Lecture 01: Intro and Motivation

COSC 272: Parallel and Distributed Computing

Spring 2023

Outline

1. Course Motivation
2. Structure and Policies
3. Parallelism and Concurrency

Question

How do we quantify the **computational power** of a computer?

- How much more powerful are computers today than 5/10/25/50 years ago?

  - memory
  - computation speed
  - speed benchmarking

Speed $\approx$ Power?

Tangible notion computational power:

- run a program and see how long it takes
- less time = faster execution = more (evident) power

Processor Speed = Power?

clock speed

My first laptop:

- Compaq Presario 2100
- Intel Celeron processor $\boxed{\text{@1.6 GHz}}$
- $ 900 new (1,500 now w/ inflation)
- now < $ 15 used

Processor Speed = Power?

My first laptop:

- Compaq Presario 2100
- Intel Celeron processor $\boxed{\text{@1.6 GHz}}$
- $ 900 new (1,500 now w/ inflation)
- now < $ 15 used

slower

My current laptop:

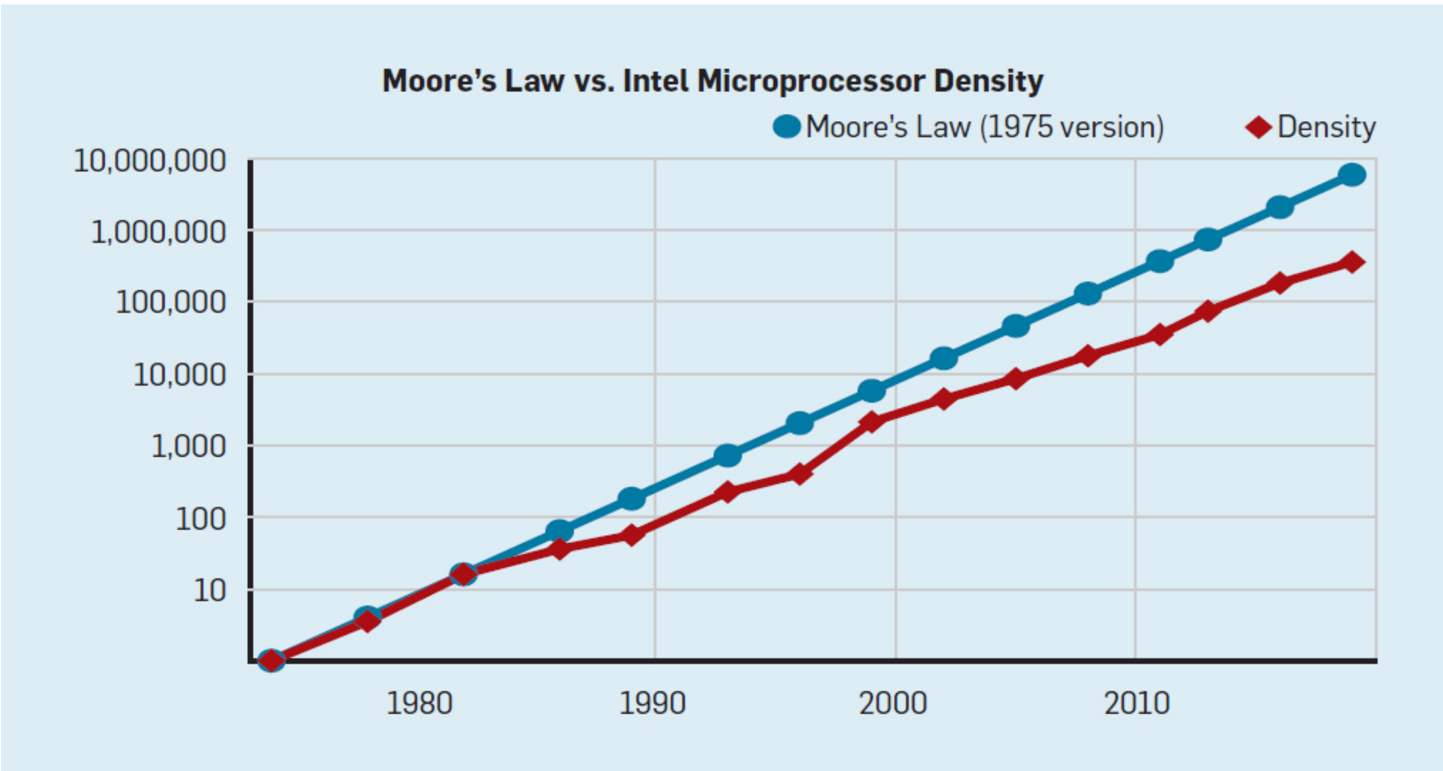
- Apple MacBook Pro, 2020
- Intel Core i5 $\boxed{\text{@1.4 GHz}}$ # cores 4
- $ 1,400 new (1,500 now w/ inflation)
- now ~ $ 600-1,000 used

Question. What gives?!

more cores?

Moore’s Law

Transister density chip doubles every 2 years



Transister density for Intel chips (img source).

But Processor *Speed* Is Not Increasing!

Year	Transistors	Clock speed	CPU model
1979	30 k	5 MHz	8088
1985	300 k	20 MHz	386
1989	1 M	20 MHz	486
1995	6 M	200 MHz	Pentium Pro
2000	40 M	2 000 MHz	Pentium 4
2005	100 M	3 000 MHz	2-core Pentium D
2008	700 M	3 000 MHz	8-core Nehalem
2014	6 B	2 000 MHz	18-core Haswell
2017	20 B	3 000 MHz	32-core AMD Epyc
2019	40 B	3 000 MHz	64-core AMD Rome

Speed vs Throughput

Processor speed = # processor clock cycles per second

- *latency* of a single operation bounded by processor speed
- phyiscal constraints (e.g., speed of light) limit processor speed

Speed vs Throughput

Processor speed = # processor clock cycles per second

- *latency* of a single operation bounded by processor speed
- phyiscal constraints (e.g., speed of light) limit processor speed

Throughput = # useful operations processor can perform per second

Question. How can we increase throughput without increasing speed?

- do more than 1 op at a time

What is Parallelism?

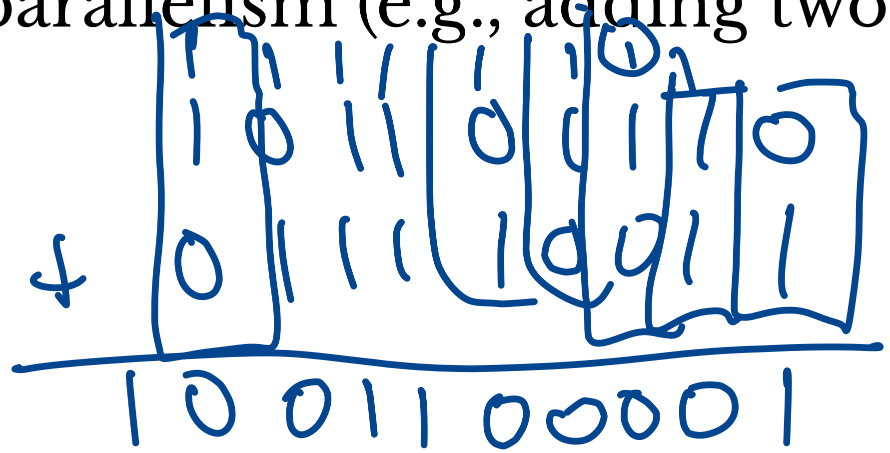
The ability to perform multiple operations *simultaneously*.

What is Parallelism?

The ability to perform multiple operations *simultaneously*.

Examples:

- bit-level parallelism (e.g., adding two 32-bit numbers)



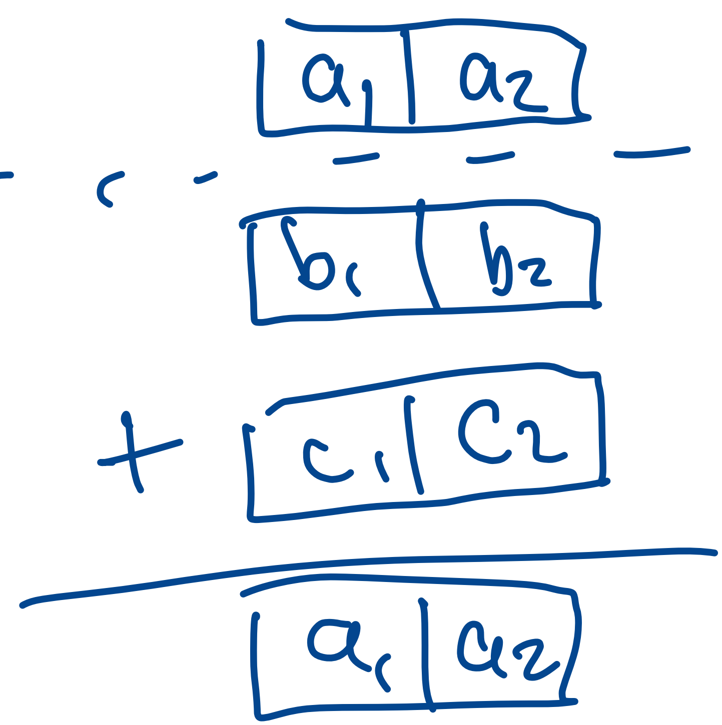
What is Parallelism?

The ability to perform multiple operations *simultaneously*.

Examples:

- bit-level parallelism (e.g., adding two 32-bit numbers)
- single instruction, multiple data (SIMD) parallelism

$$\rightarrow a1 = b1 + c1$$
$$\rightarrow a2 = b2 + c2$$



What is Parallelism?

The ability to perform multiple operations *simultaneously*.

Examples:

- bit-level parallelism (e.g., adding two 32-bit numbers)
- single instruction, multiple data (SIMD) parallelism
- multi-core: independent processors operating at same time on same computer

What is Parallelism?

The ability to perform multiple operations *simultaneously*.

Examples:

- bit-level parallelism (e.g., adding two 32-bit numbers)
- single instruction, multiple data (SIMD) parallelism
- multi-core: independent processors operating at same time on same computer
- distributed networks: clusters, server farms, internet

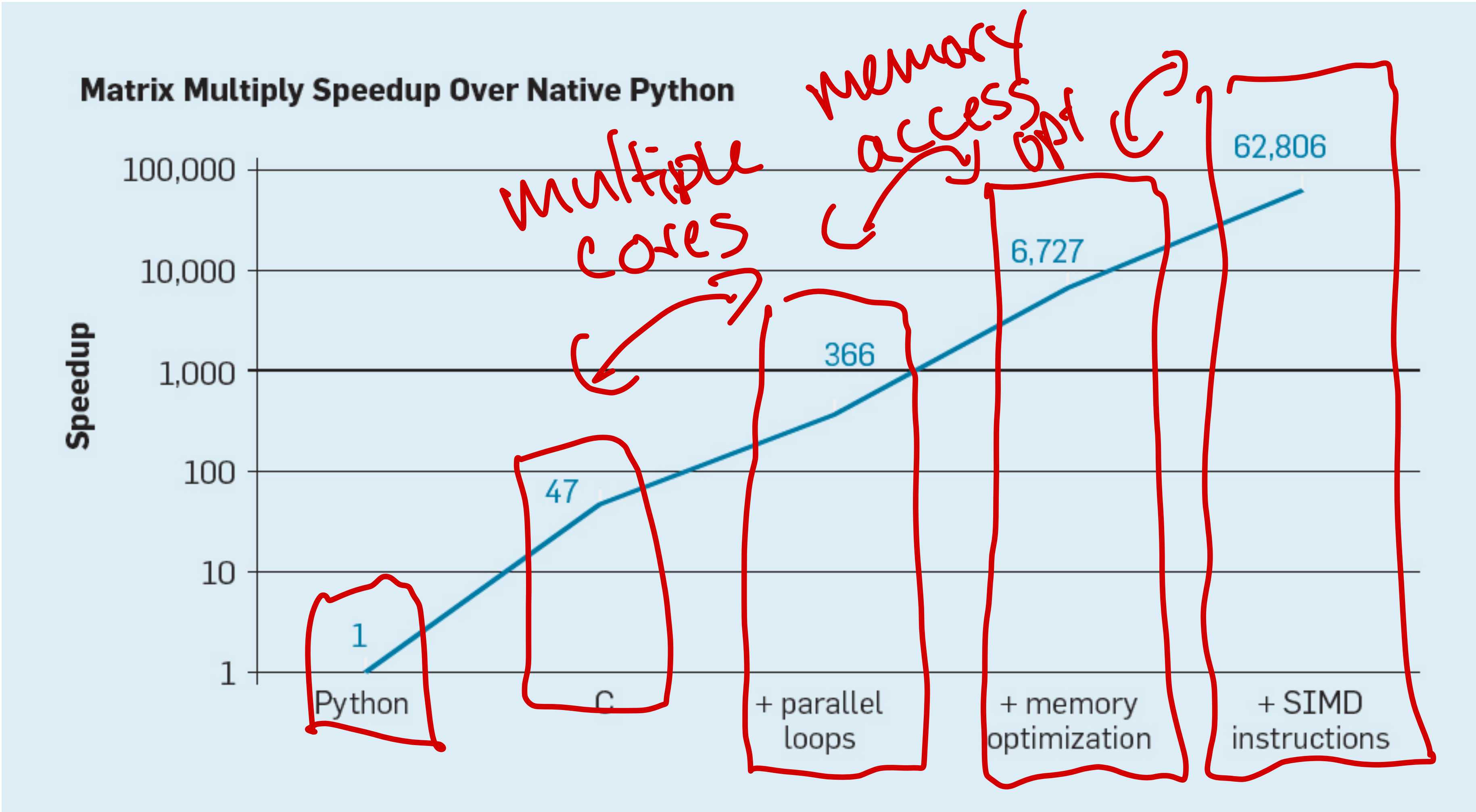
HPC Cluster

Promise of Parallelism

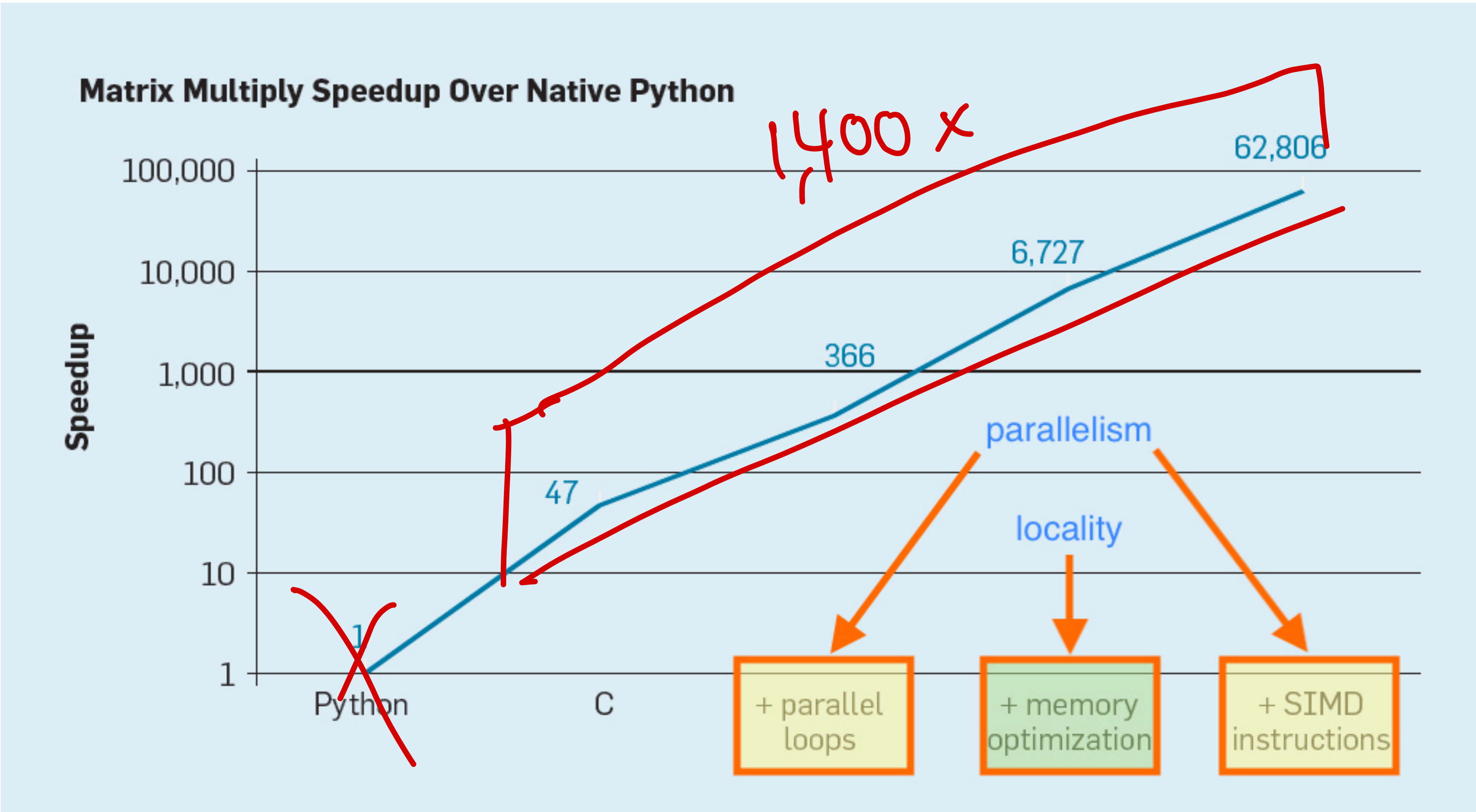
"Many hands make light work."

- More processors $\implies$ more operations per second!
- Greater throughput!
- Perform multiple operations at once!

The Power of Parallelism I



The Power of Parallelism II



Perils of Parallelism

More processors, more problems

Perils of Parallelism

More processors, more problems

- Some computations need to be done sequentially in order
  - next step relies on result of current step ← dependencies between ops

Perils of Parallelism

More processors, more problems

- Some computations need to be done sequentially in order
  - next step relies on result of current step
- Processors must *share* resources
  - communication and sychronization are costly

Perils of Parallelism

More processors, more problems

- Some computations need to be done sequentially in order
  - next step relies on result of current step
- Processors must *share* resources
  - communication and sychronization are costly
- Nondeterminism
  - different executions give different behavior
  - algorithms must account for all possible executions!

Unavoidability of Parallel & Distributed Computing

Modern computing is inherently distributed!

- Different parts of the computer interact
  - cores within processors
  - processor registers, cache, main memory, IO, etc.
- Different computers interact
  - local computer networks
  - clusters and server farms
  - internet

What this Course is About

1. Exploiting parallelism to write performant code
    - write programs that are 100s of times faster than simple sequential code
2. Reasoning about parallel programs and executions
    - parallel programs can be incredibly subtle!
    - arguing correctness is significantly more challenging than for sequential programs

Course Structure

Expected Background (Programming)

- Object Oriented design in Java
  - classes and inheritance
  - interfaces
  - exception handling
  - generics

Expected Background (Conceptual)

- Basic data structures:
  - linked lists
  - stacks
  - queues
  - (balanced) trees
- Supported operations, and their complexities

Main Topics Covered

- multithreaded programming in Java
- mutual exclusion,
- concurrent objects,
- locks and contention resolution,
- blocking synchronization,
- concurrent data structures,
- SIMD/vector operations

Course Materials

- *The Art of Multiprocessor Programming* (Moodle -> Course Reserves)
- Notes (posted to course website)
- Recorded lectures

Course Focus

- *Principles* of parallel computing:
  - conceptual & technical issues that are fundamental to parallel programming
  - indpendent of computing technology
  - want provable guarantees for behavior
- We care about *performance* but...
  - newest technologies will *not* be emphasized
  - prefer methods that enhance our understanding of a problem

Course Structure

- 3 lectures/week
  - guided discussion
  - small group discussion
  - mixture of lecture/discussion/activities
  - small-group activities will require your laptop
- Readings posted to course website
  - do readings **before** class

Coursework

- Coding Assignments (bi-weekly)
  - some individual, some pairs
  - focus on performance
  - semi-competitive
- Written Assignments (bi-weekly)
  - focus on formal reasoning/problem solving
  - small groups
- Quizzes/in class activities
- Final Project (small group)

Attendance & Illness

Attendance

- Regular attendance is expected
- No penalty for a few missed classes
  - lectures will be recorded and posted to Moodle

Illness & Masking

- do **not** attend class if you are sick (e.g., with fever)
- if mild symptoms:
  - take a Covid test before coming to class
  - wear a mask
- otherwise come to class, masks optional

Office Hours

Will’s office: SCCE C216

Drop-in (in person):

- M/W/F After Class (11:00–11:30)

By appointment (in person or on Zoom):

- Thursday (time tbd)

please wear a mask to in-person office hours

This Week

1. Writing multithreaded programs in Java
2. Subtleties of multithreading

What is Multithreading?

Preliminary question. What is a *program*?

What is Multithreading?

Preliminary question. What is a *program?*

- A sequence of *operations* to be performed
- some operations may depend out the outcomes of other operations, others may be independent:

```
a1 = b1 + c1;
a2 = b2 + c2;
p = a1 * a2
```

What is Multithreading?

Preliminary question. What is a *program?*

- A sequence of *operations* to be performed
- some operations may depend out the outcomes of other operations, others may be independent:

```
a1 = b1 + c1;
a2 = b2 + c2;
p = a1 * a2
```

A **thread** is a sequence of operations—think *sub*program

- different threads specify **logically independent** sequences operations

Art of Multithreading

Goal. Partition a program into multiple (logically indpendent) threads.

Payoff. Different threads can be executed in parallel (on parallel computer architecture)

- computer with k cores could see up to a k-fold increase in throughput!

Art of Multithreading

Goal. Partition a program into multiple (logically indpendent) threads.

Payoff. Different threads can be executed in parallel (on parallel computer architecture)

- computer with k cores could see up to a k-fold increase in throughput!

Challenges.

- How to partition a program into threads?
- How to synchronize resources that must be shared by threads? (e.g., memory)
- How to ensure program **always** gives desired output?
  - OS ultimately decides how to allocate resources...