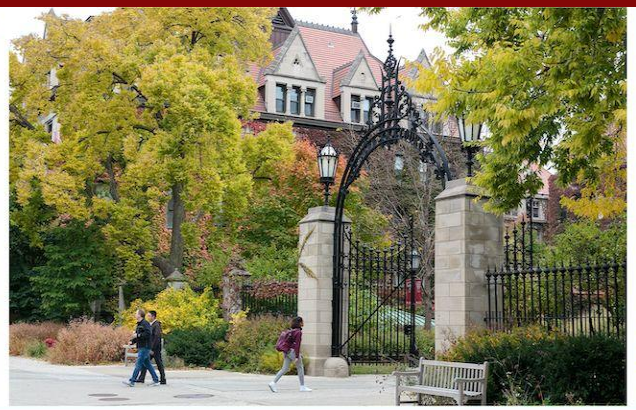
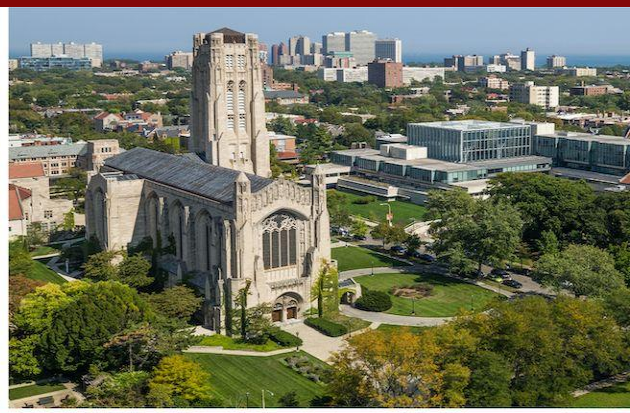
MPCS 52060 - Parallel Programming
M1: Introduction to Parallel Programming

Lamont Samuels

Course Logistics

- All course information is located on the course website

  https://www.classes.cs.uchicago.edu/archive/2024/spring/52060-1/index.html

Motivation for Parallelism

Basic Architecture of a Computer

- The modern computers are based an architecture introduced by John Von Neumann (i.e., Von Neumann architecture). These are the main components:
  - Memory - a collection of locations, each of which is capable of storing both instructions and data. Every location consists of an address, which is used to access the location, and the contents of the location.
    - A computer can contain various types of memory components (registers, caches, RAM, secondary memory (i.e., hard drive) etc.)
  - Central Processing Unit (CPU) - is the controller of a computer and is composed of many different parts. However there are two significant parts: a *control unit* and a *arithmetic and logic unit*
    - Control unit - responsible for deciding which instruction in a program should be executed. (the boss)
    - Arithmetic and logic unit (ALU) - responsible for executing the actual instructions. (the worker)
  - Input devices - devices that send information to a computer for processing (e.g., keyboard, mouse, etc.)
  - Output devices - devices that display/use the results of processing of tasks by the computer. (e.g., monitor, printer, speakers)

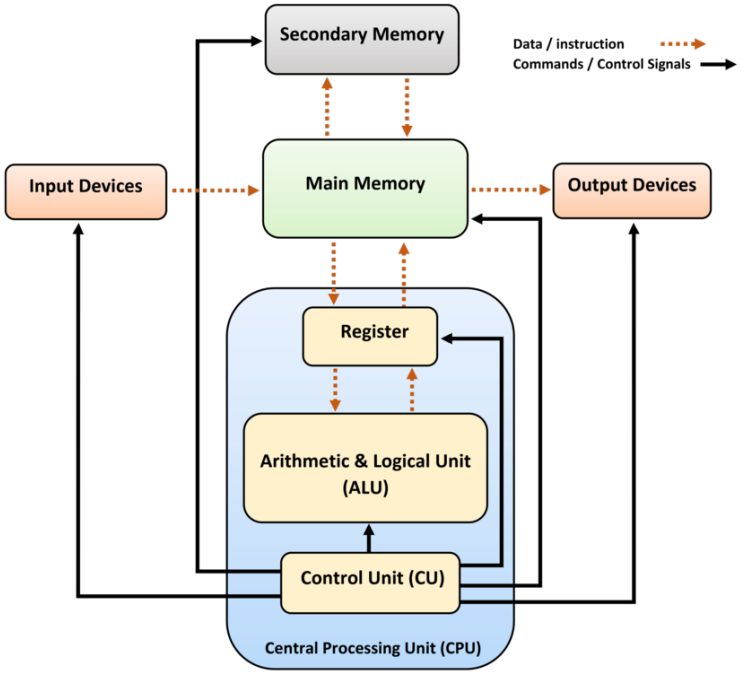


Diagram Source: https://commons.wikimedia.org/wiki/File:Computer_architecture_block_diagram.

Sequential Programs

- As programmers, we are overwhelmingly accustomed to developing sequential programs
  - A program solves a problem that is composed of a series of textual statements that specifies their order of execution.
  - Each statement is converted into an instruction that is executed by the CPU.
  - Sequential programs are deterministic, which means which means that every time the program runs with the same input the resulting output is the same.

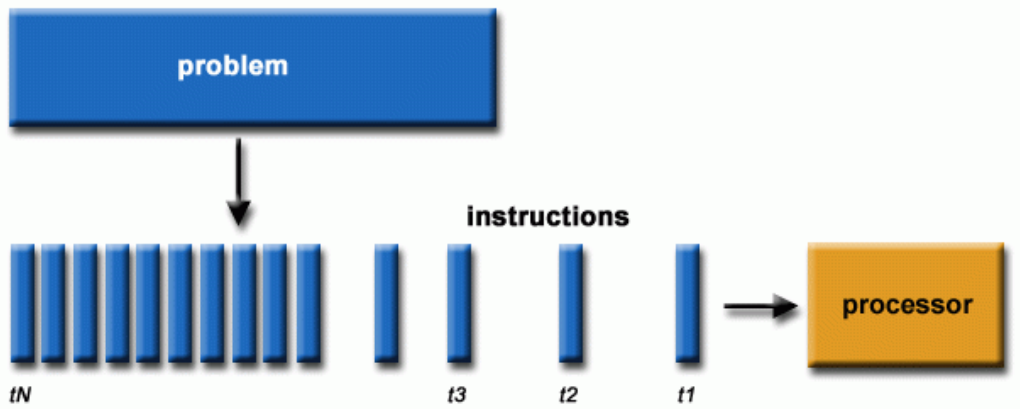
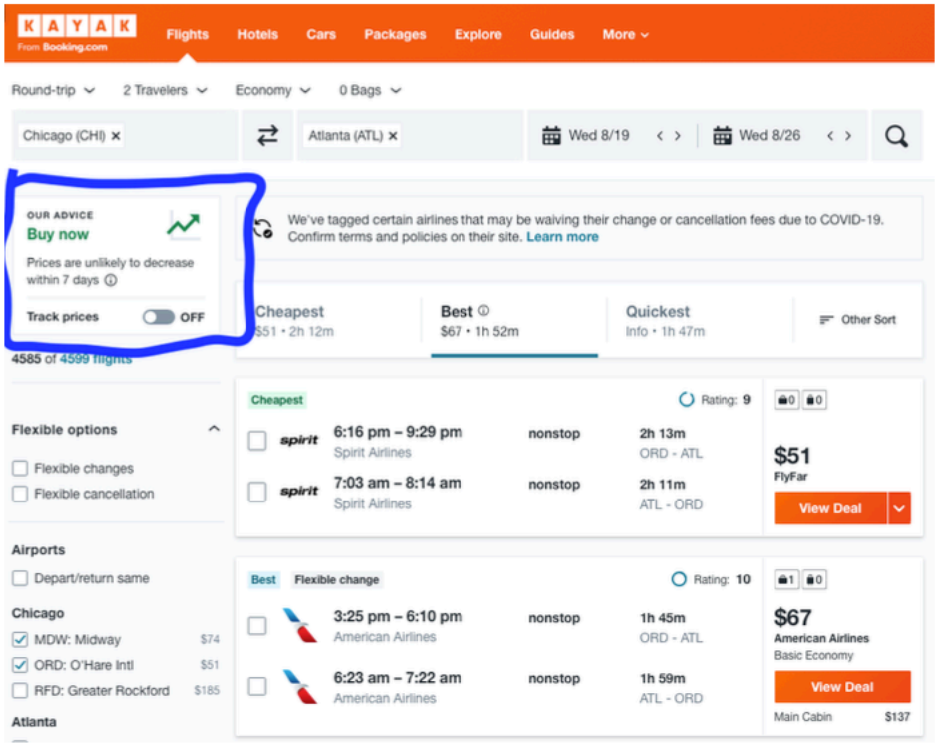


Diagram Source: https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial

Sequential Program Example

- Simple example of serial computation:
  - Problem: Develop a program that gathers data about route information from different cities. The overall objective is to use this program to help develop a classifier to tell a user whether to buy a ticket now or later.



Crawler Components

- Components of the route finder application:
  - A web crawler that crawls flight route information for popular cities on Fare Detective (https://www.faredetective.com/farehistory/).
  - Program Output: A CSV file with lowest-cost fare information from the cities listed on the site.

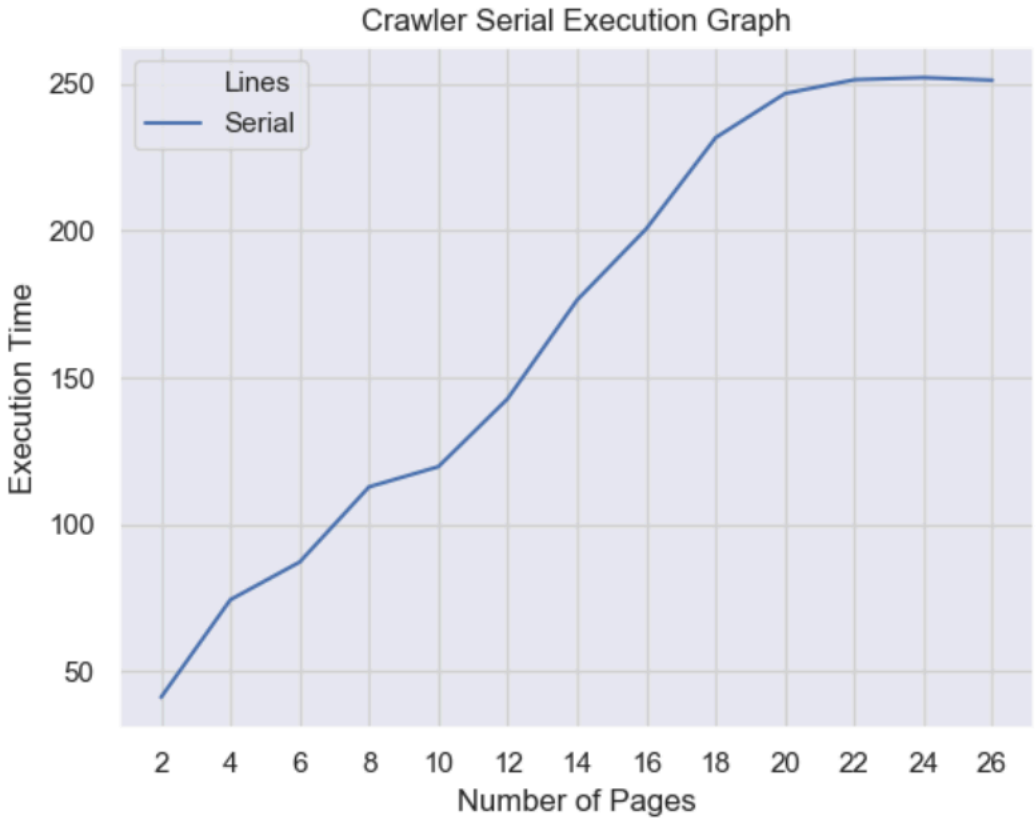
Partial CSV Output

routes

From	To	Fare
Amsterdam	Dublin	50
Amsterdam	South Hampton	88
Amsterdam	Birmingham	108
Amsterdam	Belfast	128
Amsterdam	Belfast	136
Amsterdam	Aberdeen	137
Amsterdam	Manchester	185
Amsterdam	Venice	190
Amsterdam	Ljubliana	206
Amsterdam	Bristol	209
Amsterdam	Trieste	236
Amsterdam	Gothenburg	296
Amsterdam	Minneapolis	338
Amsterdam	Boston	391
Amsterdam	Montreal	445
Amsterdam	Washington DC	549
Amsterdam	Los Angeles	601
Amsterdam	San Francisco	619
Amsterdam	Vancouver	624
Amsterdam	West Palm Beach	643
Amsterdam	Kuching	664
Amsterdam	Toronto	682
Amsterdam	Shanghai	711
Amsterdam	Houston	718
Amsterdam	Washington DC	724
Amsterdam	Calgary	732
Amsterdam	Tyler	734
Amsterdam	Oakland	865
Amsterdam	Portland	876
Amsterdam	Denpasar Bali	895
Amsterdam	San Jose	908
Amsterdam	Jakarta	941
Amsterdam	Shanghai	949
Amsterdam	Santa Ana	977
Amsterdam	Atlanta	1000
Amsterdam	Portland	1018
Amsterdam	Denver	1026
Amsterdam	Norfolk	1063
Amsterdam	Honolulu	1069

Crawler Performance

- Execution time (in seconds):



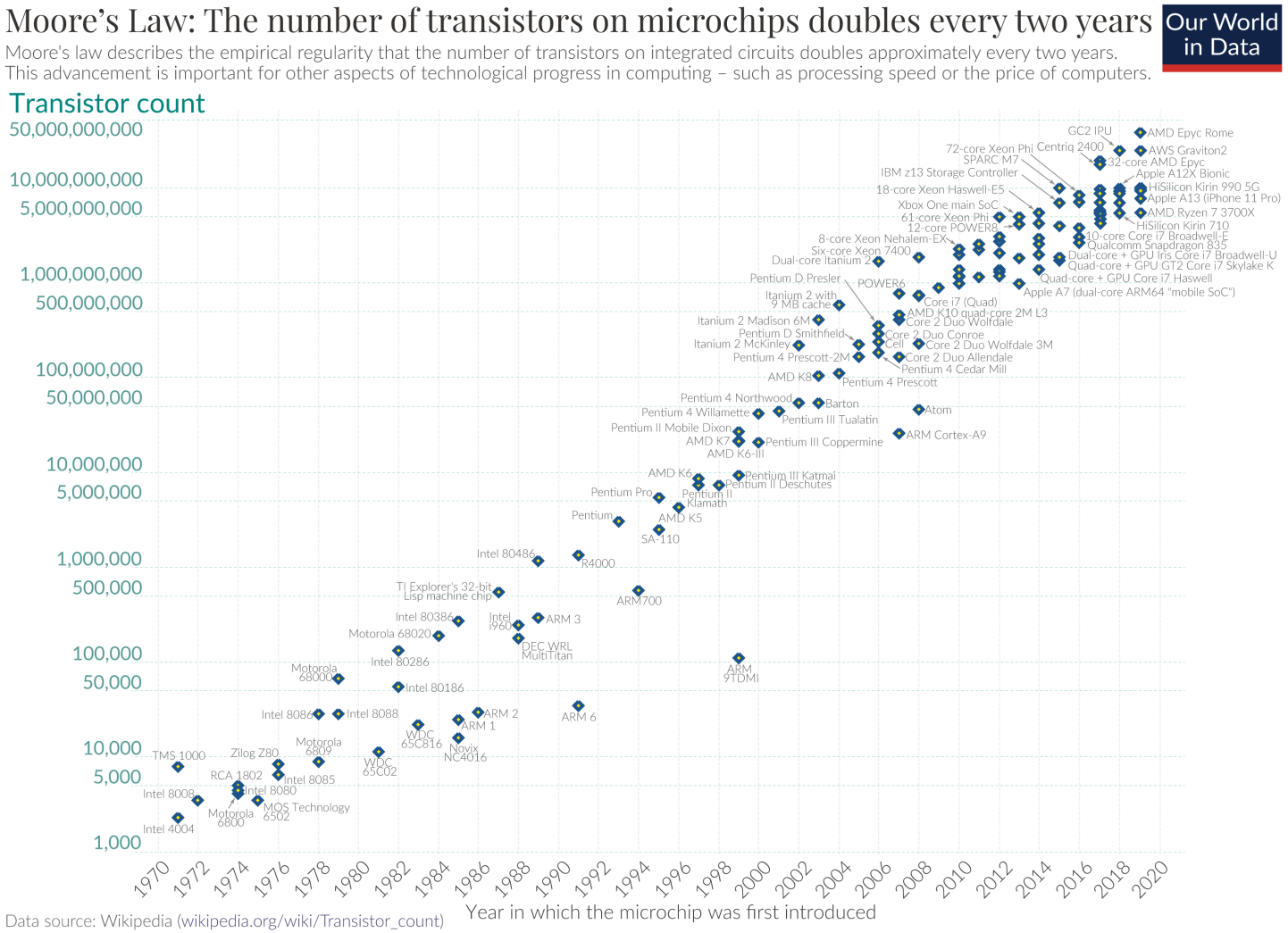
Is Sequential Programming Good Enough?

- For many years, sequential programs executed faster on newer processors with no modification.
- Since ~2003, single-processor performance improvement has slowed to about 20% per year. *Maybe not?*
- History of Hardware Trends
  - Increase in single processor performance has been driven by increasing the density of transistors on the CPU chip.
    - Transistors - electronic components on integrated circuits that act as switches in order to construct logical gates. We use logic gates to form logical units capable of arithmetic and complex logical operations.
  - As the size of the transistors decreases, their speed can be increased, and the overall speed of the integrated chip will be increased (i.e., increasing clock rate).

Moore's Law

- Computing power tends to approximately double every two years.
  - What the law really means is that the number of transistors that can be packed into a given unit of space will roughly double every two years.

History of Moore's Law



Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)
OurWorldinData.org – Research and data to make progress against the world's largest problems.

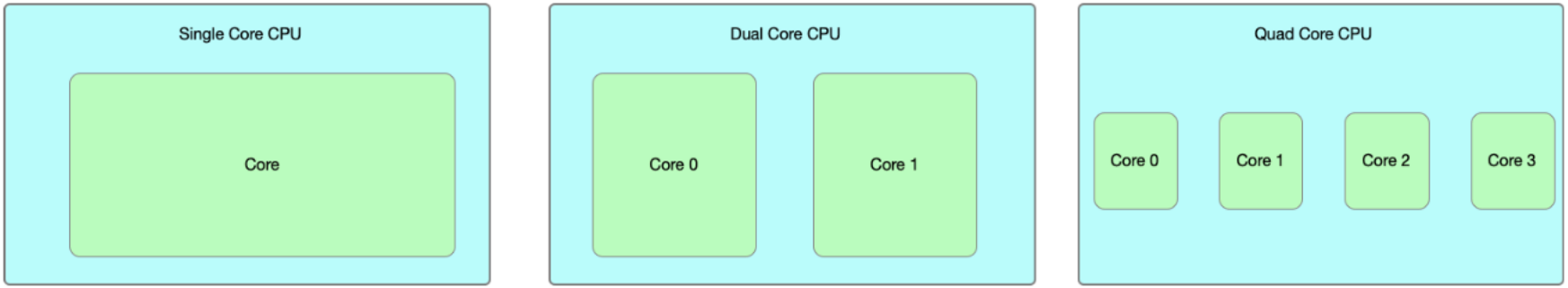
Diagram Source: https://commons.wikimedia.org/wiki/File:Moore's_Law_Transistor_Count_1970-2020.png

Moore's Law Implications

- Transistors are starting to suffer power consumption and integrity issues:
  - As the speed of transistors increases, their power consumption also increases; therefore this exceeds the power density that can be dealt with by air cooling.
  - Power consumption is dissipated as heat and when an integrated circuit gets hot it becomes unreliable.
  - Transistor gates have become too thin, affecting their structural integrity, which leads to currents starting to leak.
  - Physical manufacturing problems such as quantum tunneling (the inability to keep electrons contained beyond a certain thickness threshold) is also leading to a slow in single processor production
- Overall, its becoming impossible to continue to increase the speed of integrated circuits (hovering around 3.0GHz-3.7Ghz). Although with overclocking we've, seen this rise between 4.0GHz-5.0+GHz.

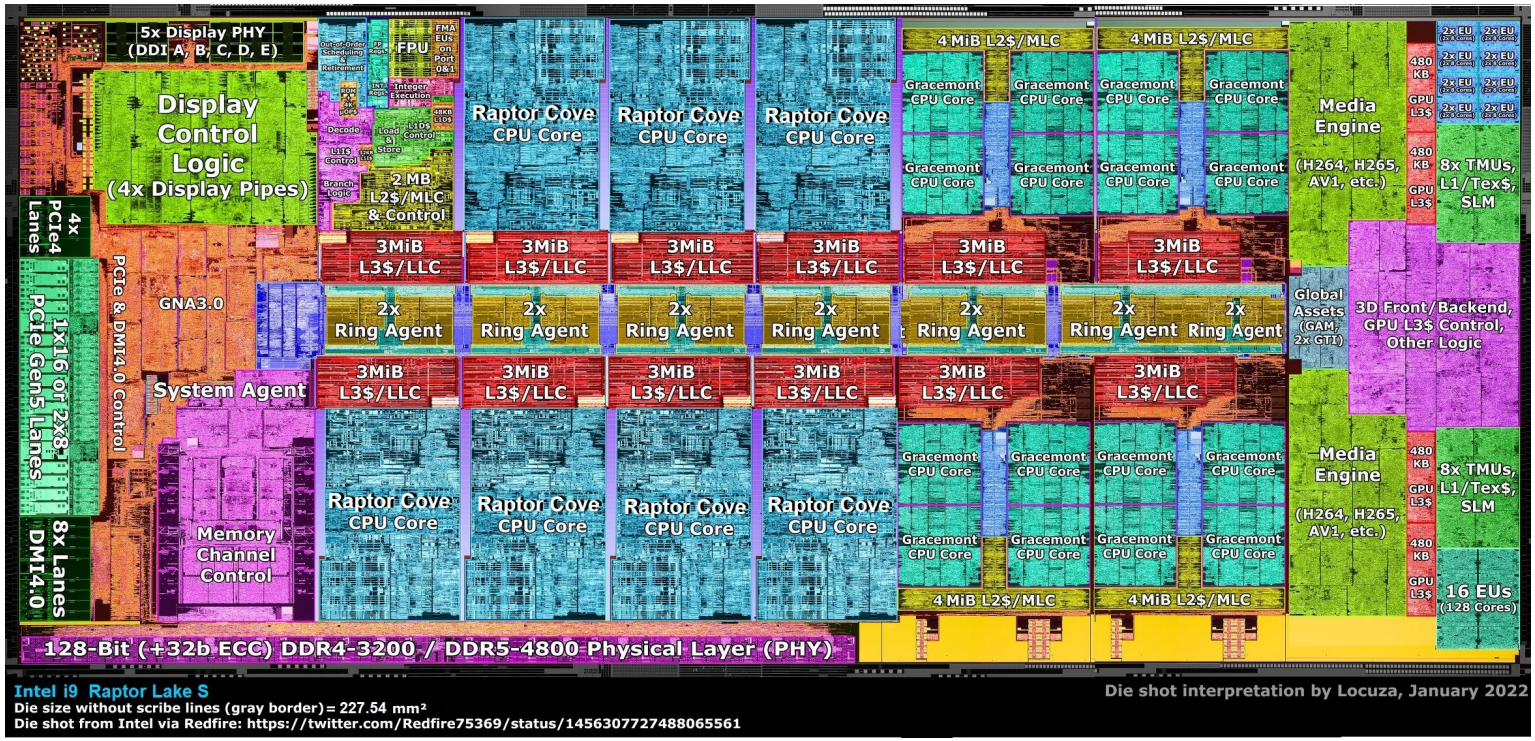
Multicore Architecture

- To manage CPU power dissipation, hardware manufacturers have moved to a multicore chip design
  - Continue to increase transistor density by having multiple and lower clocked processors on one chip (CPU).
- Terminology:
  - Core (also known as processor) : synonymous with central processing unit (CPU).
  - Multicore (also known as (also known as multiprocessor): more than one core on an integrated circuit.



Real-World Multicore Architectures: Intel's Alder Lake-S Silicon

- Die shot of the Raptor Lake- processors. These are Intel's newest (October 2022) processors used in the Core i9 , Core i7, and Core i5, and other Intel chips.
  - Up to 8 Raptor Cove performance cores (P-Cores)
    - Normal CPU cores that are large, and run at high clock- speed
  - Up to 16 Gracemont efficiency cores (E-Cores)
    - Small CPU cores that run at reduced clock speed



Real-World Multicore Architectures: Apple Silicon

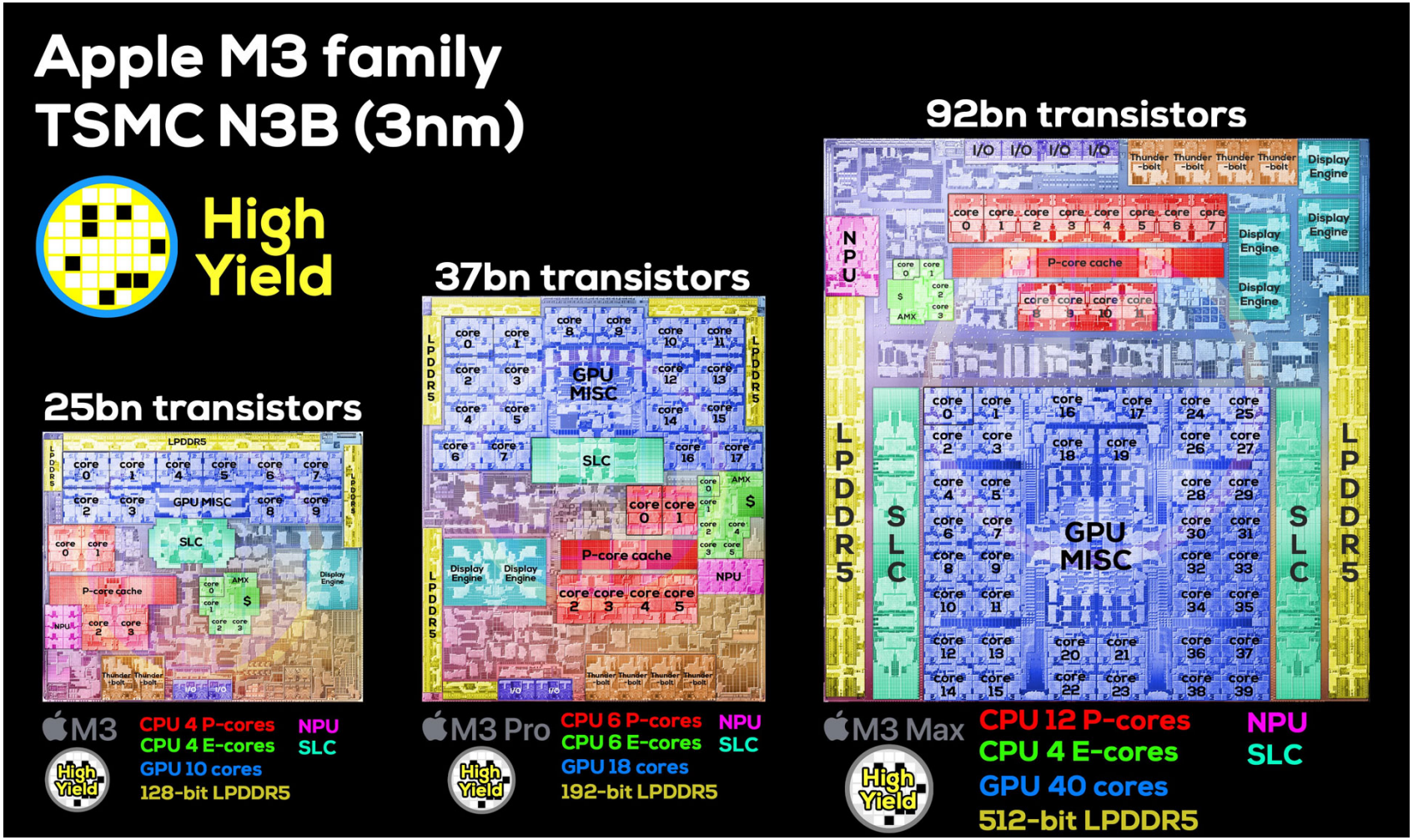
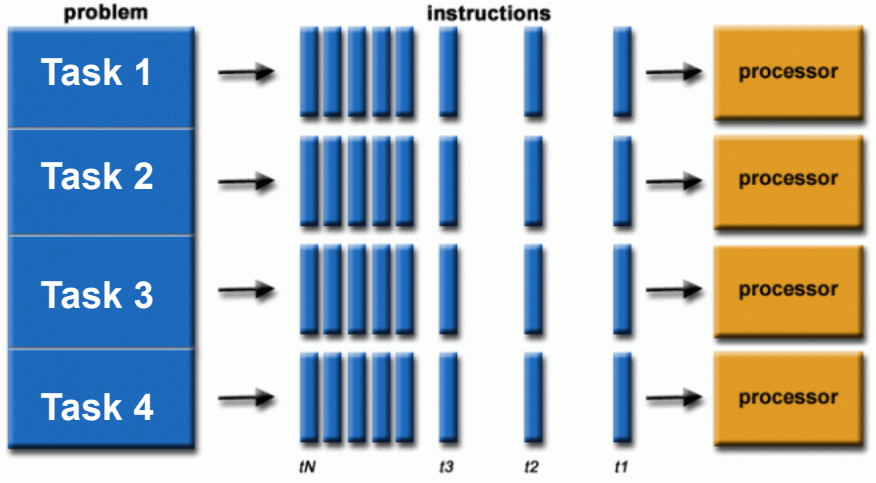


Diagram Source: https://www.tomshardware.com/news/annotated-apple-m3-processor-die-shots-bring-chip-designs-to-life

Programming on Multicore Architectures

- Programming on multicore chips requires programmers to break the problem into discrete parts (i.e., tasks) where each part can be ran on a separate core.
  - A task is a unit of work, where in this class, work is a series of program statements. Tasks can potentially to be executed in parallel but this is not a requirement.
  - At a high-level, a parallel program solves a problem that consists of multiple tasks running on multiple processors simultaneously .



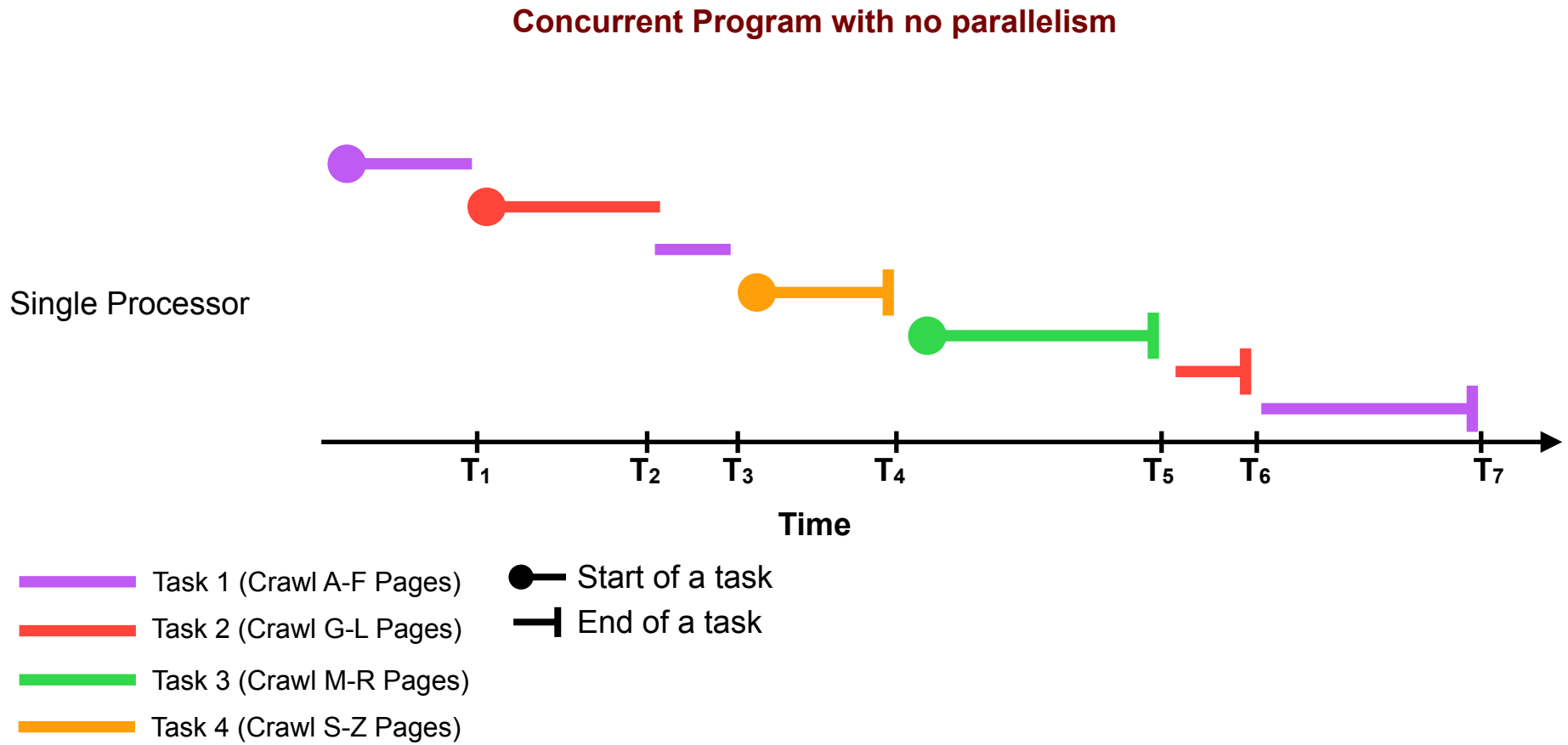
Concurrency vs Parallelism

- You may hear the terms concurrent and parallel referring to the same notion of executing a task at the same time; however, there is a key distinction between the two.
  - Concurrency is about dealing with tasks that are logically happening simultaneously. Two tasks are concurrent if they can be logically active at some point in time.
  - Parallelism is about dealing with tasks that are physically happening simultaneously.
- Concurrency provides a way to structure a solution to a problem that may (but not necessary) have tasks executed in parallel.

Crawler Example as a Concurrent Program

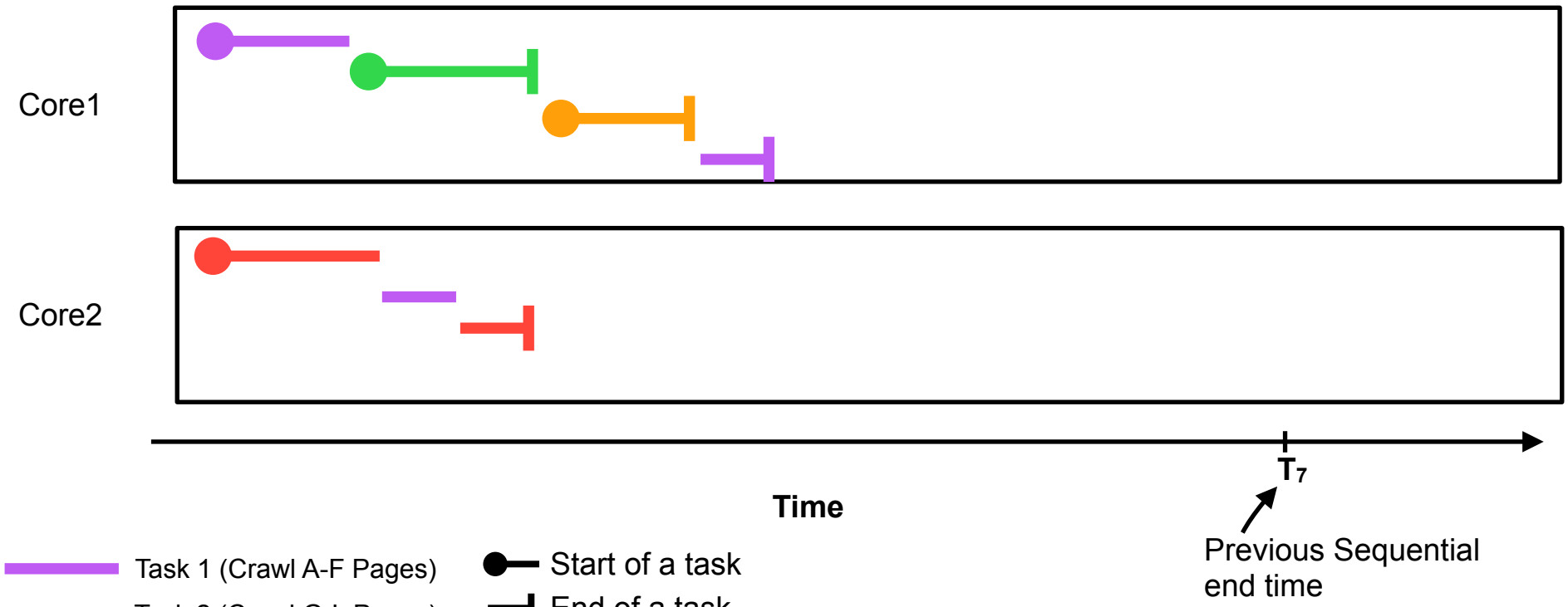
- One sequential solution could be to use a for-loop along with some code to crawl all 26 pages (A-Z) to produce the CSV file.
- One concurrent solution is to break the problem down into tasks where each task crawls a certain number of pages. The results of each task will be combined together to produce the final CSV file.
  - Task #1 - Crawl Pages A-F (Number of Pages: 6)
  - Task #2 - Crawl Pages G-L (Number of Pages: 6)
  - Task #3 - Crawl Pages M-R (Number of Pages: 6)
  - Task #4 - Crawl Pages S-Z (Number of Pages: 8)
- **This is not the only way to break this problem down into tasks**. This is one solution to many. We will discuss various other ways throughout this course.

Crawler Example as a Concurrent Program



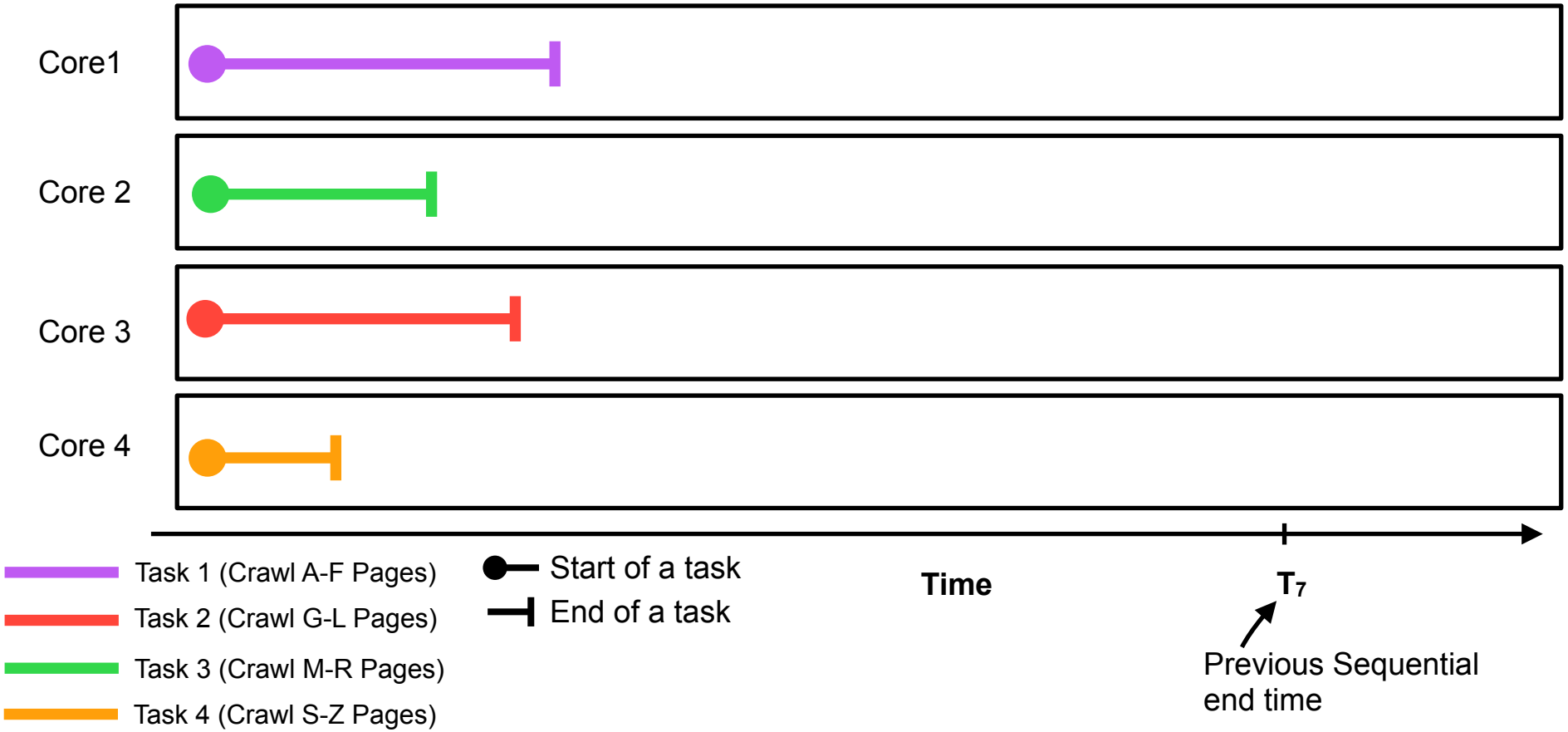
Crawler Example as a Concurrent Program (with Parallelism)

Parallel Implementation with Dual Core Chip



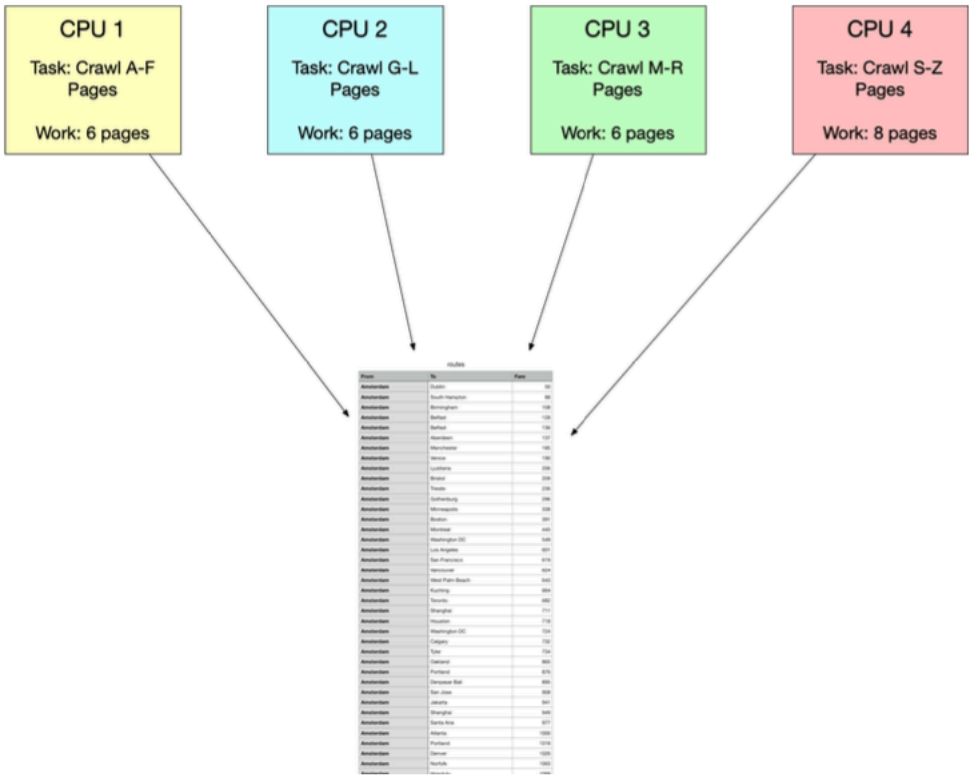
Crawler Example as a Concurrent Program (with Parallelism)

Parallel Implementation with Quad Core Chip



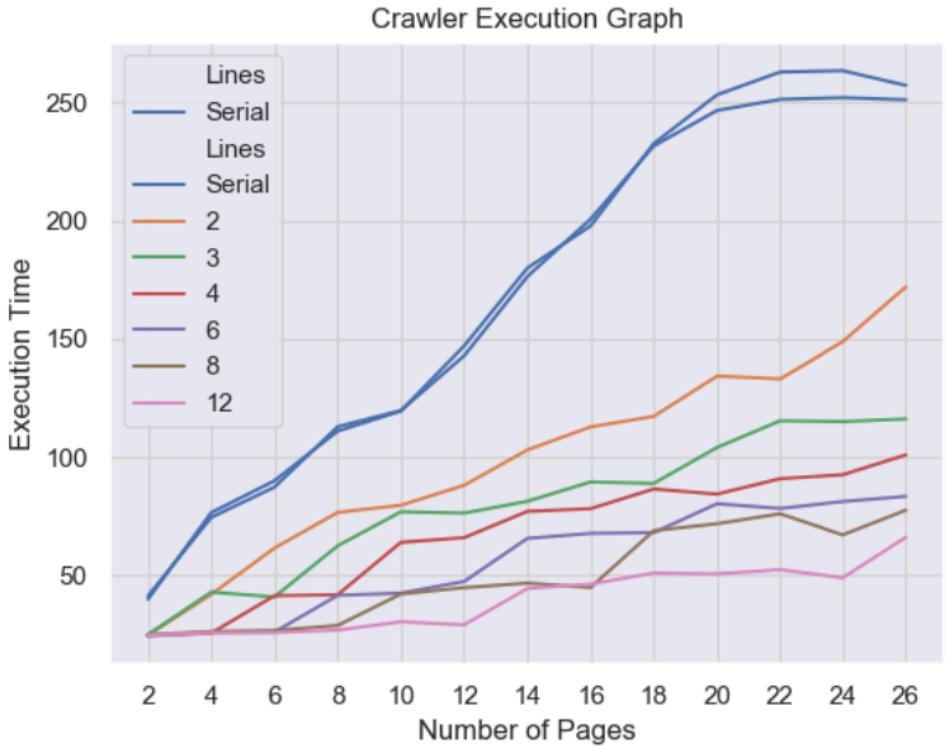
Crawler Example as a Parallel Program

- Demo: Crawler Program



Crawler Execution Time

- Execution time (in seconds) with a parallel component for our crawler:



What will you learn in this class?

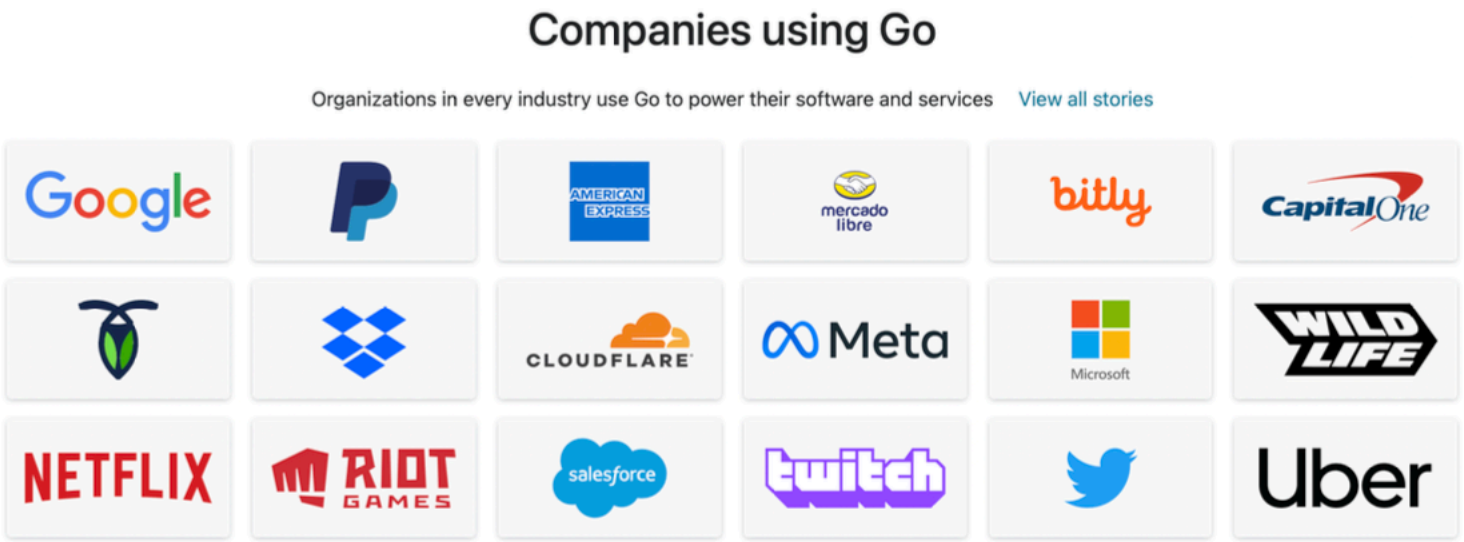
- Main objectives:
  - Best practices for designing and implementing parallel programs.
  - Synchronization mechanisms to maintain deterministic results
  - Schemes for processing tasks in parallel: algorithms, patterns and techniques to help with maintaining performance when scaling your application.
  - Understanding parallel architectures for fine tuning performance
- **Overall Goal:** Regardless of your programming language of choice (C, Java, C++, Python, etc.), you should be able to use the techniques, algorithms, patterns, and practices taught in this class and apply them to developing parallel applications in those languages.

Go Bootcamp

Design of Go

- Go was designed by Google (specifically Robert Griesemer, Rob Pike, and Kenneth Thompson in 2007) to solve problems that Google faces.
- Goals
  - Eliminate slowness
  - Inefficiencies
  - Maintain and improve scale
  - Make a concurrent language
- Types of problems Google faces?
  - C++ for servers, plus lots of Java and Python mixed in
  - Large employee base
  - Millions upon Millions of lines of code for various projects
  - Distributed build system
  - Millions of compute cluster machines, needing to perform tasks

Who uses Go?



- Diagram Source: https://go.dev

How are we going to learn Go?

- We are actually going to learn Go by going through examples shown on: https://gobyexample.com
- Constructs you should know:
    1. Values
    2. Variables
    3. Constants
    4. Importing
    5. Iteration statement: For, Range
    6. Selection statements: If/Else, Switch
    7. Core Data Structures: Arrays, Slices, Maps
    8. Functions
    9. Multiple Return Values
    10. Variadic Functions
    11. Pointers (VERY IMPORTANT!)
    12. Structs
    13. String Formatting and Functions
    14. Generics