Lecture 02: Multithreading in Java

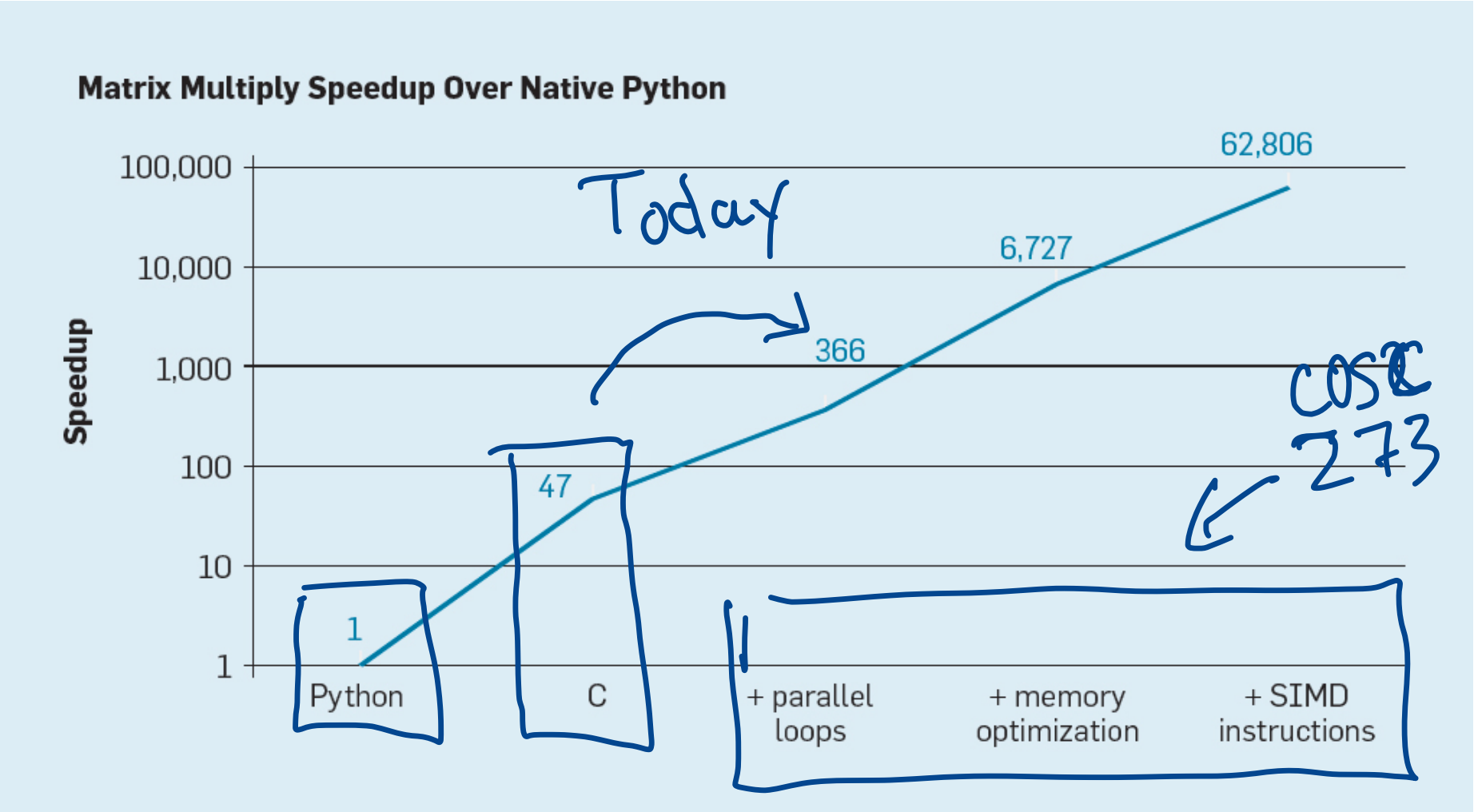
COSC 272: Parallel and Distributed Computing

Spring 2023

Outline

1. What is multithreading?
2. Writing multithreaded programs in Java
3. Activity: Counter Example
4. RAM and PRAM

Last Time: Motivation

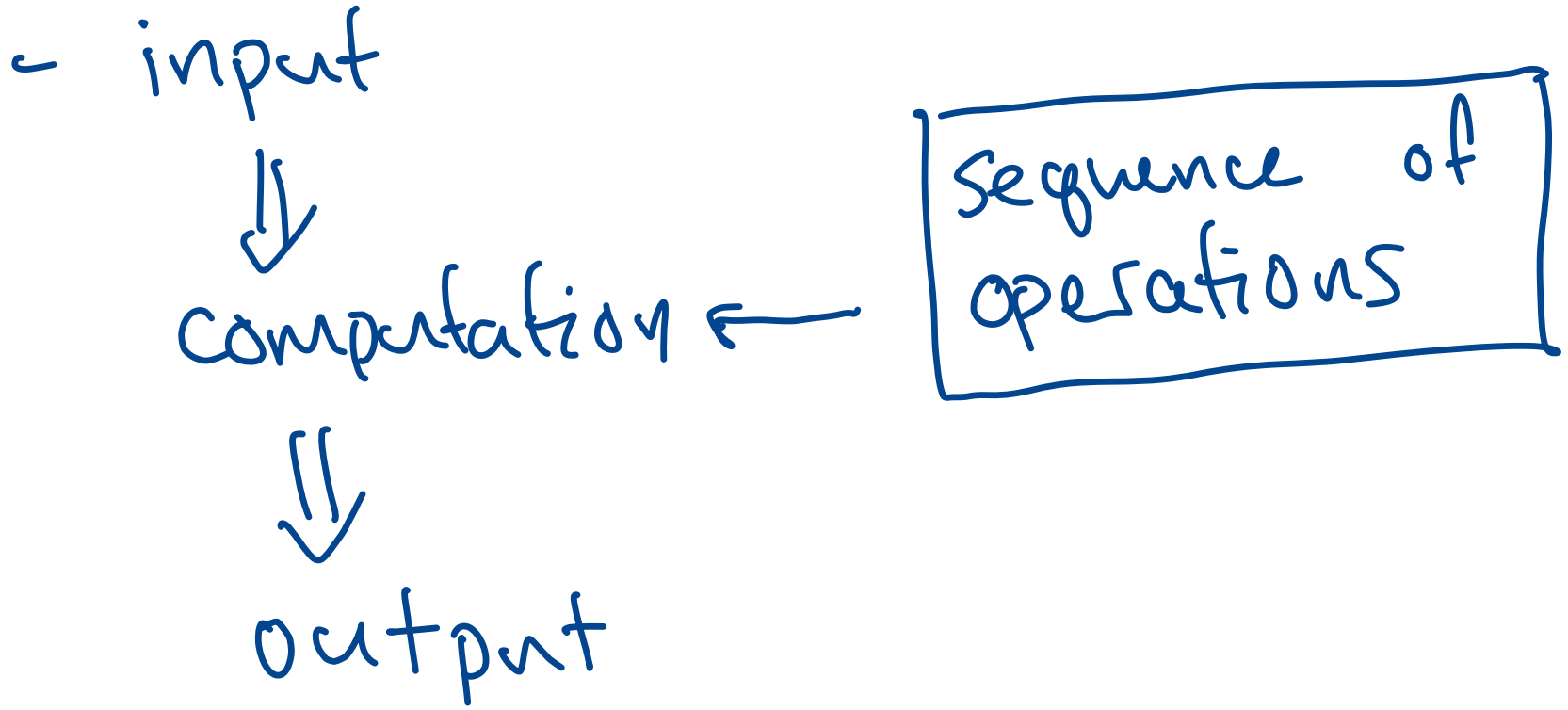


Today

Writing multithreaded programs!

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:

logically independent

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...

Multithreading in Java

Steps to writing a multithreaded program

1. Define a `Runnable` object
   - class implements the `Runnable` interface
   - must implement a method `void run()`
   - `run()` defines what your thread should do
2. Create a `Thread` instance initialized with an instance of your `Runnable` object
3. Start the thread
4. (optional) Wait for the thread to complete

built-in

"join"

Example

A thread that increments a counter a bunch of times.

- lec02-shared-counter.zip

Step 1: Define Runnable Object

```
public class CounterThread implements Runnable {
    private Counter counter; private long times;

    public CounterThread (Counter counter, long times) {
        this.counter = counter; this.times = times;
    }

    public void run () {
        for (long i = 0; i < times; i++) {
            counter.increment();
        }
    }
```

what thread does

What about the Counter?

```
public class Counter {
    private long count = 0;

    // return the current counter value
    public long getCount () { return count; }

    // increment the counter
    public void increment () { ++count; }

    // reset the counter value to 0
    public void reset () { count = 0; }
}
```

← # times increm. since last reset

Next Steps

Step 2. Create a `Thread` instance initialized with an instance of your `Runnable` object

Step 3. Start the thread

Step 4. (optional) Wait for the thread to complete

- See `CounterExample.java`

Activity (Small Groups)

1. Run `CounterExample` with `NUM_THREADS` set to 1. What happens?
2. Run `CounterExample` with `NUM_THREADS` set to 2.
   - How does the final count change?
   - How does the running time change?
3. Repeat 2 for `NUM_THREADS` set to 4, 8, 16, 1000, 1000...

What Happened?

- What happened with final counts as number of threads increased?

1 thread
2 thread
4 ——
1000 ——

100M
~50M
~28M
~97M, 98M, 65M

50,548,490

- What happend with running times?

7 ms,

256 ms
150 ms
~~80~~ ms
151 ms

1
2
4
16

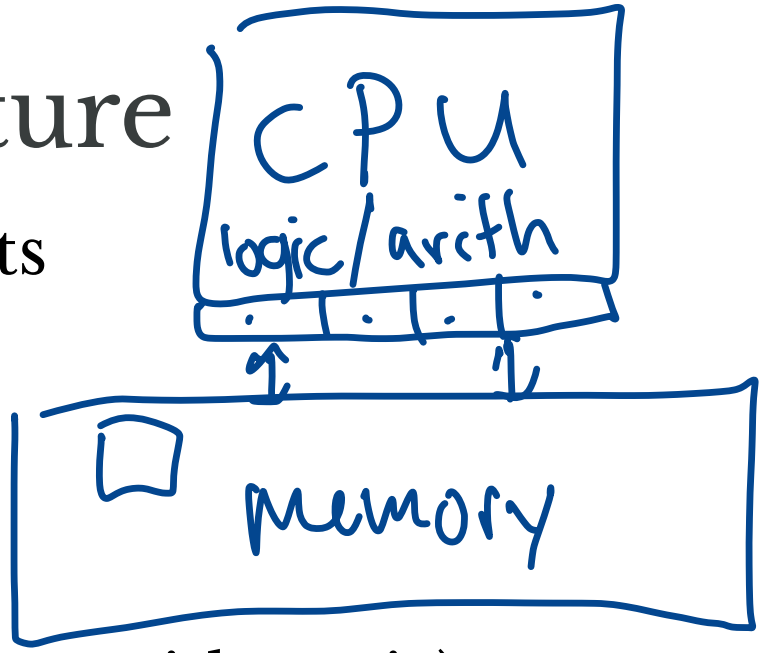
Question

Why did this behavior occur?

Understanding What Happened

Computer Architecture, Oversimplified

von Neuman Architechture



Computer has two main components

- Central Processing Unit (CPU)
- Memory Unit

CPU Capabilities:

- perform fixed set of operations (e.g., arithmetic)
- program control (e.g., branching)

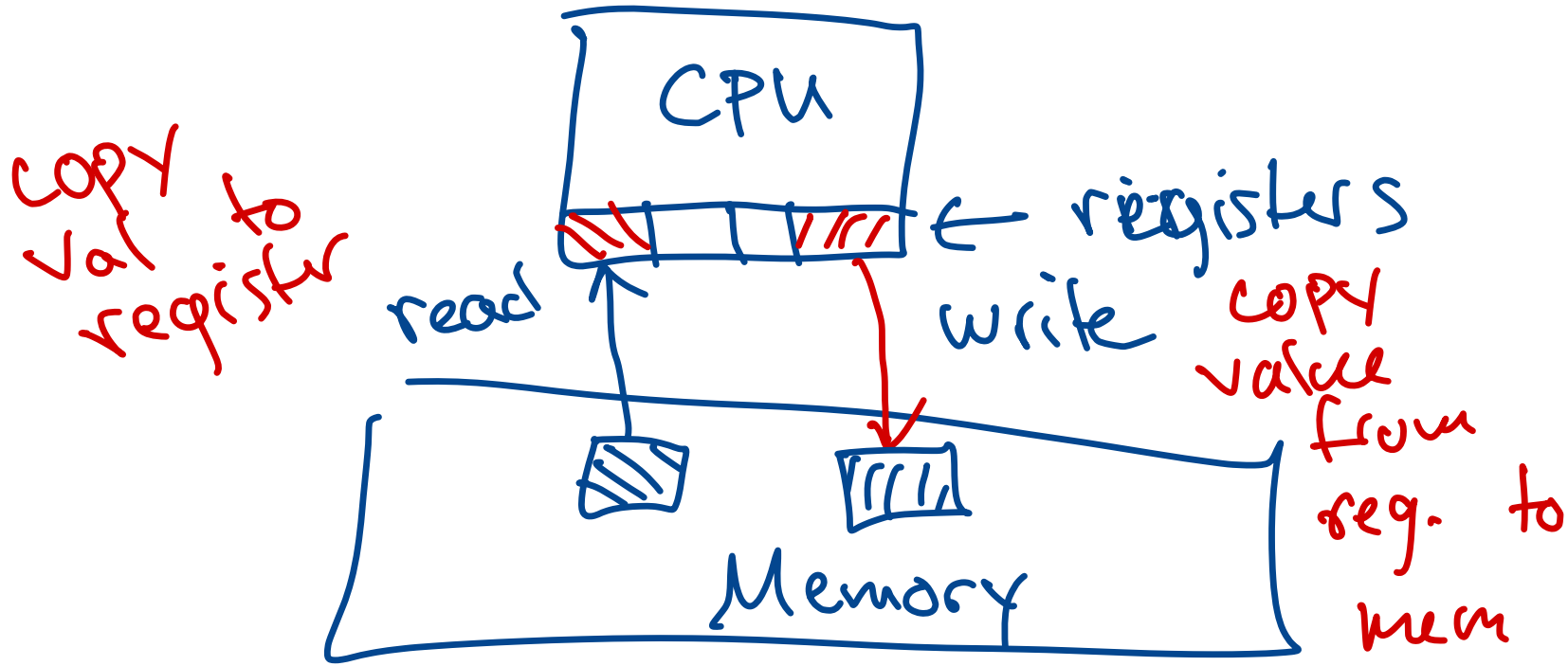
Memory stores:

- program instructions
- data

CPU/Memory Interactions

Random Access Machine (RAM) model interactions:

- **read** a value from memory address
  - load value into CPU register
- **write** a value to memory address
  - copy value stored in CPU register



Counter Example, 1 thread

- `Counter` object is stored in memory
  - `Counter` stores a value `count`
- `CountThread` instructions stored in memory
- When `CounterThread` is executed, it follows these instructions

```
for (long i = 0; i < times; i++) {
  counter.increment();
}
```

- In turn:

```
public void increment () { ++count; }
```

++count —

{ read val of count to reg.
increment register
write new value back to mem.

Question

What are CPU/Memory interactions when `counter.increment()` is executed?

```
public void increment () { ++count; }
```

Multicore Architecture

Modern computers:

- multiple cores
  - think of them as separate, independent CPUs
  - different cores *can* execute different threads simultaneously
- **shared memory**

Multicore Counter Example

- two threads perform `increment` operation on different cores
- threads both try to `increment` **same** `Counter` concurrently

Question

Suppose: `count = 7` & two threads both call `increment()` concurrently

What are the possible outcomes? What are results of different read/write operations?

PRAM model

Parallel Random Access Machine (PRAM)

- Abstract model for parallel computing
- Shared memory: cells w/ addresses
  - think one giant array
- Multiple processors access memory
  - basic operations are `read(i)` and `write(i, val)`

PRAM Assumptions

- `read`/`write` operations are **atomic**

Nondeterminism:

- if multiple threads access same memory location simultaneously all "consistent" outcomes are possible
  - two processes call `write(i, a)` and `write(i, b)`
  - one process calls `read(i)` another `write(i, a)`

Next Time

Consider: How could we avoid the `CounterExample` weirdness (nondeterminacy) and get a correct count with multiple threads?

More on nondeterminacy!