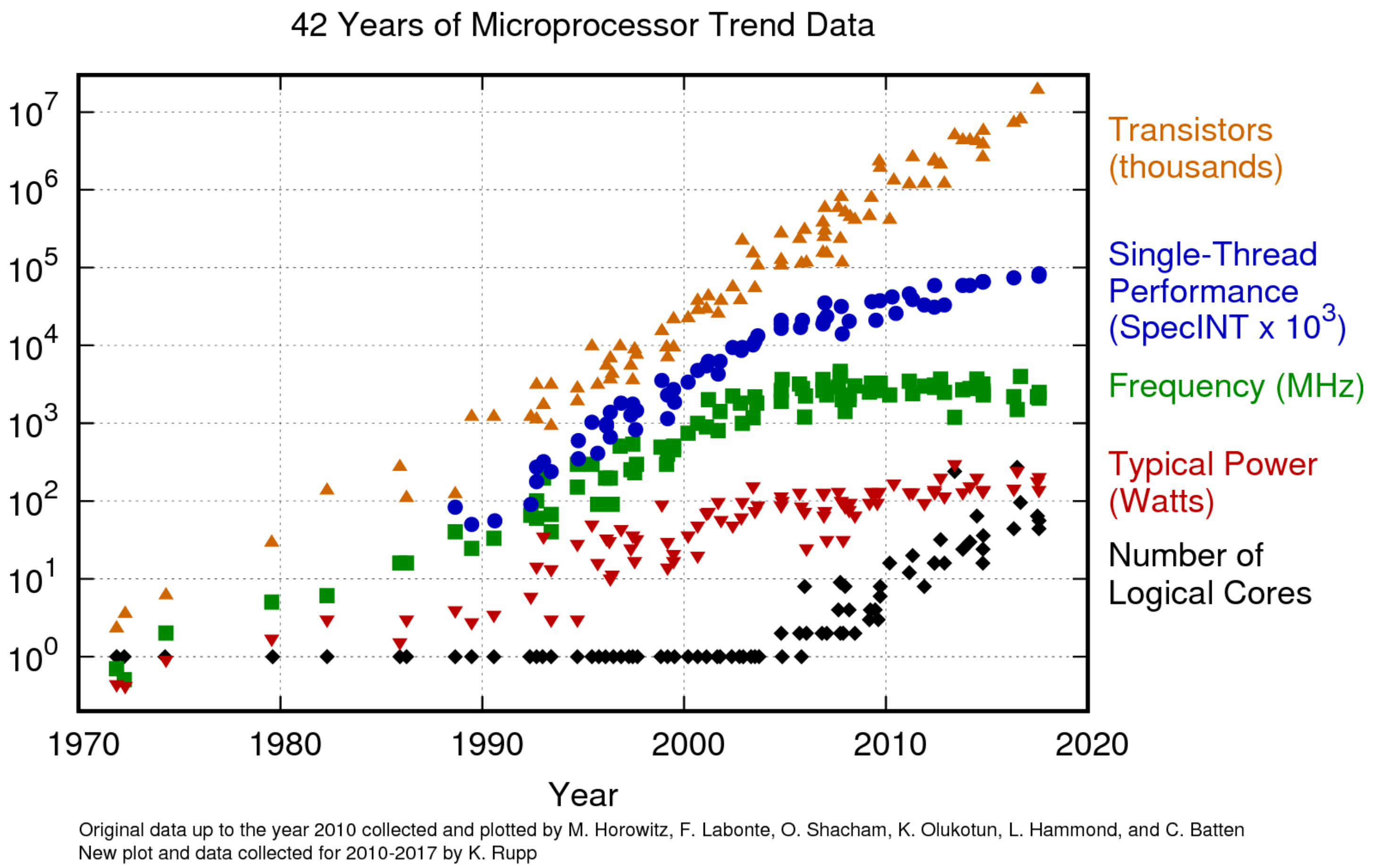
Limits (“bounds”) on Program Speed

- **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: computer has 16 GB of RAM, data file is 64 GB in size.

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

Why Parallelize?

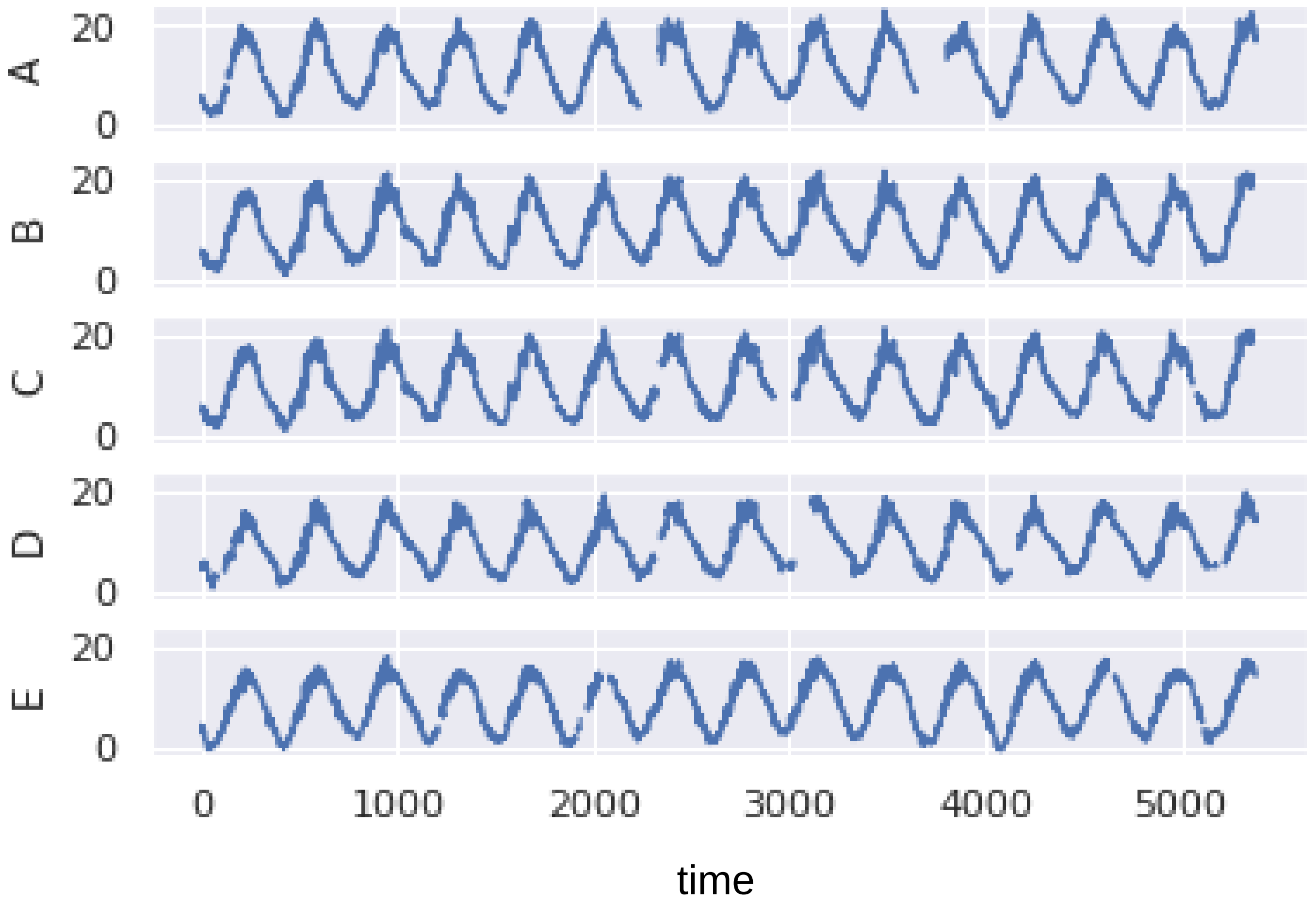


https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

BOSTON UNIVERSITY

Example 1: Buoy data

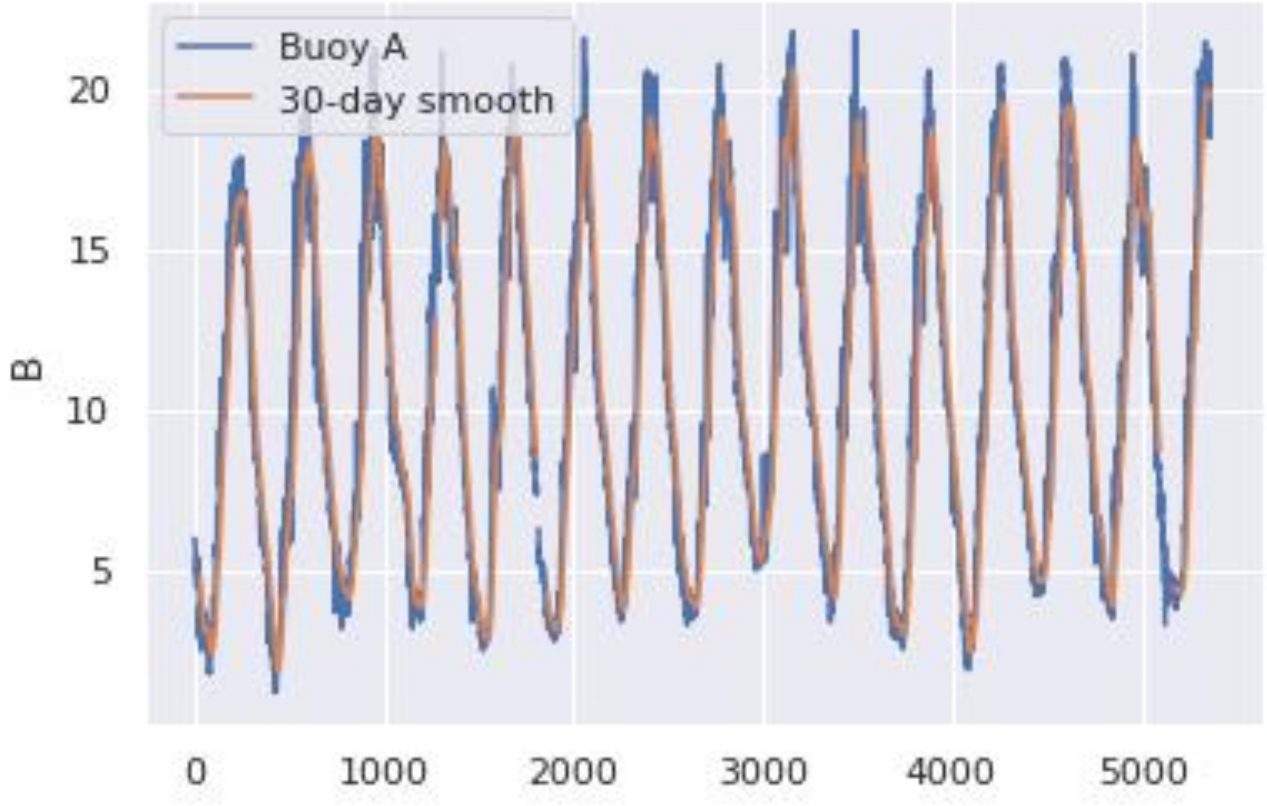
- Each row is the water temperature in Celcius at a depth of 2m from different ocean buoys off of the coast of New England.
- We want to find the average daily temperature across all the buoys.
  - i.e. sum them all vertically and divide by 5.
- How can we parallelize this?



BOSTON UNIVERSITY

Example 2: Buoy data

- Now consider the temperatures from 1 buoy. This is daily data. How can we smooth the data by 30 days?
- Smoothing:
  - Pick 30 days worth of data points starting at day 0. Average them and assign that value to a new array at day 0.
  - Pick another 30 days of data starting from day 1. Assign that average value to day 1.
  - Etc.
- How can we parallelize this?



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.

```
AGTCCCCGTCTTGCCGCGCGGGGGCGGGCGCGGGAAAAAAGCCGCGCGGGGGCGC
CCGCGGGAAGGCAGCCCCGCGGCGCGCGGGGGGAGGGGCGGCGCCCGCGGGGGAG
CGGCCGGCTCCGGGGGAGGGACGGGGAAGGGGGCGCGCGGGGCTGCCCTGCCGCC
CGCCCGCCGCCGCCGCCCGCCTTCGCGCCCCCCCCCAAAAAACACCCCCCCCGGA
```

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

BOSTON UNIVERSITY

Example 3: *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 concurrently?

BOSTON UNIVERSITY

Outline

- Parallel Examples
- Parallel Strategies
- Hardware
- Processes and threads
- Libraries & your own code
- Parallelization 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 Python 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

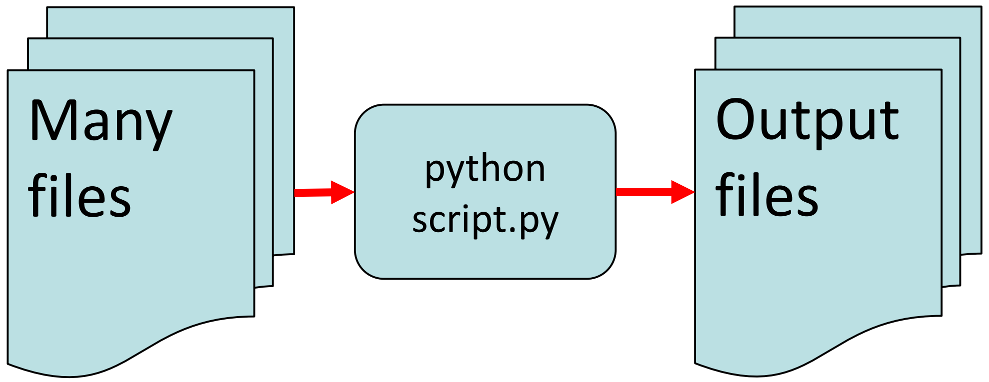
- This can easily be computed in parallel. Break into 2 chunks, sum them, and sum the chunks:
  - Or break it down into even smaller computaitons.

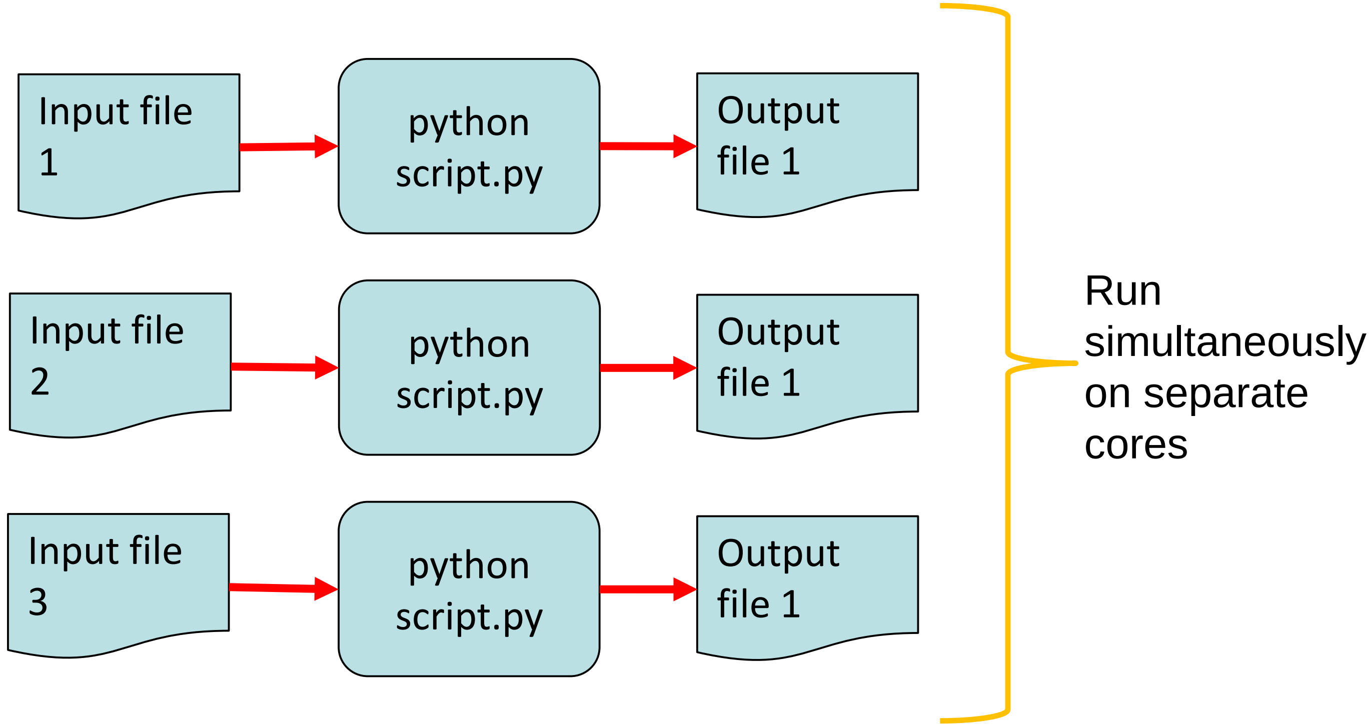
1+2+3+4+5 + 6+7+8+9+10

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.

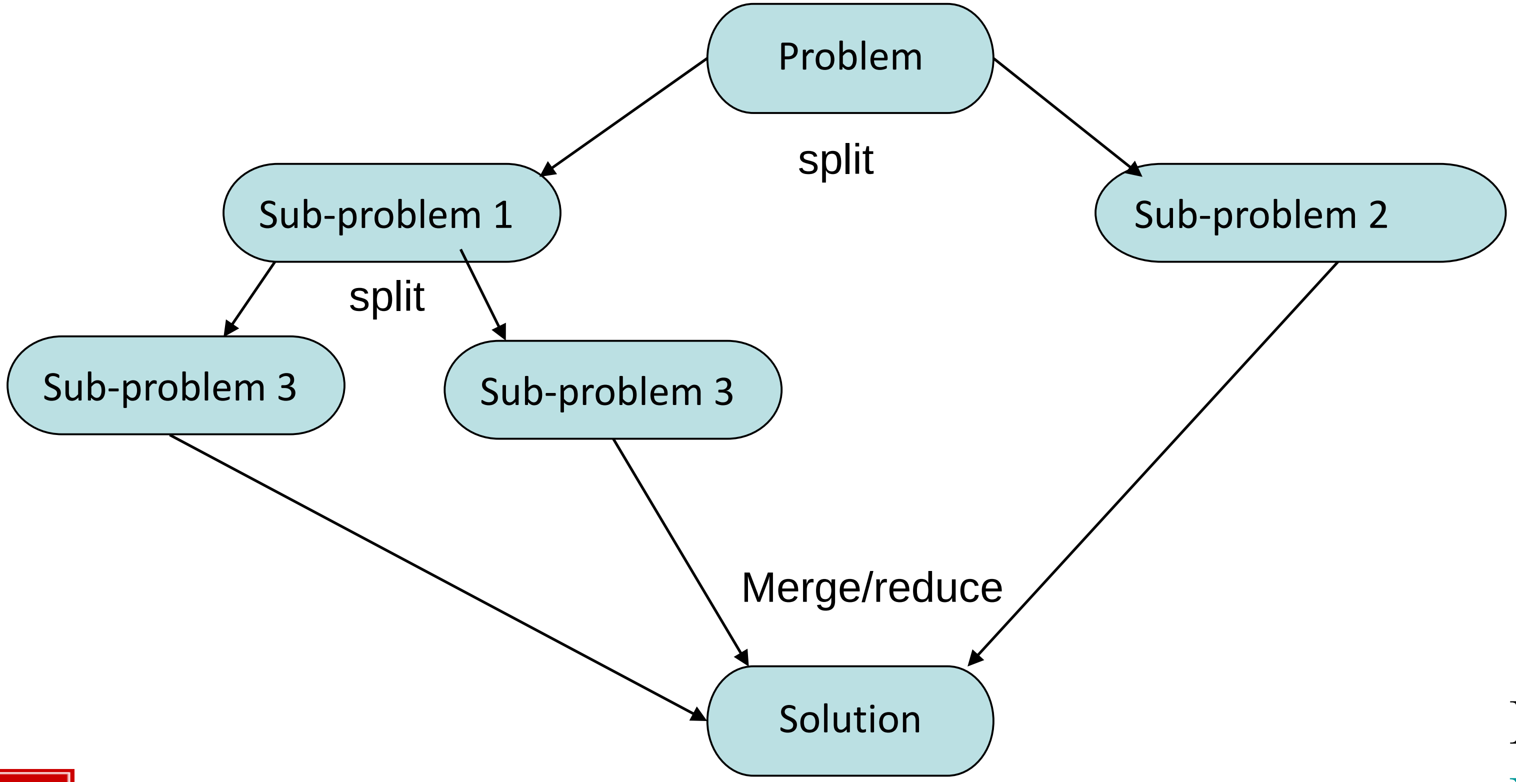
```
x = [1,2,3,4,5]

% Each loop iteration has no dependence
% on any other loop iteration.
for i = 1:5
      x(i) = some_func(x(i))
```

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.

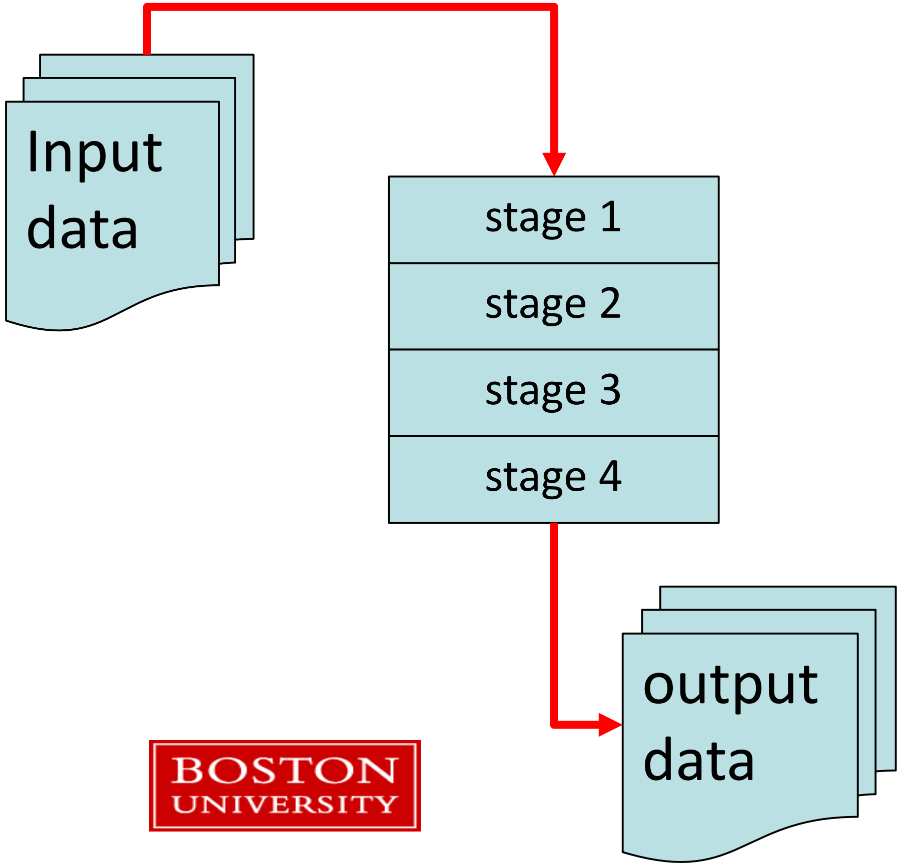
Or both 3's with 2.

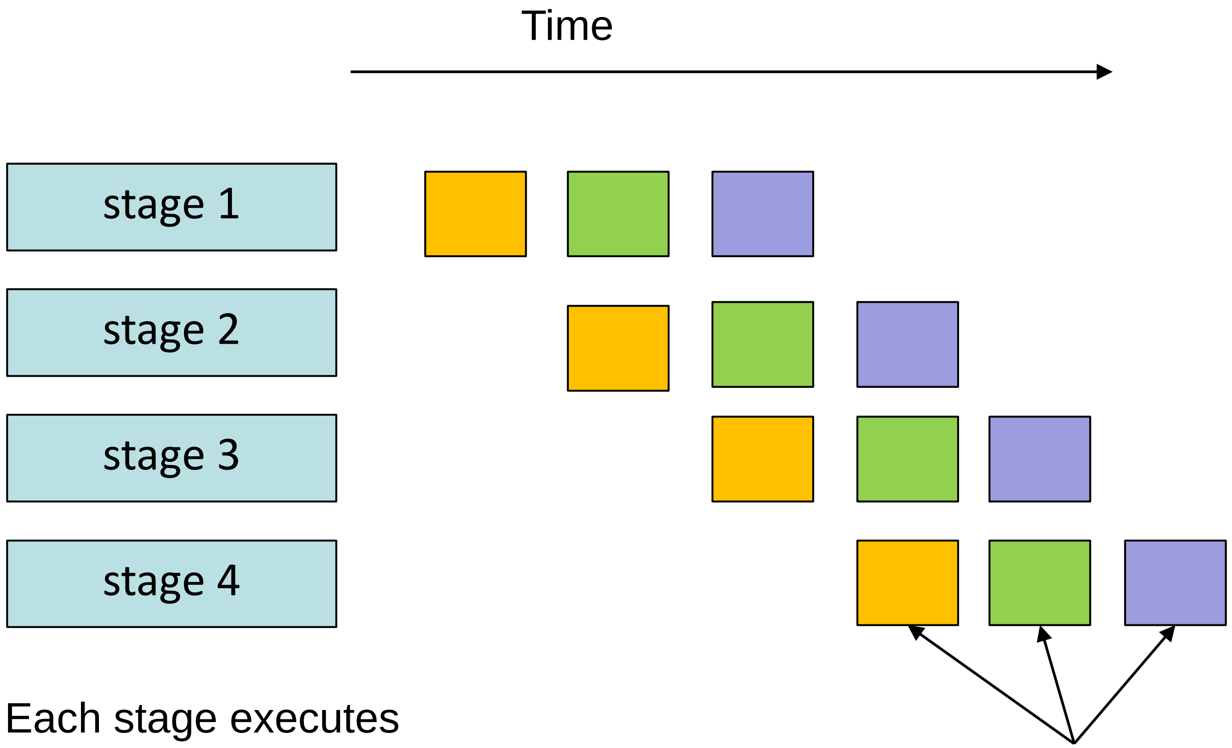
Example: the famous MapReduce algorithm.

BOSTON UNIVERSITY

Pipeline

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





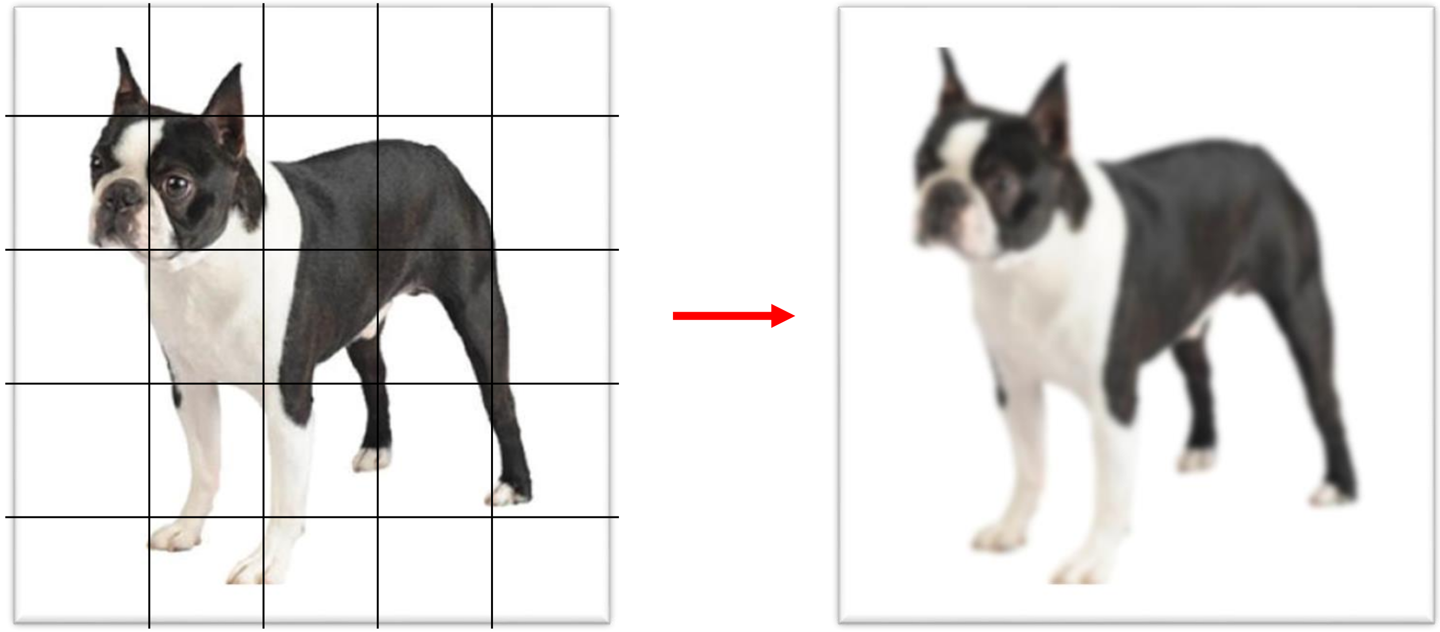
Each stage executes in parallel.

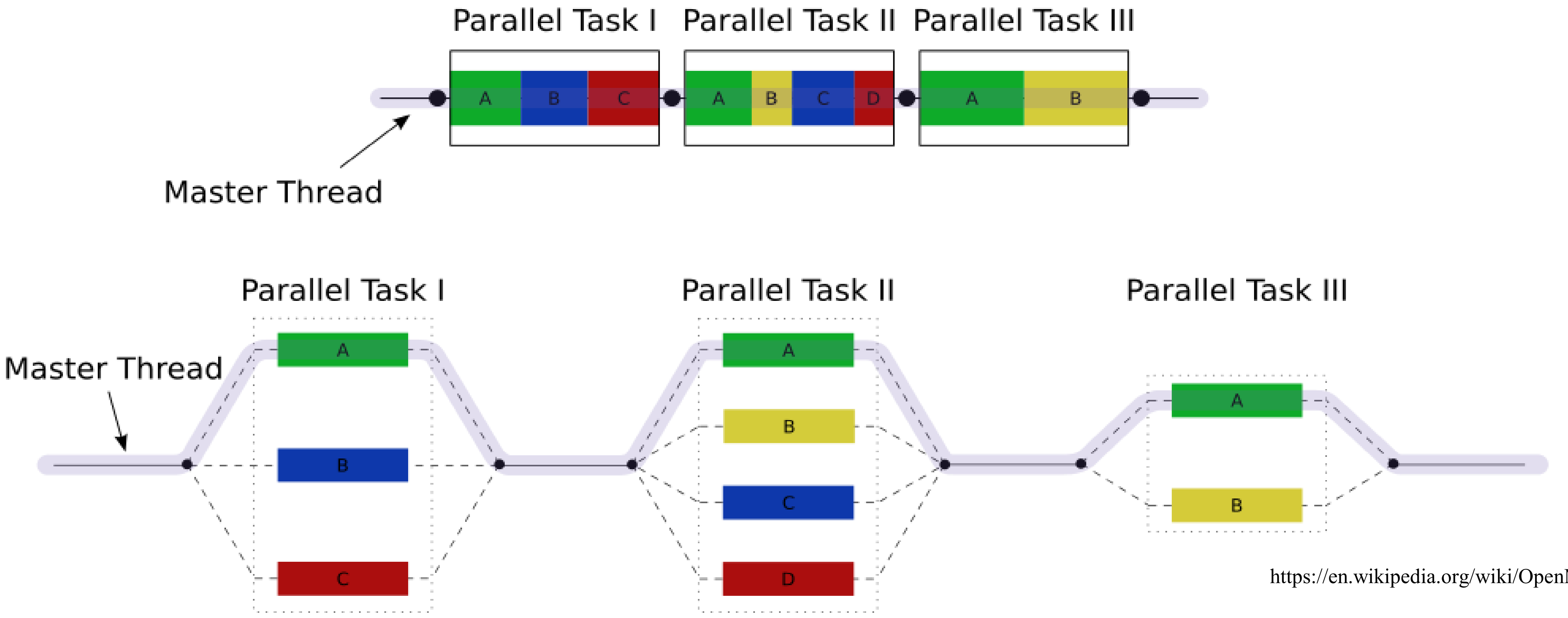
Chunks of input data

BOSTON UNIVERSITY

Geometric

- The problem can be broken up into predictable patterns.
- Example: blurring an image – the image can be broken into overlapping tiles processed in parallel.





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

- Different parts of a program may use *different parallel strategies* during execution.

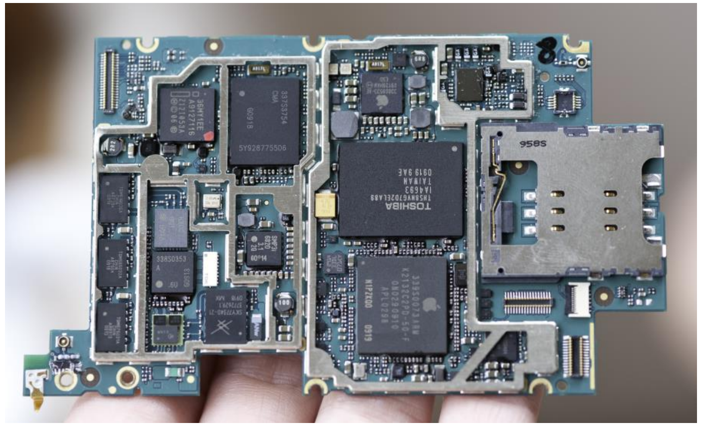
Outline

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

BOSTON UNIVERSITY

Hardware for Parallel Computation

Lenovo ThinkSystem HPC cluster



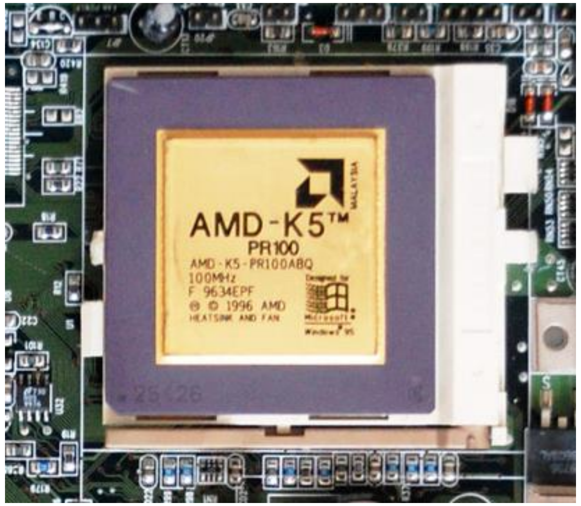
iPhone motherboard

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

BOSTON UNIVERSITY

CPUs and cores

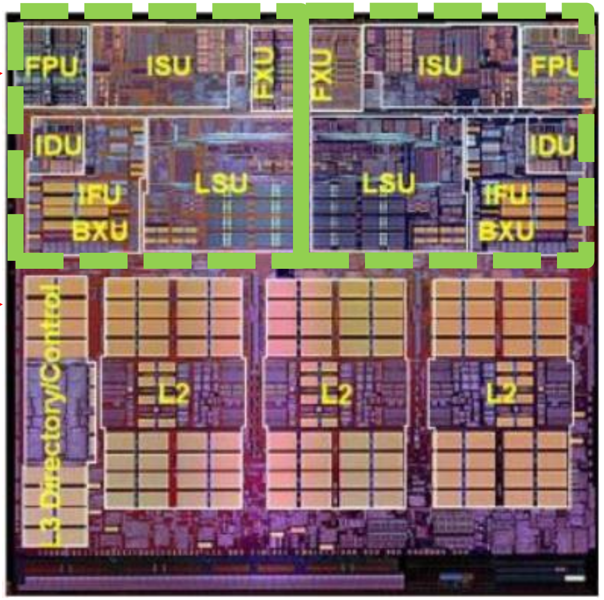
- In the beginning...a CPU plugged into a socket in the computer.
  - The term “core” wasn’t in use but we’d call this a 1-core CPU today.
  - Multiple CPU computers had multiple CPU sockets.



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

core



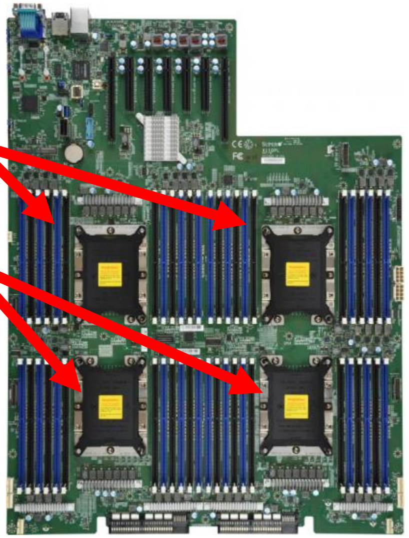
core

POWER4 circuit view

BOSTON UNIVERSITY

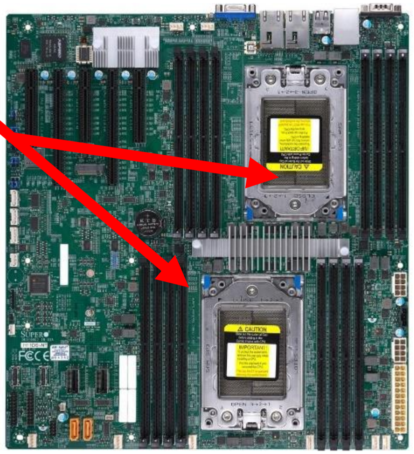
Modern configurations

- Quad Intel Xeon CPUs
- Up to 28 cores per CPU



- Single Intel CPU
- 4 cores (Core-i3, ~$100)

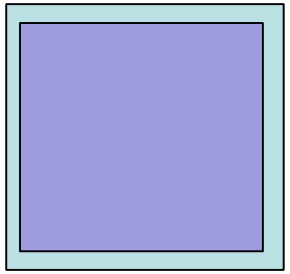
- Dual AMD Epyc CPUs
- Up to 64 cores per CPU



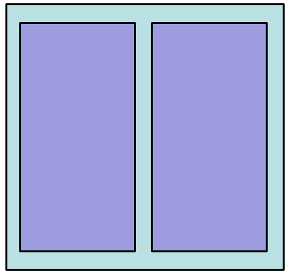
- For PC and server hardware the high end has very high core counts.
- Entry-level systems still have multiple cores.
- Parallel capabilities are everywhere these days.

BOSTON UNIVERSITY

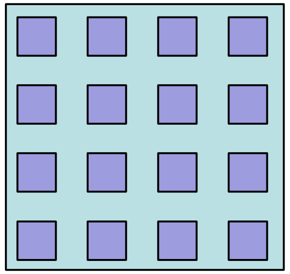
CPUs and cores



1 CPU, 1 core
1 program at a time



1 CPU, 2 cores
2 programs simultaneously



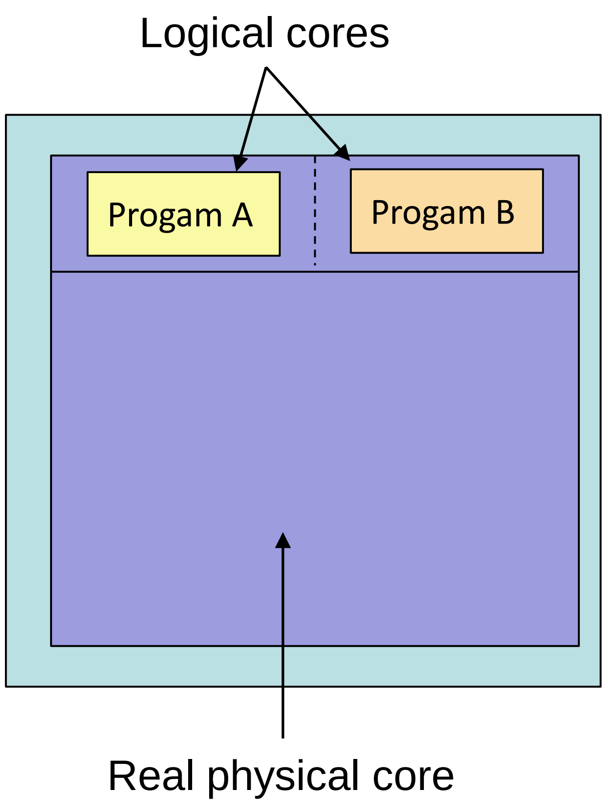
1 CPU, 16 cores
16 programs simultaneously

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

Hyperthreading (Intel trademark)

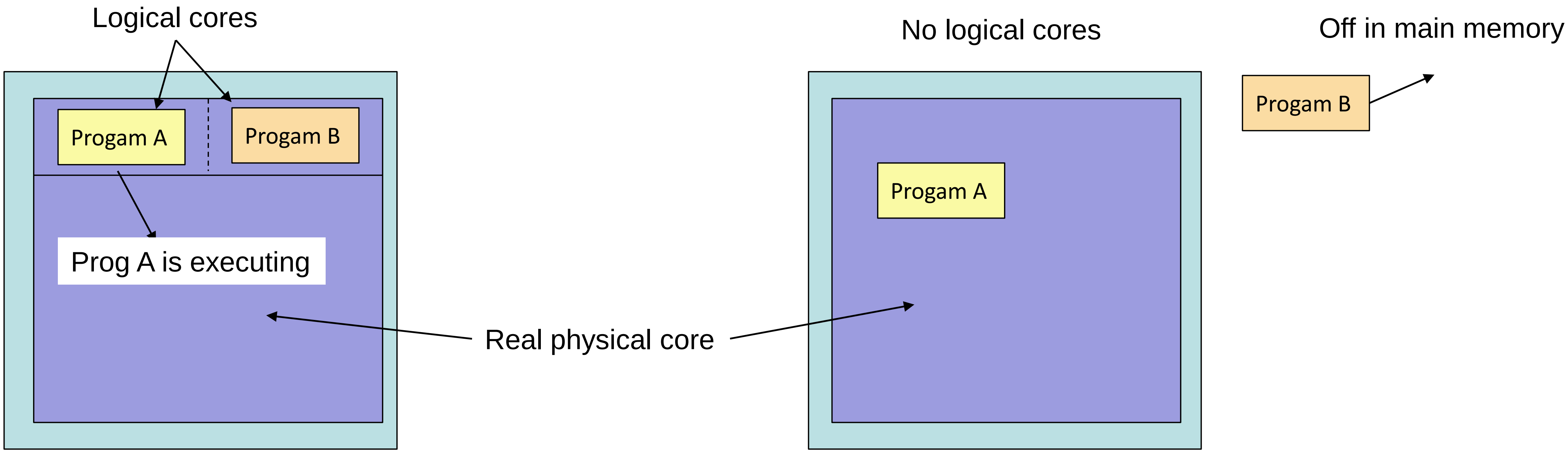
“Logical Cores” or “hardware threads”

- CPUs with this feature have some additional hardware that lets a program have its execution state pre-loaded onto a core while another program is actually executing on that core.
    - The hardware allows the OS to switch the physical core to run the other program very quickly.
- For many sets of programs (especially I/O bound) this makes better use of the physical core.



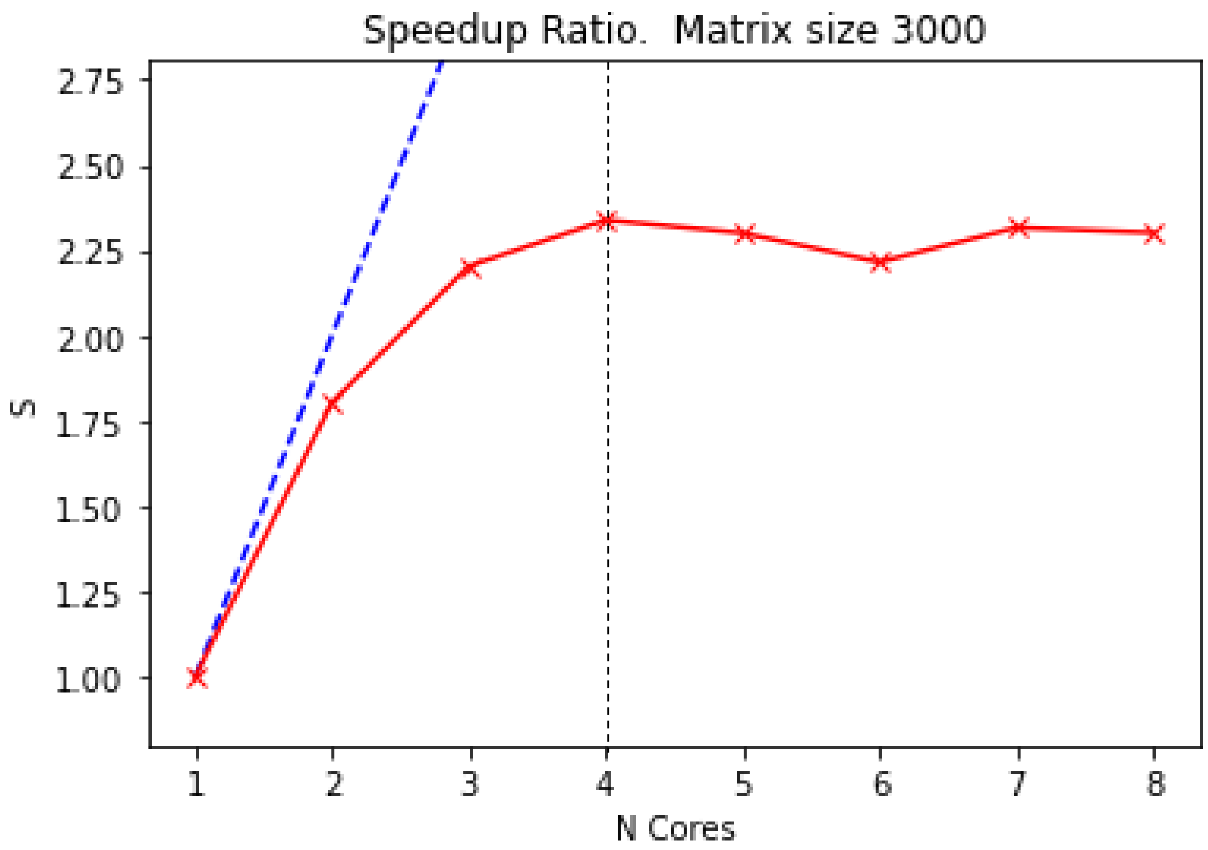
- Intel claims overall system performance can be 30% better.

BOSTON UNIVERSITY



- For regular CPUs the program switching is slower.
- For parallel CPU or compute bound programs the “extra cores” are of no benefit and typically **degrade** overall system performance.
- Hyperthreading or logical cores does **not** double the computational resources.

Hyperthreaded Intel i5-9300h CPU

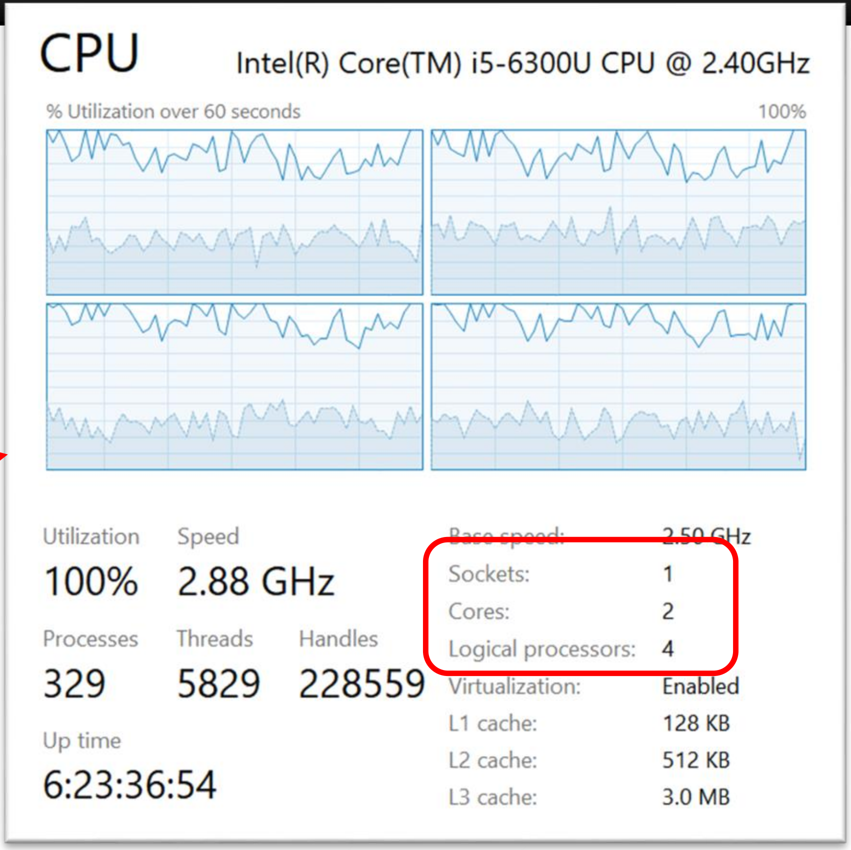


- A linear algebra matrix-matrix multiply.
- 4 real cores, 4 logical cores.
- Note performance increases stop for cores > 4.
- CPU-bound programs can only take advantage of **real** cores.

BOSTON UNIVERSITY

Count Your Cores

- Operating system utilities are the easiest way.
- Windows Task Manager



- 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
```

BOSTON UNIVERSITY

Final Comments

- On your personal or lab computers, check to see if logical cores are present.
- If so, beware of using all of them for a parallel computation.
  - It's best to use just the physical cores if CPU-bound.

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

Outline

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

BOSTON UNIVERSITY

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.

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
```

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

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

Process Scheduling by Analogy

- Consider this 4-burner stove.

- There are 17 pots of soup to cook.
  - Some must be kept very hot (lots of time on a burner)

- The chef swaps pots frequently to share the burners as fairly as possible.

BOSTON UNIVERSITY

Process Scheduling by Analogy

- Consider this 4-burner stove.
- There are 17 pots of soup to cook.
  - Some must be kept very hot (lots of time on a burner)
- The chef swaps pots frequently to share the burners as fairly as possible.

- Consider a 4-core CPU
- There are 17 processes to run
  - Some require a lot of CPU time
- The OS swaps processes on and off the cores to share computation time as fairly as possible.

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 each other and the original process.
  - Does not have a separate process ID.
  - Can exchange data with other threads or with other processes.

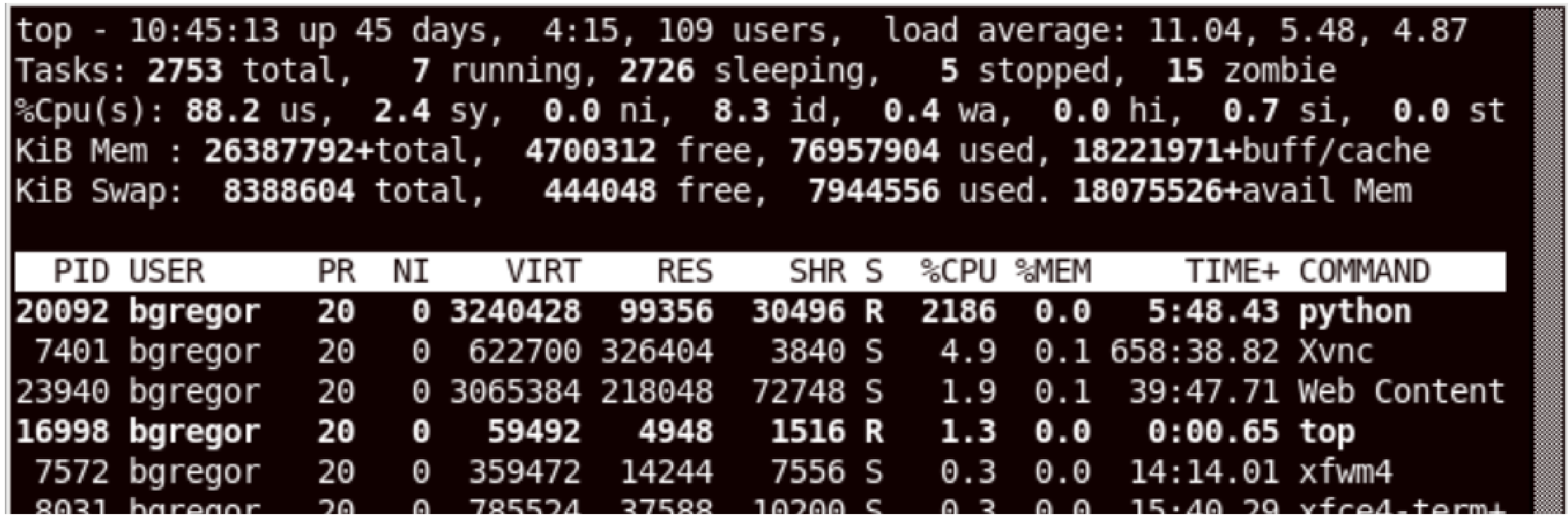
- Python running threads on 22 cores.
  - In the *top* program 100% of CPU means 1 core is 100% busy.
  - 2186% means 22 cores are busy.

- The cooking analogy equivalent would be a large pot that covered several burners.

BOSTON UNIVERSITY

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

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
```

- On the SCC, use *top*
- To see your processes only: top -u username

- 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

Parallelize with Processes or Threads?

- 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 run 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.
  - Higher performance intra-thread communication compared with processes.
  - More complicated parallelization patterns can be implemented.
    - Easy to start & stop threads.

BOSTON UNIVERSITY

Outline

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

BOSTON UNIVERSITY

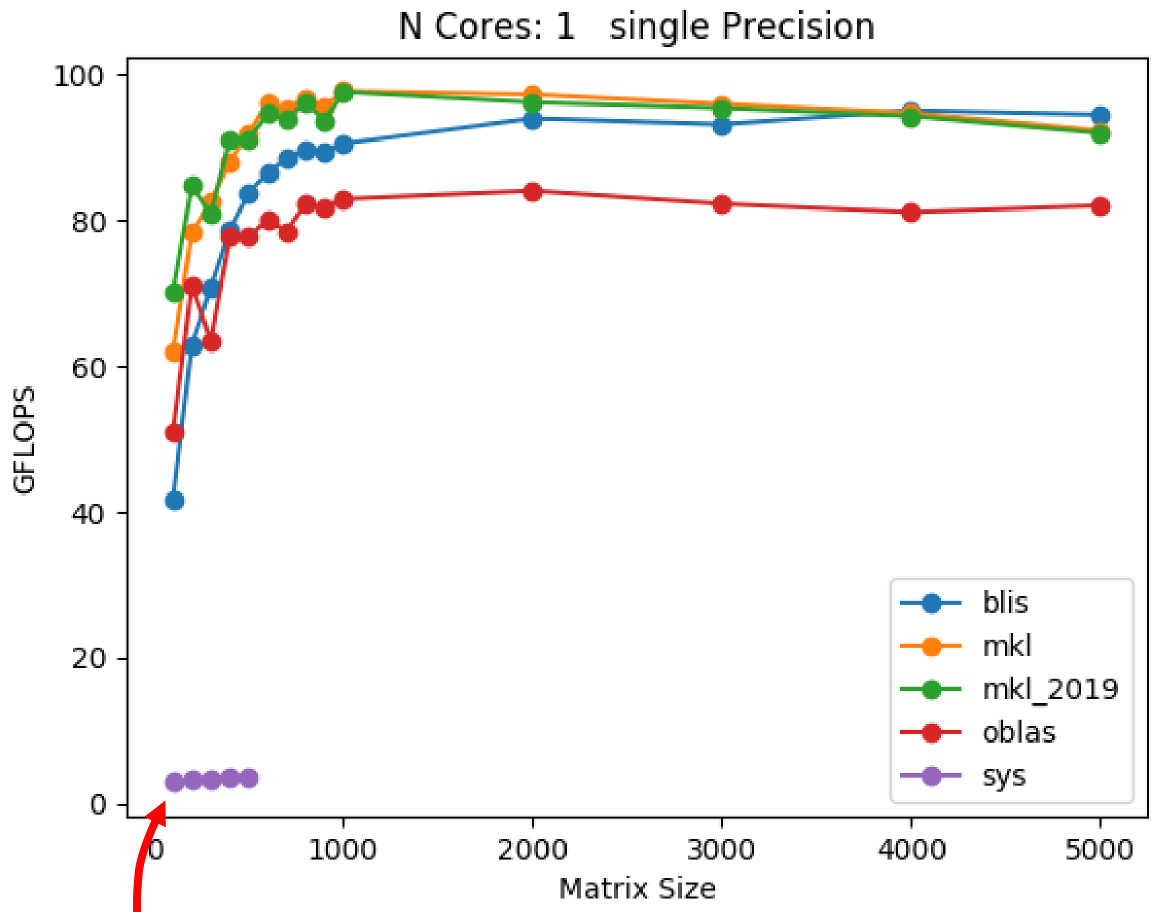
Common Parallel Libraries

Library	Parallelization	Notes
Python *multiprocessing*	Processes	Standard language library.
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)	Threads	Linear algebra. Widely used, for example by R and Python (via the numpy library).
Intel Math Kernel Library (MKL)	Threads	Linear algebra and a lot more. Widely used.
FFTW	Threads	Fast Fourier Transforms.
OpenCV	Threads	Image processing.
Tensorflow	Threads	Machine learning.
PETSc	Processes and threads	Partial differential equation solver, multi-compute node.
Hadoop and Spark	Processes and threads	Multi-compute node, includes a parallel file system.
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.

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



- SCC benchmark.
- Note poor performance of default Linux system BLAS library!

BOSTON UNIVERSITY

Enable OpenMP Threading Libraries on the SCC

- Most software on the SCC that uses multiple cores are built with the OpenMP threading library.
  - Including BLAS routines as commonly used by R and Python.
- The number of threads that will be used by your program can be set using the environment variable *OMP_NUM_THREADS*
  - If a program uses the Intel MKL library the threads can be set with *MKL_NUM_THREADS* instead
- The SCC sets *OMP_NUM_THREADS=1* by default.
- **NEVER** request more threads than there are cores for the job.

```
#$/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 nobel_winner/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?
nobel_winner ...etc...
```

BOSTON UNIVERSITY

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. In that case try setting the variable VECLIB_MAXIMUM_THREADS.

BOSTON UNIVERSITY

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!

The Message Passing Interface (MPI)

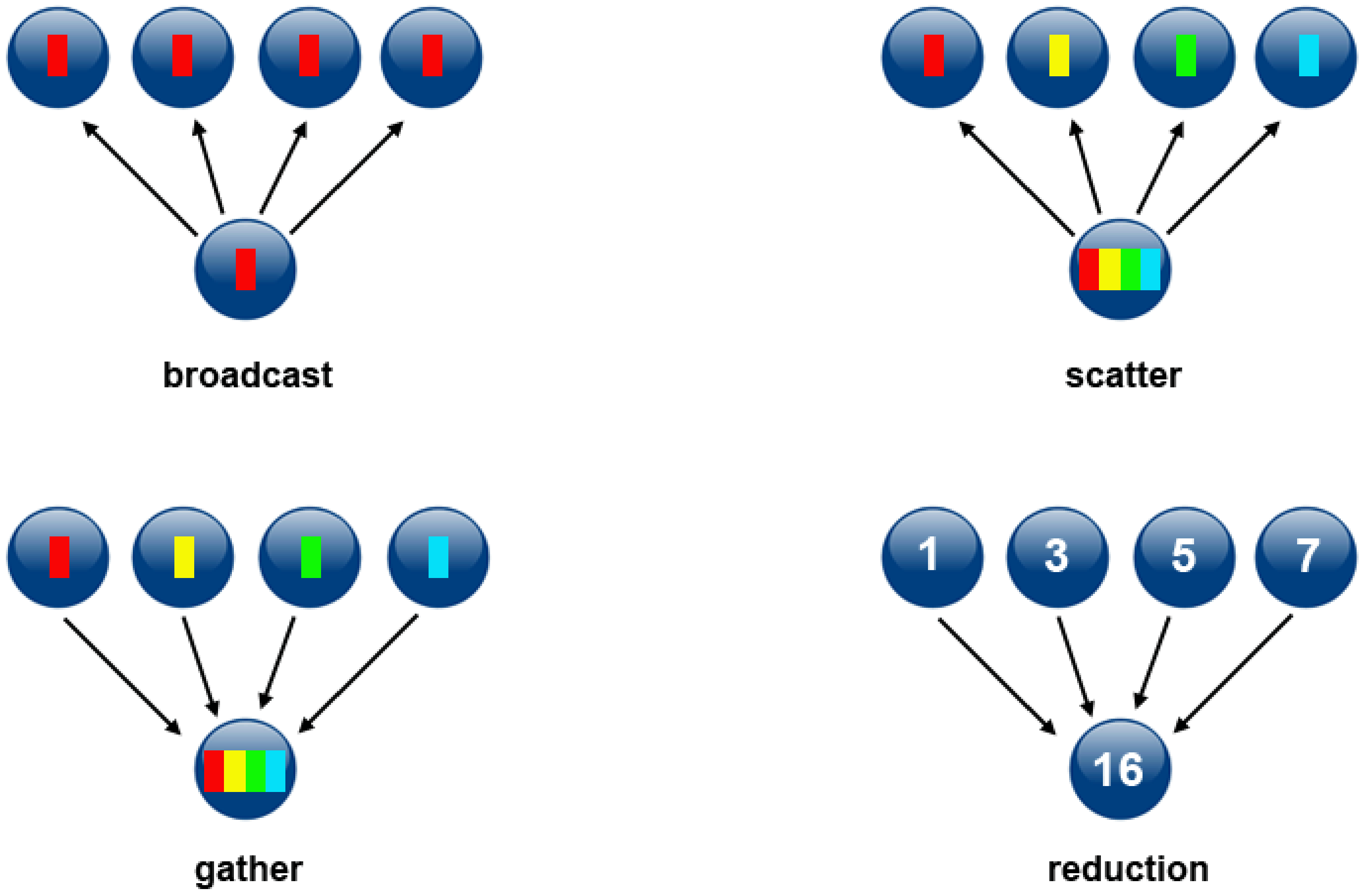
- With the right software tools processes can be run on multiple computers simultaneously and communicate with each other across a network.

- 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.



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.

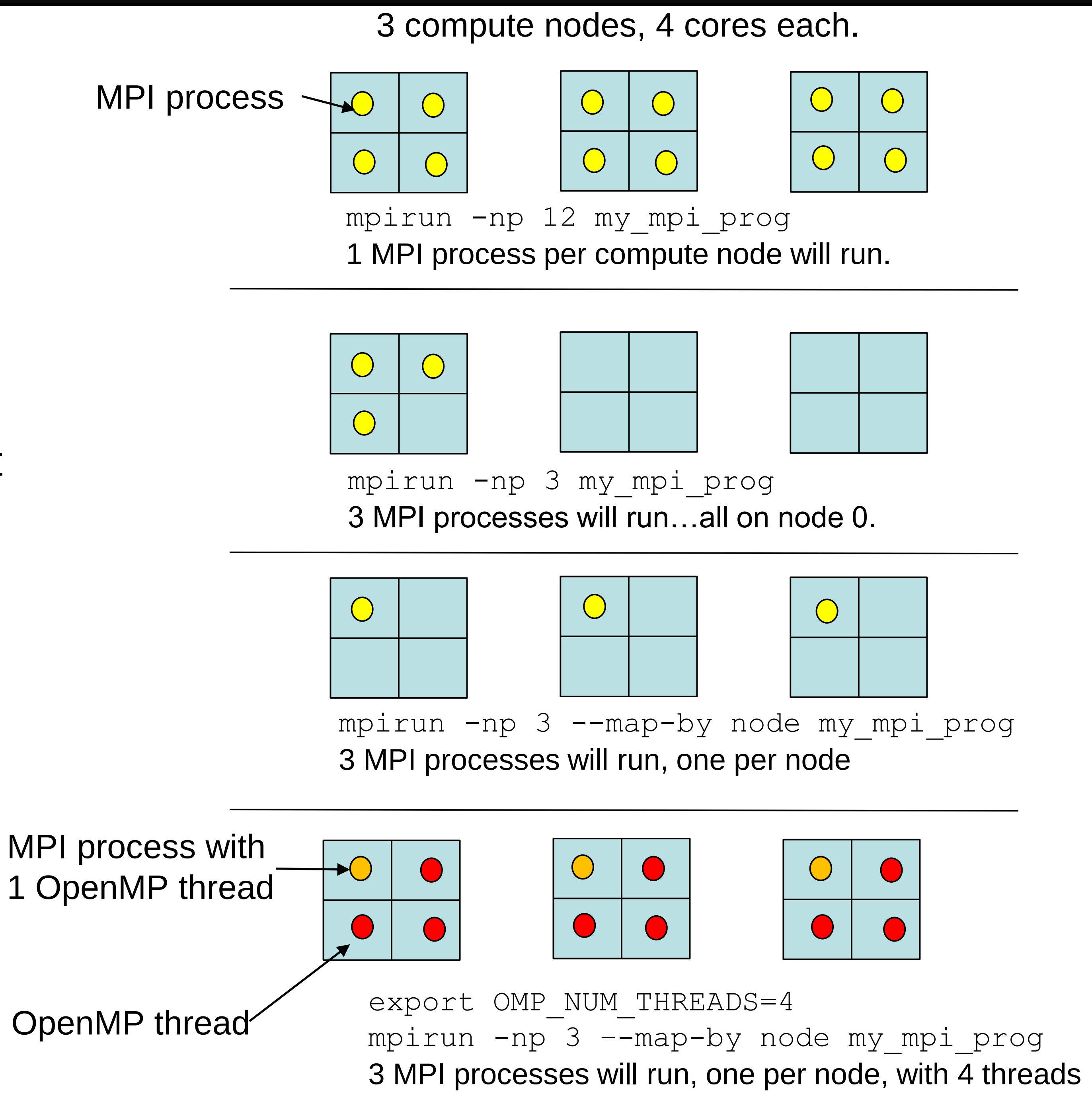
Language	Library
Python	mpi4py
R	Rmpi
Julia	MPI.jl
C#	MPI.NET

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

BOSTON UNIVERSITY

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.



BOSTON UNIVERSITY

SCC MPI Nodes

- 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.

BOSTON UNIVERSITY

Network Type	Bandwidth (Gbit/sec)	Latency (μs)
10gig Ethernet	10	12.5
FDR Infiniband	13.64	0.7
EDR Infiniband	25	0.5

- These jobs run on dedicated compute nodes connected with an *Infiniband* network. There are a couple of versions on the SCC.
- Latency is how quickly a data transfer can be initiated. For MPI computations this is often the limit, not the bandwidth.
- MPI jobs on a *single* compute node should use the regular "-pe omp N" queues.

Parallel Speedup

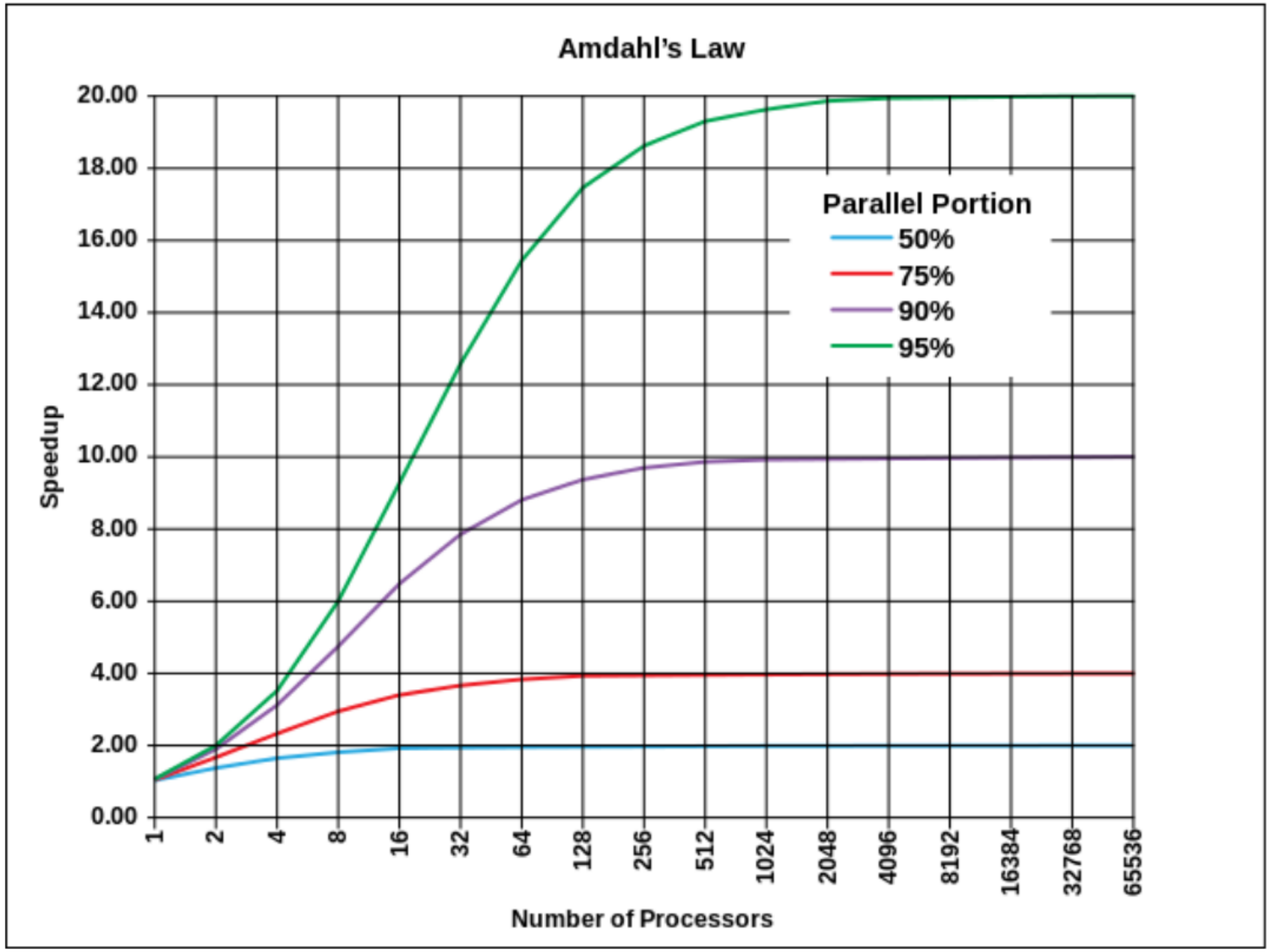
- There are many ways to parallelize code.
- …is it worth the effort and how much will it benefit you?

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

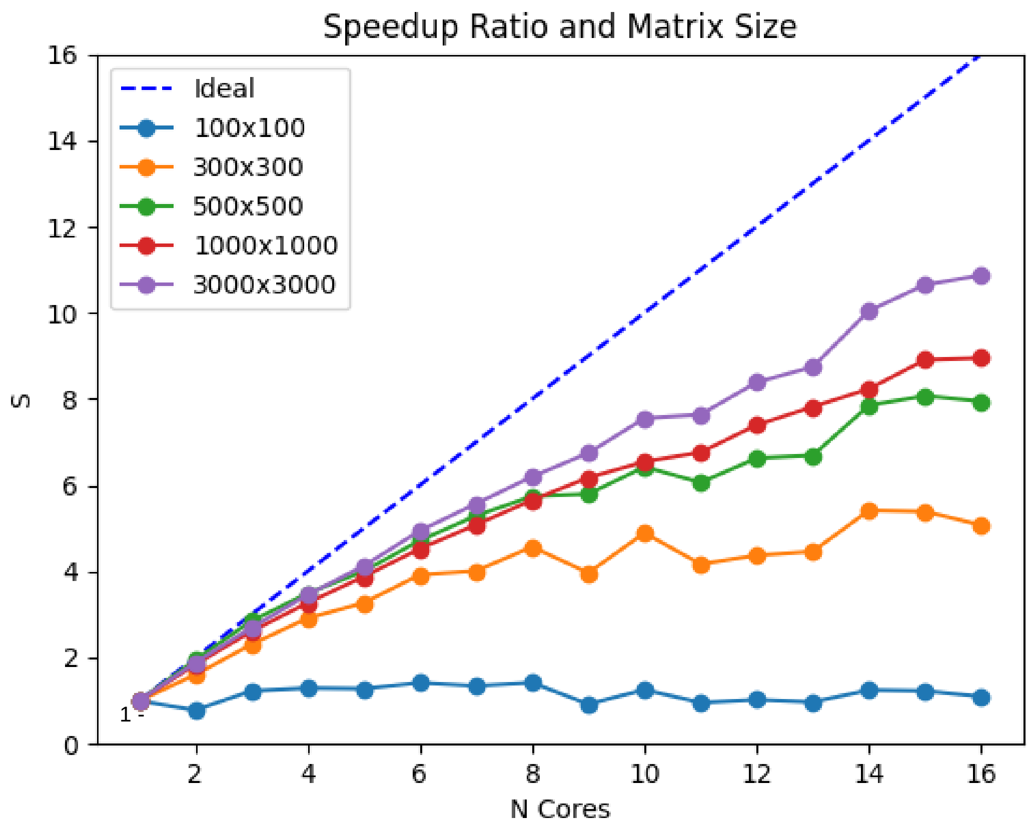


- This is the **theoretical** best speedup achievable with parallelization.

Figure from Wikipedia.

BOSTON UNIVERSITY

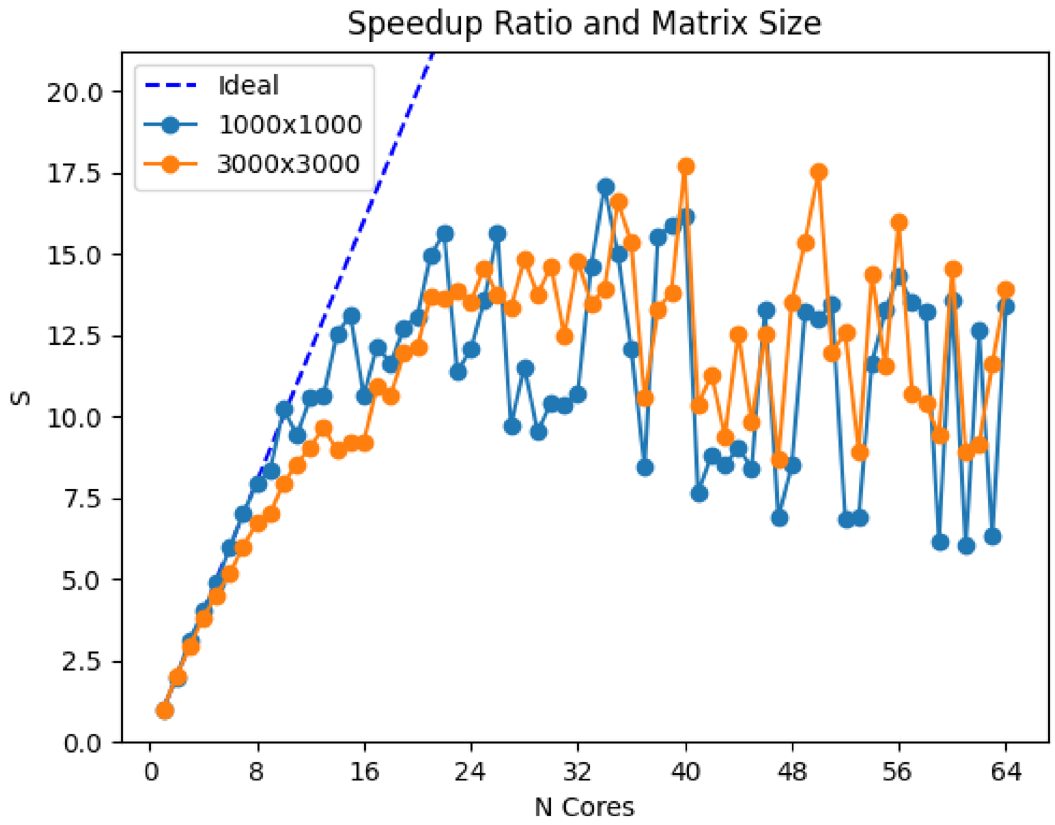
Intel Xeon CPU E5-2650 v2 @ 2.60GHz. 16 physical cores, 2 sockets (scc-pi2)



- 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 doesn't help.
  - You have to test…test…and test some more!
- SCC suggestion – try 4 threads/cores.

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.

BOSTON UNIVERSITY

Outline

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

Parallelization Difficulties

```
x = np.random.random([100])

# This cannot be parallelized.
for i in range(1,x.shape[0]-1):
        x[i] = x[i] - x[i-1] + x[i+1]
```

- 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 to parallelize.
- Sometimes you can alter a loop with additional copies of data to make it parallelize.
  - Trading off memory usage for computation time.

- Choose your battles wisely
- Use profiling to identify code that is worth improving.

BOSTON UNIVERSITY

Parallelization Difficulties

- Random number generation is not straightforward.
- Computing RNG's in parallel requires different random seeds for each worker.

- Do not assume that different workers, if seeded by default or from the system clock, will be generating different sequences of RNGs.

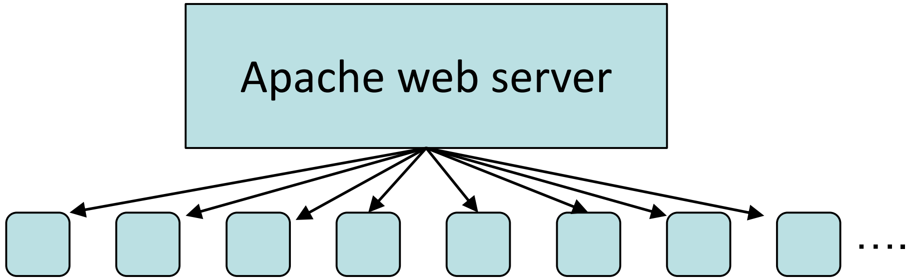
Parallelization Difficulties

- 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.

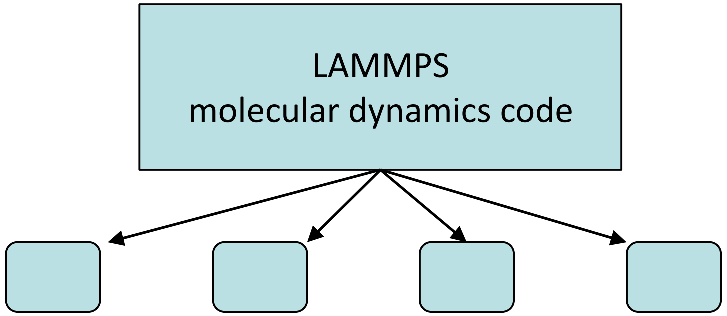
BOSTON UNIVERSITY

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

* Worker: thread or sub-process of a program

BOSTON UNIVERSITY

What happens with too many workers?

- For CPU-bound problems, use no more than 1 worker per core.
- More than 1 results in workers competing with each other for access to the cores and memory bandwidth.
- Performance will suffer significantly with excess workers.
- Watch for mixing multiple processes and multithreading (like OpenMP): each process can end up launching many threads, overloading the cores!

BOSTON UNIVERSITY