

Parallel Programming

15-110 – Friday 03/13

Announcements

- **Hw4** due on Monday
 - Remember – it's extra-large! Don't leave it until the last minute!
- You can complete the midsemester surveys by the Hw4 deadline for bonus points
 - **Course Survey:** <https://bit.ly/110-s26-mid-course>
 - **TA Survey:** <https://bit.ly/110-s26-mid-tas>

Exam Announcements

- **Exam2** takes place **Wednesday 03/25**
 - Content covers Unit 2: Lists and Methods – Tractability
 - Unit 1 material also eligible, but not the focus
 - Same logistics as Exam1
 - Study materials now available:
<https://www.cs.cmu.edu/~110/assessments.html>

Learning Goals

- Recognize and define the following keywords: **concurrency**, **CPU**, **scheduler**, **multitasking**, **multiprocessing**, **parallel programming**, **data dependency**, and **shared resource**
- Calculate the **work** and **span** of a simple parallel algorithm
- Create **pipelines** to parallelize an algorithm by splitting **tasks** across cores

New Unit: Scaling Up Computing

New Unit: Scaling Up Computing

In the unit on Data Structures and Efficiency, we determined that certain algorithms may take a long time to run on large pieces of data.

In this short unit, we'll address the following questions:

- How is it possible for complex algorithms on huge sets of data (like internet search) to run quickly?
- How can we write algorithms that work across multiple computers, instead of running on just one machine?

Two Ways to Increase Efficiency

When we compute Big-O runtimes of programs, we count the number of abstract actions taken. On real computers those actions must be executed using real circuits, and we can influence the speed of those circuits through the design of the computer's hardware. (The same number of actions are taken – they can just happen a lot faster).

There are two ways we can easily speed up a computer:

- Increase the number of **transistors** on the computer
- Have the computer run actions **in parallel** instead of sequentially

Transistors Provide Electronic Switching



A **transistor** is a small device that makes it possible to switch electric signals between on/off. Transistors are a fundamental part of circuit design.

More transistors lead to more circuits, which leads to faster computers. But we're limited by the **size** of the transistors and the size of the computer; there's only enough space to add so many transistors inside a computer.

Over time, engineers worked to fit more and more transistors on computers. This meant that computers could get faster and faster every year as new advances were made.

Switch From Smaller Transistors to Concurrency

For a while, engineers were able to double the speed of computers every two years by increasing the number of transistors in a computer. However, around 2010 it became physically impossible to keep up this doubling rate because of physical limitations related to power and heat. Computers couldn't get faster unless we made them bigger.

This led to a different tactic: instead of speeding up computers by adding more transistors, we decided to speed up computers by **having them run multiple programs at the same time**. This is called **concurrency**.

To accomplish this, we had to change some of the hardware that makes computers run...

CPUs, Multitasking, and Multiprocessing

Central Processing Units Manage Computation

A **CPU** (aka **core**) is composed of lots of circuits that actually run programs, including:

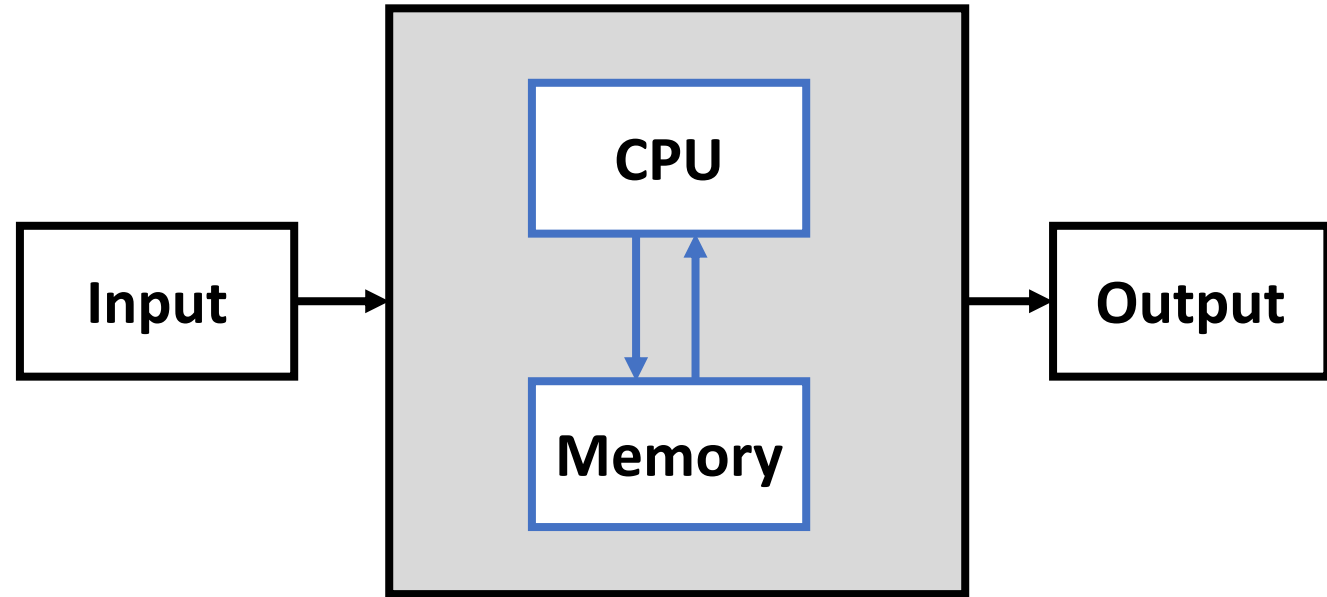
- **Control unit:** maps the individual steps taken by a program to specific circuits
- **Logic units:** individual circuits that can perform simple operations (like addition and multiplication)
- **Registers:** store information and act as temporary memory



CPU's Interact with Memory

Computers also have **memory** that the CPU can read from and write to. This is how the CPU can load instructions and save results.

Combine a CPU with memory and basic mechanisms for input and output, and you've got a simple abstract computer!

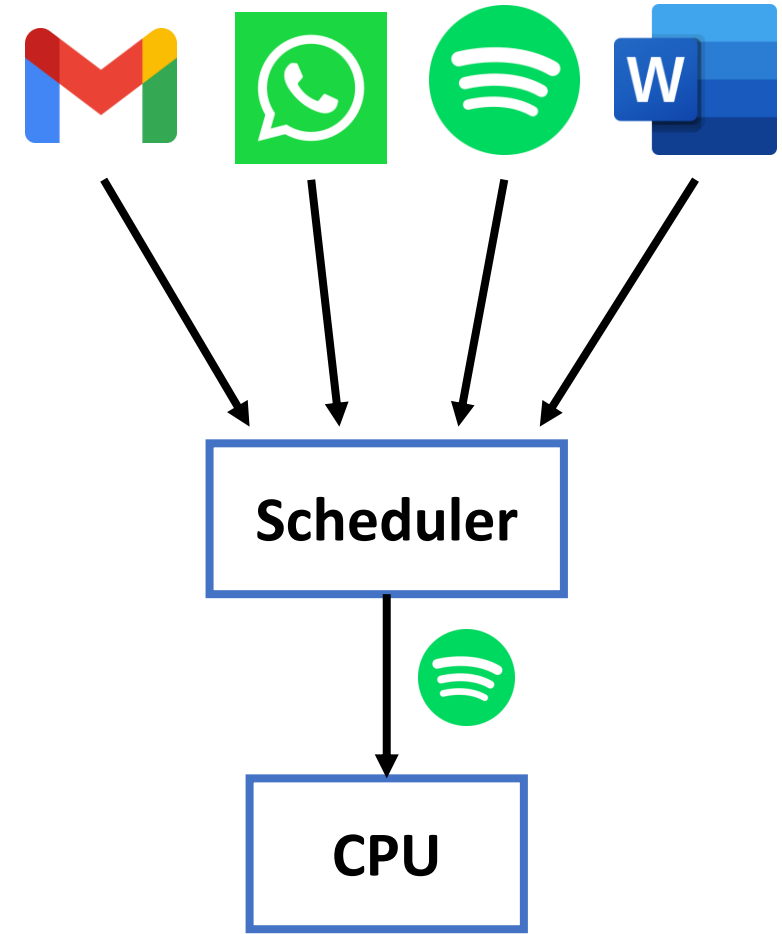


This organization of CPU, memory, input, and output is called a **von Neumann architecture**.

Schedulers Arrange Programs

When you use a computer, you likely have multiple applications open and running at any given moment. Each of these applications runs on one or more **processes** that need to execute instructions on a CPU. How does the computer decide what action to take next?

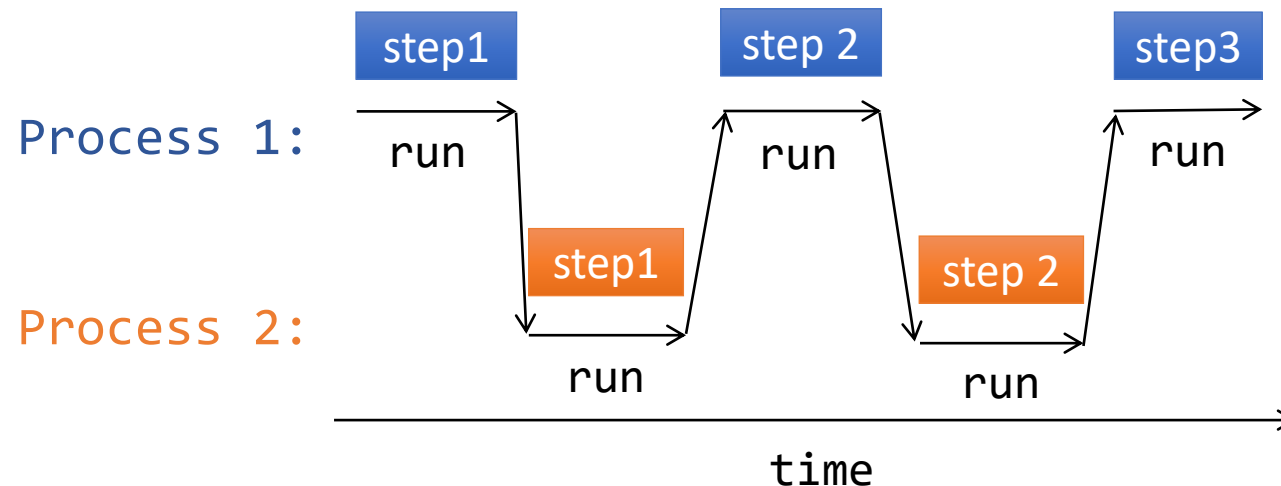
The **scheduler** is a computer component that takes information from the programs that are currently running and decides which program gets to use the CPU at any given moment.



Multitasking with a Scheduler

A scheduler could choose to let a program complete all of its actions before switching to the next program. But it usually doesn't! The scheduler can make programs **appear** to run at the same time by breaking each application's process into steps, then alternating between the steps rapidly.

If this alternation happens quickly enough, it looks like true concurrency to the user, even though only one process is running at any given point in time. This is called **multitasking**.

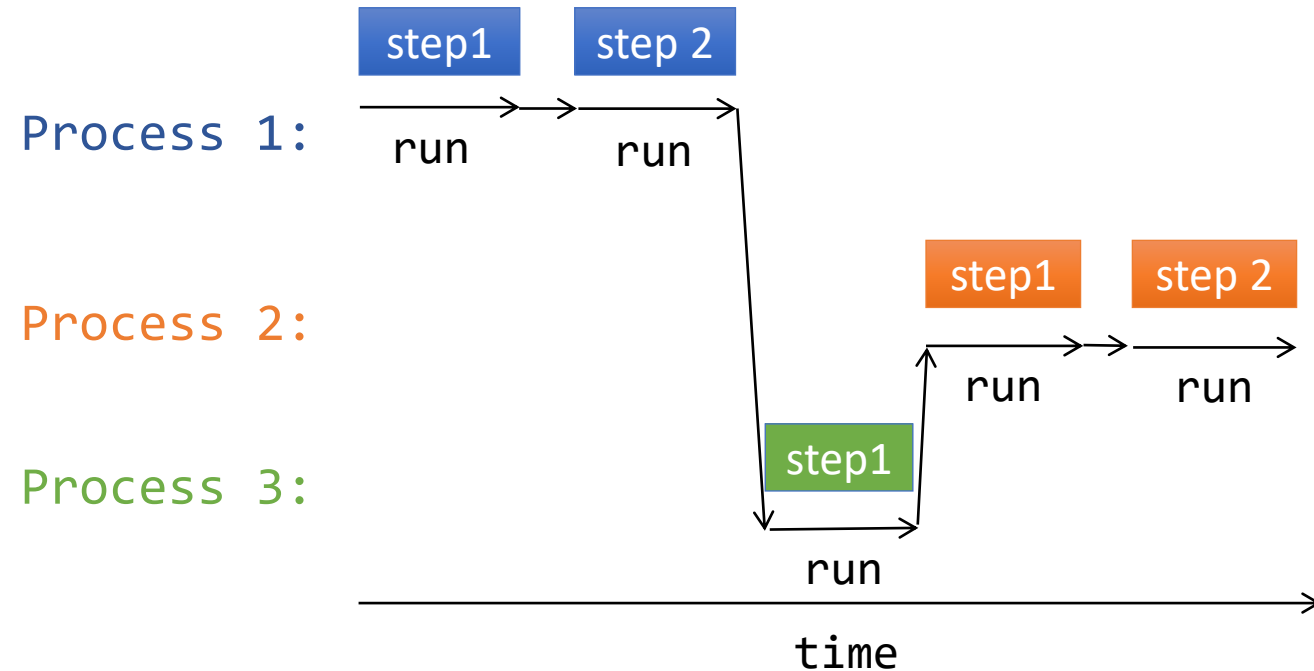


Schedulers Optimize Processes

How does the scheduler decide which process gets to run and for how long?

It considers multiple factors, including **response time** (how long it takes for a process to finish) and **throughput** (the total amount of work that can be completed).

The scheduler may choose to run several steps of one process before switching to another. It might even choose to put a process on hold for a long time if it isn't a priority or is just stalling while waiting for a user action.

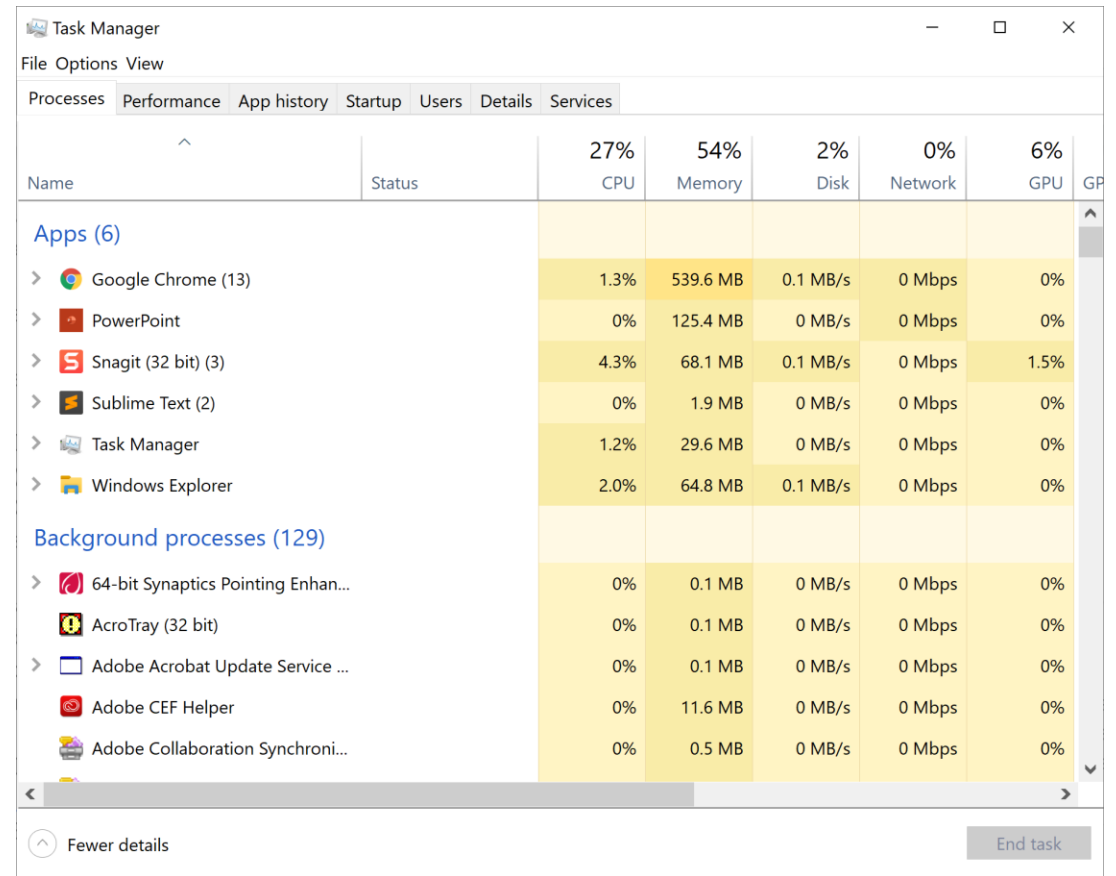


Your Computer Multitasks

Your computer uses multitasking to manage all the applications you run, as well as the background processes needed to make your computer work.

You can see all the applications your computer's scheduler is managing by going to your process manager (Task Manager on Windows, Activity Monitor on Mac, htop on Linux). You can even see how much time each process gets on the CPU!

You do: open your process manager now to see how much CPU time each application takes



Name	Status	27% CPU	54% Memory	2% Disk	0% Network	6% GPU	GP
Apps (6)							
> Google Chrome (13)		1.3%	539.6 MB	0.1 MB/s	0 Mbps	0%	
> PowerPoint		0%	125.4 MB	0 MB/s	0 Mbps	0%	
> Snagit (32 bit) (3)		4.3%	68.1 MB	0.1 MB/s	0 Mbps	1.5%	
> Sublime Text (2)		0%	1.9 MB	0 MB/s	0 Mbps	0%	
> Task Manager		1.2%	29.6 MB	0 MB/s	0 Mbps	0%	
> Windows Explorer		2.0%	64.8 MB	0.1 MB/s	0 Mbps	0%	
Background processes (129)							
> 64-bit Synaptics Pointing Enhanc...		0%	0.1 MB	0 MB/s	0 Mbps	0%	
AcroTray (32 bit)		0%	0.1 MB	0 MB/s	0 Mbps	0%	
> Adobe Acrobat Update Service ...		0%	0.1 MB	0 MB/s	0 Mbps	0%	
Adobe CEF Helper		0%	11.6 MB	0 MB/s	0 Mbps	0%	
Adobe Collaboration Synchroni...		0%	0.5 MB	0 MB/s	0 Mbps	0%	

Multitasking is Fake Concurrency

Multitasking is very useful, but it doesn't increase the speed of a complex algorithm. A single CPU can still only run one action at a time, so an algorithm is still limited to the rate designated by the design of the CPU itself.

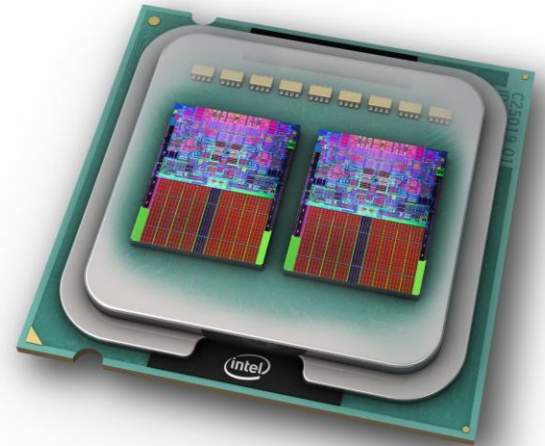
How can we speed up algorithms? We can **use multiple CPUs!**

Multiprocessing Runs Multiple CPUs

Multiprocessing is a method of concurrency where you run multiple actions **at the exact same time** on a single computer.

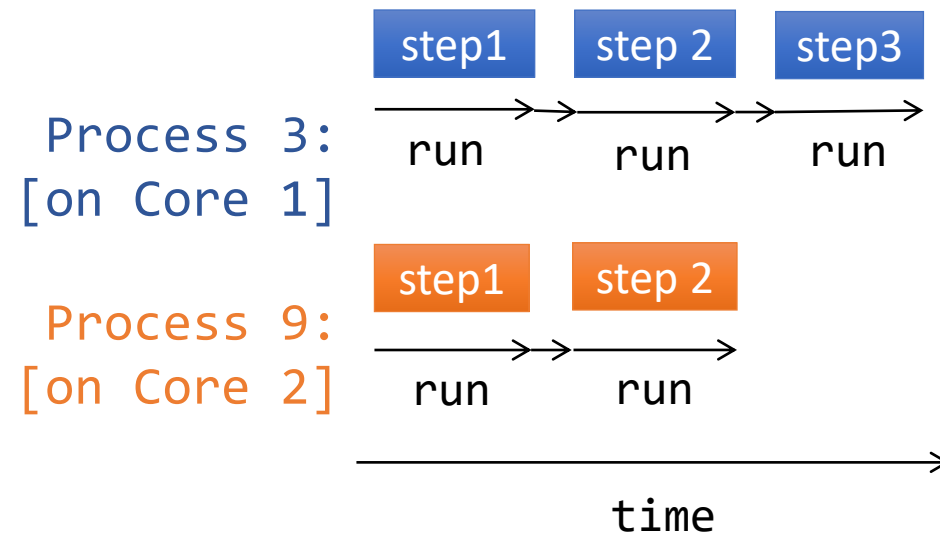
To make this possible, put multiple CPUs inside a single computer. Then the computer can send different actions to different CPUs at the same time.

If you have two CPUs instead of one, you can theoretically double the speed of your computer. With four CPUs, you could quadruple it!



Scheduling with Multiprocessing

When we use multiple cores and multiprocessing, we can run our processes **simultaneously** by assigning them to different cores.



Multiprocessing and Multitasking

The number of cores we have on a single computer is usually still limited. Most modern computers use somewhere between 2-16 cores. If you run more applications than you have cores, the cores use **multitasking** to make them appear to run concurrently.

You can check how many cores your own computer has! If you're on Windows, go back to the process manager and switch to the tab 'Performance'. If you're on a Mac, go to About This Mac > System Report > Hardware. On Linux, run `lscpu`.

Parallel Programming – Efficiency

Parallel Programming Divides a Program

Multiprocessing lets us run multiple applications at the same time, but it can also run a single process on multiple cores at the same time. This is called **parallel programming**, because the program is being run on multiple CPUs in parallel.

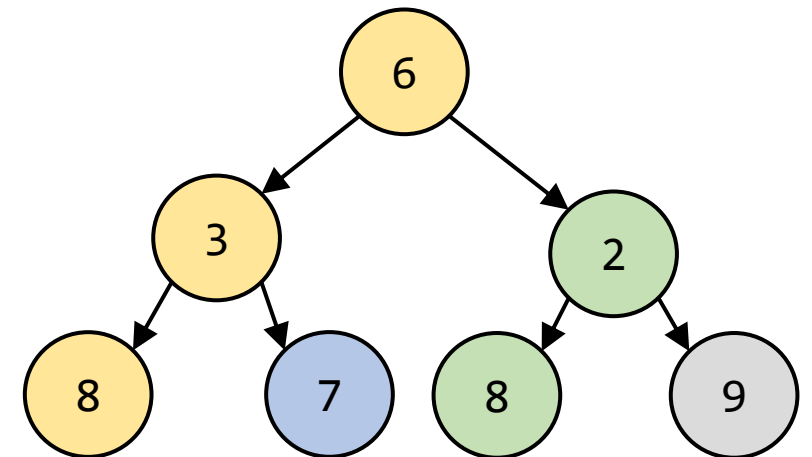
This should solve our efficiency problem! Now we can take an inefficient algorithm and just split it across a bunch of cores; that way, the work can get done a lot faster!

Summing a Tree Concurrently

Let's start with a simple example. We can write a function that can sum all the nodes in a tree. This would run in $O(n)$ time sequentially, since each node needs to be visited once. What if we do it concurrently?

We do two recursive calls in each recursive case (one on the left child, one on the right). Call the left child recursively on the current core and send the right child's call to a **new** core. This lets us do the two recursive calls **concurrently**. In our example to the right, this is shown using different colors for each core.

```
def sumNodes(t):  
    if t == None:  
        return 0  
    else:  
        total = t["contents"]  
        total += sumNodes(t["left"])  
        total += sumNodes(t["right"])  
        return total
```



Two Measures of Efficiency: Work and Span

When we want to determine the **efficiency** of a parallel algorithm, it helps to compare the total amount of work done to the 'observed' work done. To do this we'll use two new terms: **work** and **span**.

Work is the total number of actions taking place across all cores in the whole algorithm.

Span is the number of actions that take place in the longest chain of sequential (not parallel) actions. If we organize the processes optimally, this should be the number of actions that happen if we collapse together simultaneous actions.

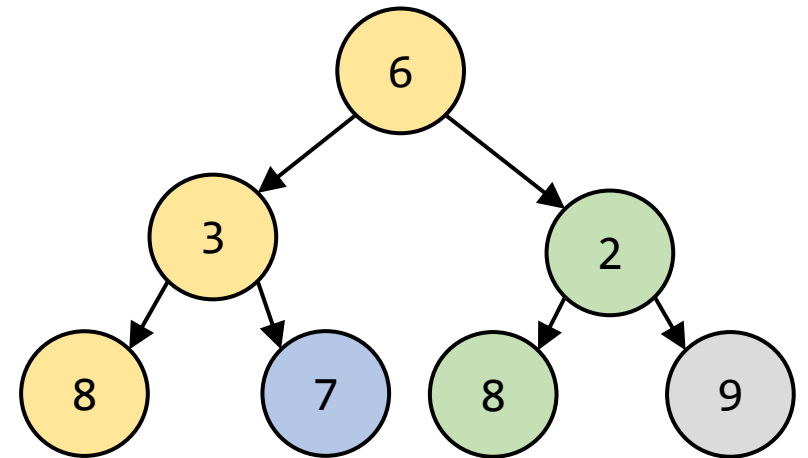
Work gives us the real amount of work that's happening, but span helps us determine how long it will take a parallel algorithm to run.

Efficiency of Tree Summing

What is the efficiency of our concurrent tree summing algorithm?

Work: we visit all 7 nodes in the tree. This is still $O(n)$, we need to visit every single node

Span: assuming we have as many cores as we want, the original core runs the longest and does one step per level of the tree. For this tree that's 3 nodes. If we assume the tree is balanced there will be $\log n$ levels, so this is $O(\log n)$.



Efficiency Gains

Going from $O(n)$ to $O(\log n)$ is a huge improvement! If we do this for every problem we solve, we can massively improve the efficiency of our code.

Unfortunately, it's not that simple...

Parallel Programming – Complications

Parallel Programming Gets Complicated

Parallel programming tends to be more difficult than regular programming. Programmers must take into account many new complicating factors, including:

- Data dependencies
- Shared resources
- Communication and synchronization between processes

Because this is so difficult, we won't write actual parallel programs in this class. But we will talk about common algorithmic approaches for writing parallel code.

Data Dependencies Complicate Splits

When designing a parallel program, we first ask the question: how much of the work can actually be split up into different processes?

A **data dependency** occurs when a step in a program refers to data created by a previous step. Step B cannot occur until after Step A runs, so they must run **sequentially**, not in parallel.

Consider a program like the one shown on the right. Every iteration of the loop uses the previous iteration's `total` value. It's not possible to split up the work of this program into completely separate processes.

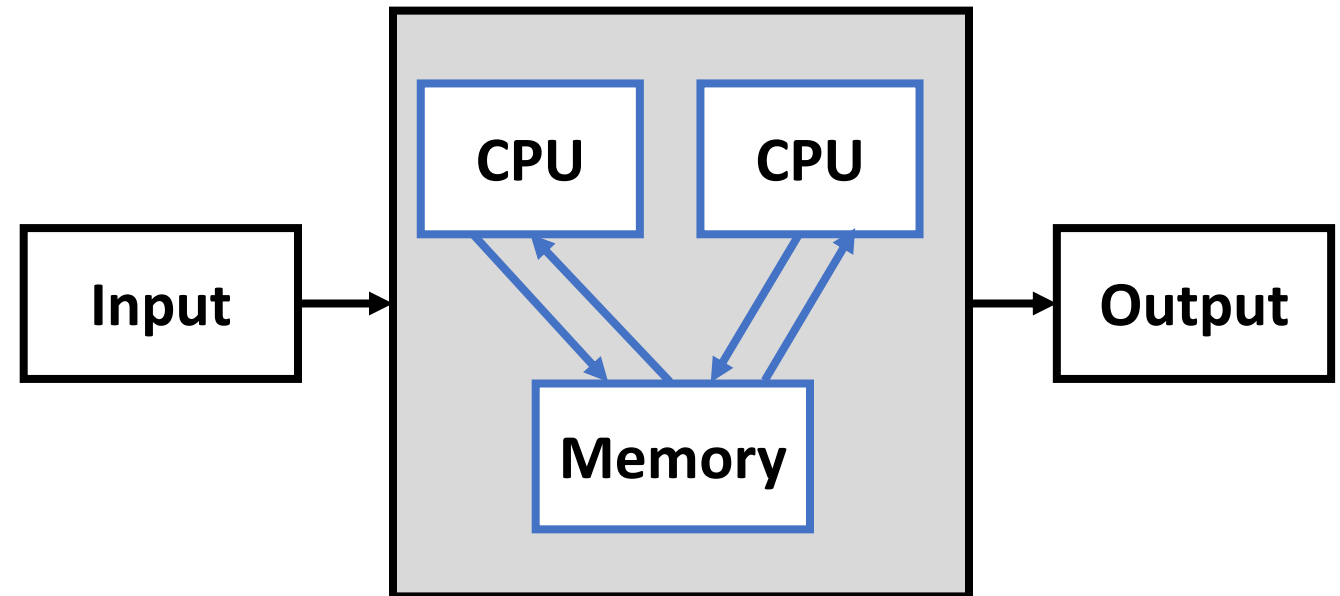
```
def addAll(lst):  
    total = 0  
    for num in lst:  
        total = total + num  
    return total
```

Processes Share Resources

Let's assume we're able to split up the work of an algorithm across multiple processes. Those processes might need to access a **shared resource** on a single machine.

For example, two different processes might want to access the same part of the computer's memory at the same time. They might both want to update the computer's screen or play audio over the computer's speaker.

We can't just let these processes update resources simultaneously – that might create garbled results.



Locking and Yielding Resources

To avoid this situation, processes put a **lock** on a shared resource when they access it. While a resource is locked, no other process can access it.

Then, when a process is done with a resource, it **yields** that resource back to the computer system, where it can be given to the next process that wants it.

Sidebar: if we want two processes to use a resource simultaneously, we usually use a third process to combine the actions together, and that third process is the one that accesses the resource. For example, if you listen to music while watching a lecture recording, your computer **mixes** the two audio tracks together and plays the combined result.



Communication and Synchronization

Finally, if a single algorithm uses multiple processes, that algorithm will need to have extra steps to manage those processes.

Sometimes processes need access to information generated on another process. The processes will need to **communicate** by passing a message from one to another.

Sometimes it's important that Step A on Process X happens before Step B on Process Y. The algorithm will need to **synchronize** the processes to ensure everything happens when it's supposed to.

All of this can reduce the efficiency of the algorithm, since we can't just run all processes simultaneously all the way to the end.

Generic Parallel Approaches

Writing parallel algorithms is hard, but there are some general parallel approaches that can be adapted for specific tasks.

We'll discuss one common approach today (**pipelining**) and another in the next lecture (**MapReduce**).

Pipelining

Pipelining is Task Parallelization

In **pipelining**, you start with an algorithm that repeats the same sequence of tasks over many different pieces of data.

The tasks of the procedure are split across different cores. Each core is like a single worker on an **assembly line**; when it is given a piece of data it executes its task, then passes the result to the next core.

Just like in an assembly line, the cores can run **multiple pieces of data simultaneously** by starting new computations while the others are still in progress.



Demo: Real-Life Pipelining

Let's compare pipelining to sequential work with a real-life race!

We need to generate ten greeting cards. We can divide the process of writing a greeting card into three sequential tasks:

1. Write 'Wish you were here!' on the paper
2. Fold the paper and put it inside an envelope
3. Seal the envelope

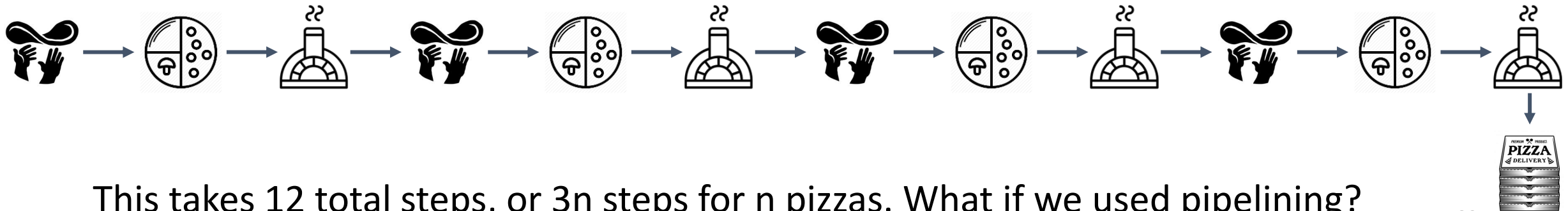
What happens if we have one process (person) complete all three tasks vs. having three processes (people) complete the tasks using a pipeline?

Sequential Pizza – 1 worker, 1 oven, 3n steps

Let's consider pipelining through the lens of line cooking. To make a pizza, we must:

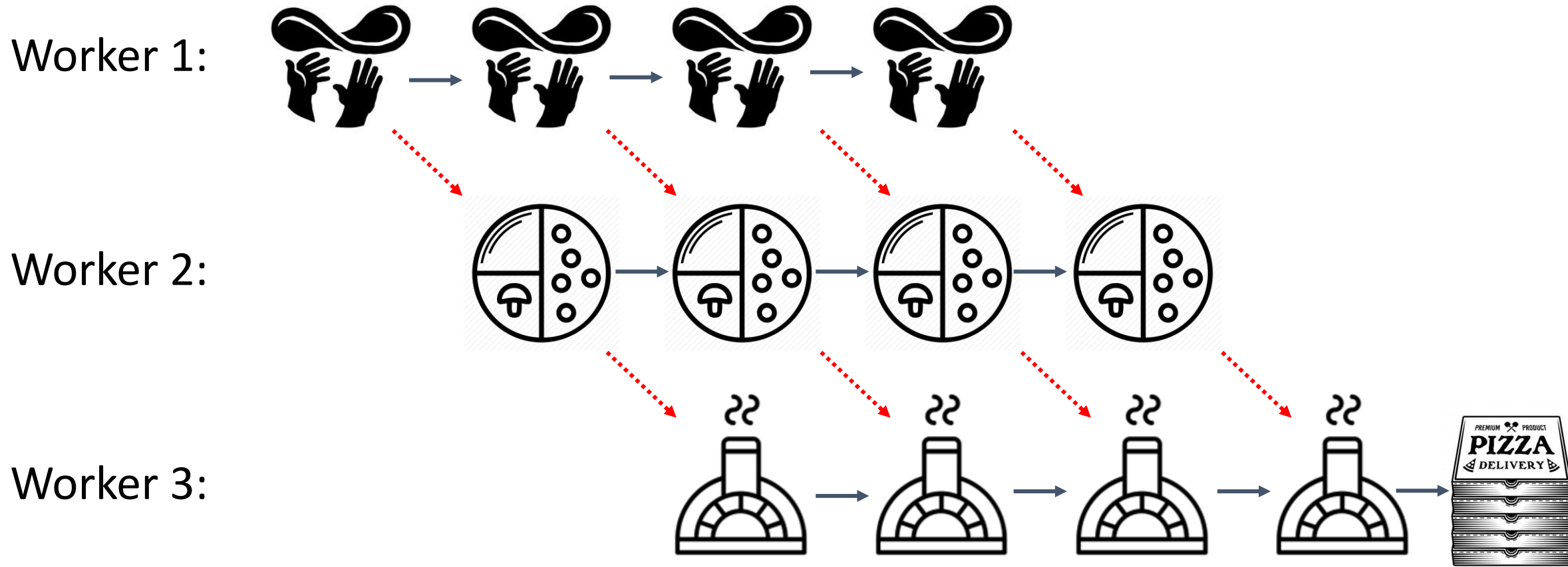
1. Flatten the dough
2. Apply the toppings
3. Bake in the oven

If we need to make four pizzas without parallelization, it will look like this:



This takes 12 total steps, or $3n$ steps for n pizzas. What if we used pipelining?

Pipelining Pizza - 3 workers, 1 oven, $n+2$ steps



Each worker has **one task**. #1 flattens dough, #2 arranges toppings, #3 bakes in the oven. The **work** is still 12 steps, but the **span** is just 6 steps. We go from $\sim 3n$ steps to $n+2$ steps.

Rules for Pipelining

When designing a pipeline, it's important to remember that **each task relies on the task that came before it**. You cannot start applying toppings until the dough has been flattened.

Additionally, the length of time that the pipelining process takes **depends on the longest task**. If flattening dough and applying toppings are fast (maybe 5 minutes each) but cooking in the oven is slow (maybe 20 minutes), the whole process will have to wait on the slowest task to conclude.

Benefits of Pipelining

Pipelining doesn't really change the Big O runtime of our algorithms; it only reduces work by a constant multiplier. But it's still useful when the number of shared resources is **limited**. For example, you probably use pipelining when doing laundry at home, because you have a limited number of washers and driers to work with.

In computer science, pipelining is used to increase the efficiency of certain operations, like matrix multiplication. It's also used in the Fetch-Decode-Execute cycle, which is how the CPU processes instructions.

Learning Goals

- Recognize and define the following keywords: **concurrency**, **CPU**, **scheduler**, **multitasking**, **multiprocessing**, **parallel programming**, **data dependency**, and **shared resource**
- Calculate the **work** and **span** of a simple parallel algorithm
- Create **pipelines** to parallelize an algorithm by splitting **tasks** across cores