Parallel Programming

15-110 – Monday 11/02

Learning Goals

- Recognize certain problems that arise while multiprocessing, such as **difficulty of design** and **deadlock**

- Create **pipelines** to increase the efficiency of repeated operations by executing sub-steps at the same time

- Use the **MapReduce pattern** to design and code **parallelized algorithms** for distributed computing

Designing Concurrent Algorithms

Last time, we discussed the four levels of concurrency used by computers: circuit-level concurrency, multitasking, multiprocessing, and distributed computing.

Today, we'll discuss how to design algorithms so that they can run concurrently. This is often referred to as **parallel programming**.

We won't actually write parallelized code in this class (apart from a bit of MapReduce code where the parallelization is provided for you), but we will discuss common problems and algorithms in the field.

Difficulties in Parallelization

Difficulty of Design

Parallel programming is more difficult than regular programming, as it forces us to think in new ways and adds new constraints to the problems we try to solve.

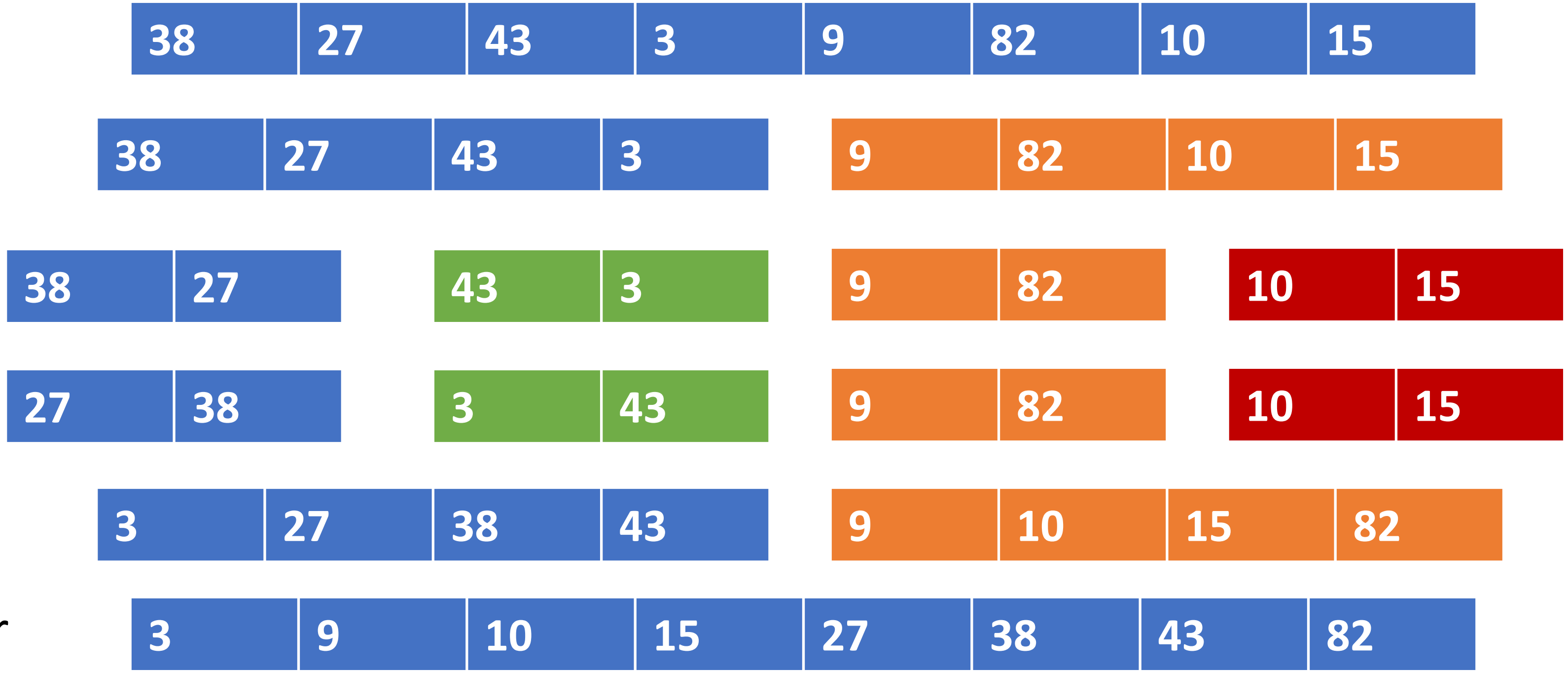
First, we have to figure out how to design algorithms that can be split across multiple processes. This varies greatly in difficulty based on the problem we're solving!

Making Merge Sort Concurrent

Let's start with an easy example – Merge Sort. The algorithm for Merge Sort adapts nicely to concurrency; instead of running mergeSort on the two halves of the lists sequentially, run them **concurrently**. Then send the results back to a single core to be merged.

Assume each color is a different core. How many steps does a single core take in the worst case?

The blue core is the worst case. It does n+n/2+n/4+... splits, then ...+n/4+n/2+n merges. The series n+n/2+n/4+... approaches 2n, so it does about 4n actions, or **O(n) work**.



38 27 43 3 9 82 10 15

38 27 43 3   9 82 10 15

38 27   43 3   9 82   10 15

27 38   3 43   9 82   10 15

3 27 38 43   9 10 15 82

3 9 10 15 27 38 43 82

Making Loops Concurrent

It's easy to make recursive problems like merge sort concurrent if they make multiple recursive calls. It's harder to think concurrently when writing programs that use loops.

We could plan to identify all the iterations of the loop and run each iteration on a separate core. But what if the results of all the iterations need to be combined? And what if each iteration depends on the result of the previous one? This gets even harder if we don't know how many iterations there will be overall, like when we use a while loop.

A bit later, we'll talk about how to use algorithmic plans to address these difficulties.

```
def search(lst, target):
    for item in lst:
        if item == target:
            return True
    return False
```

```
def getSum(lst):
    sum = 0
    for item in lst:
        sum = sum + item
    return sum
```

```
def powersOf2(n):
    i = 2
    while i < n:
        print(i)
        i = i * 2
```

Sharing Resources

The next difficulty of writing parallel programs comes from the fact that multiple cores need to **share individual resources** on a single machine.

For example, two different programs 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.

Locking and Yielding Resources

We can't just let two programs play audio or update the screen simultaneously- this will result in garbled results that the user can't understand.

For example, if one program wants to print "Hello World" to the console, and the other wants to print "Good Morning", the user might end up seeing "Hello Good World Morning".

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

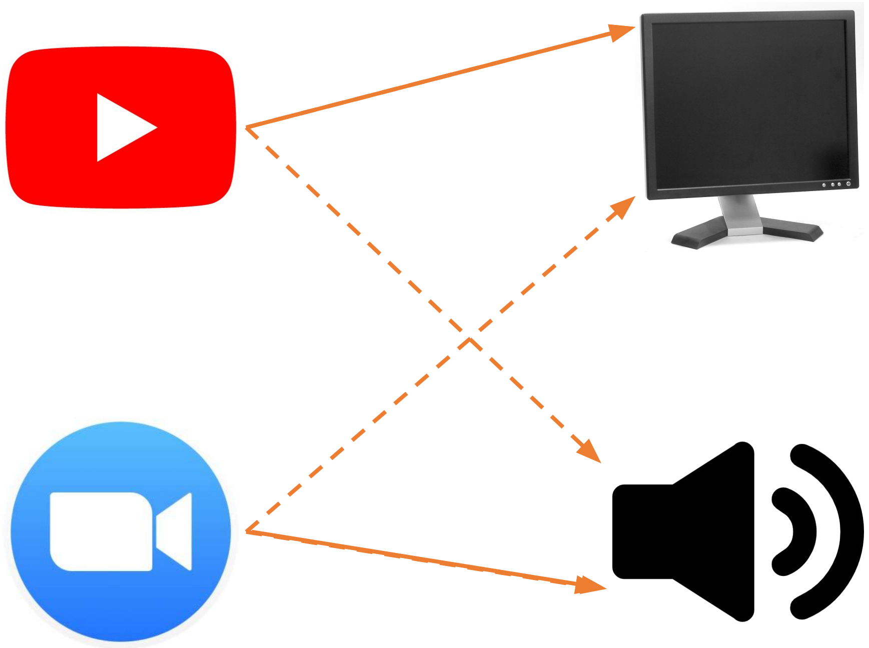
Then, when a program is done with a resource, it **yields** that resource back to the computer system, where it can be sent to the next program that wants it.

Deadlock Stalls the System

In general, this system of locking and yielding fixes most cases where programs might try to use a resource at the same time. But there are some situations where it can cause trouble.

Two programs, Youtube and Zoom, both want to access the screen and audio. They put their requests in at the same time, and the computer gives the screen to Youtube and the audio to Zoom.

Both programs will lock the resource they have, then wait for the next resource to become available. Since they're waiting on each other, they'll wait forever! This is known as **deadlock**.



Deadlock Definition

In general, we say that deadlock occurs when two or more processes are all waiting for some resource that other processes in the group already hold. This will cause all processes to wait forever without proceeding.

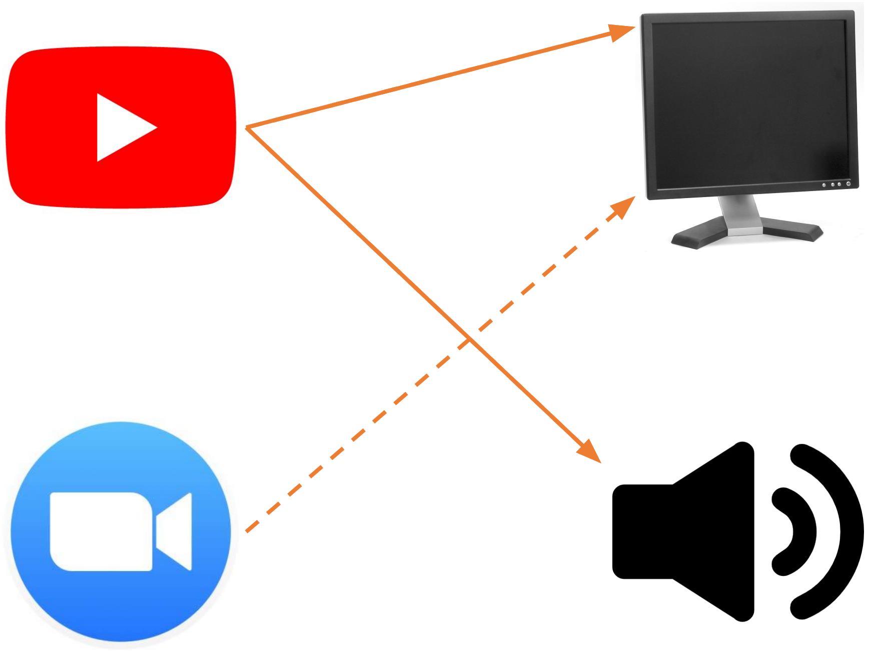
Deadlock can happen in real life! For example, if enough cars edge into traffic at four-way intersections, the intersections can get locked such that no one can move forward.

Fix Deadlock With Ordered Resources

In order to fix deadlock, impose an **order** that programs always follow when requesting resources.

For example, maybe Youtube and Zoom must receive the screen lock before they can request the audio. When Youtube gets the screen, it can make a request for the audio while Zoom waits for its turn.

When Youtube is done, it will yield its resources, and Zoom will be able to access them.

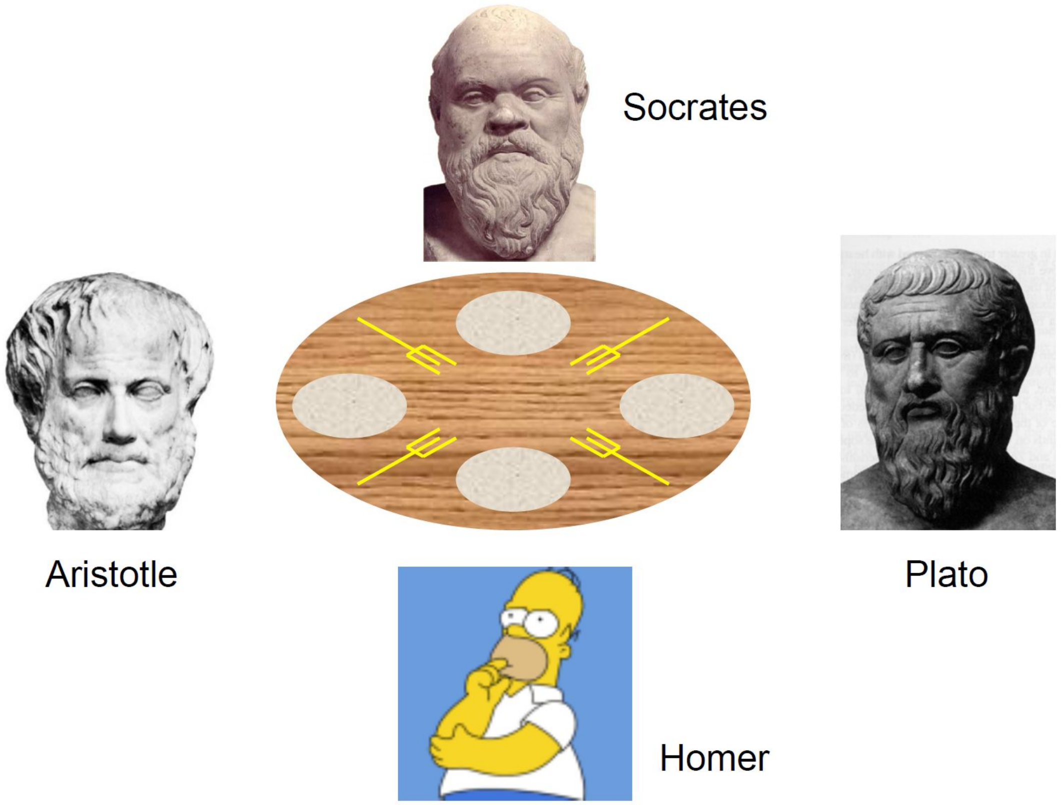


Activity: Dining Philosophers

Another example of deadlock occurs in the Dining Philosophers problem.

Several philosophers sit down at a circular table to eat. Each thinks for a while, then picks up their left fork, then picks up their right fork, then eats a bit. Then they put down the forks to think some more, then eat some more, etc.

You do: How can these philosophers get into deadlock? How can we solve that deadlock?



Some Processes Need to Communicate

We can't always guarantee that the processes running concurrently on a computer are independent. Sometimes, a single program is split into multiple tasks that run concurrently instead.

These tasks might need to share partial results as they run. They'll need a way to **communicate** with each other.

Processes Pass Messages to Share Data

Data is shared between processes by **passing messages**. When one task has found a result, it may send it to the other process before continuing its own work.

If one process depends on the result of another, it may need to halt its work while it waits on the message to be delivered. This can slow down the concurrency, as it takes time for data to be sent between cores or computers.

For example, in merge sort, once a core has finished splitting, it will need to wait for the result of the alternate core to merge the two halves of the list together.

Writing algorithms that can pass messages is tricky. We'll discuss two approaches that make it easier: **pipelining** and **MapReduce**.

Pipelining

Pipelining Definition

One algorithmic process that simplifies parallel algorithm design is **pipelining**. In this process, you start with a task that repeats the same procedure over many different pieces of data.

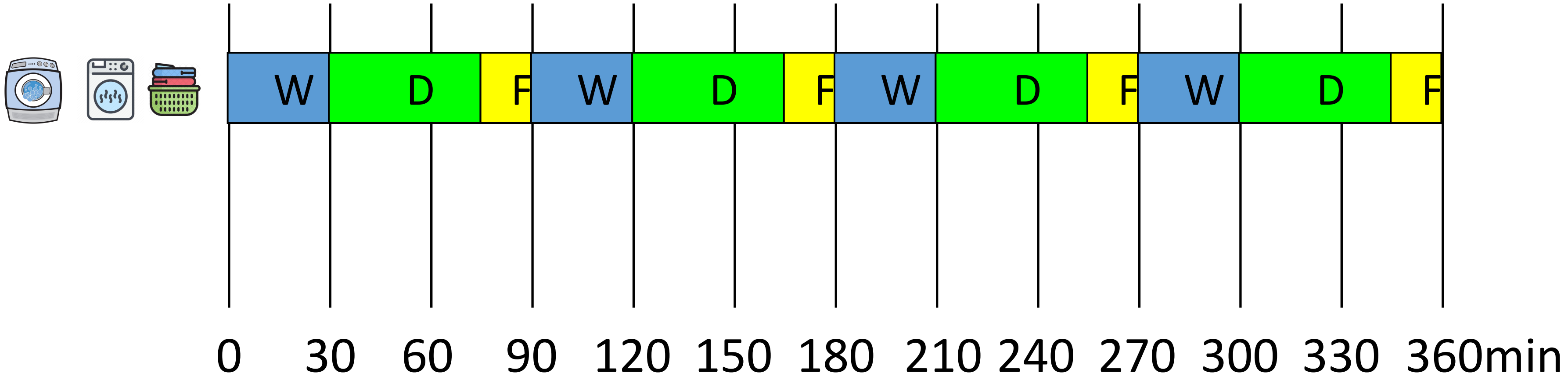
The steps of the process are split across different cores. Each core is like a single worker on an **assembly line**; when it is given a piece of data, it executes the step, then passes the result to the next core.

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.

Example: Laundry Without Pipelining

You probably already use pipelining when you do laundry. Let's look at an example where we assume you need to wash, dry, and fold several loads of laundry. Washing [W] takes 30 minutes; drying [D] takes 45; and folding [F] takes 15.

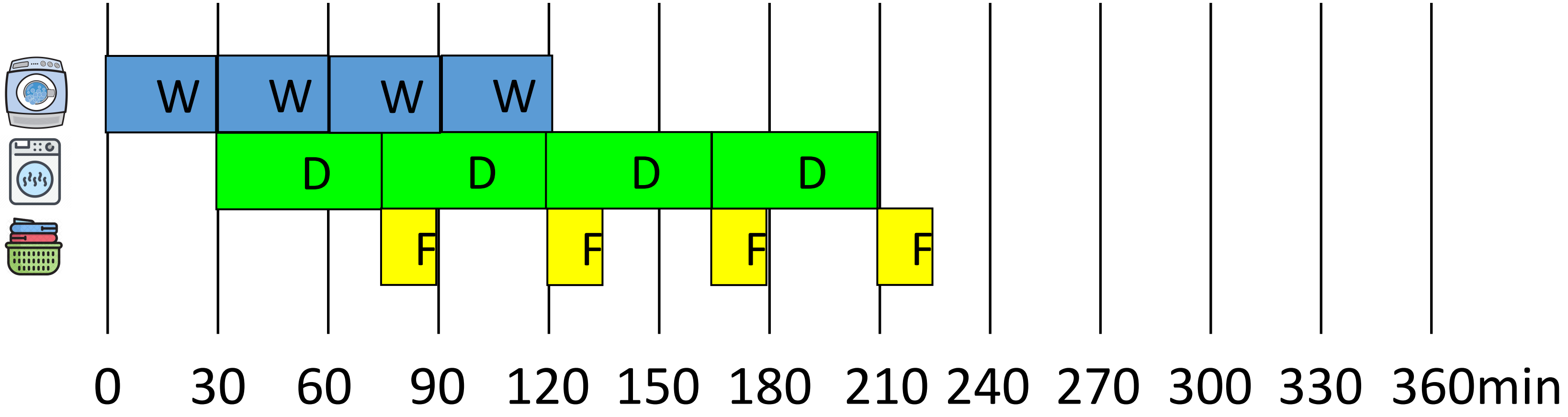
If you don't use pipelining, doing four loads of laundry takes **six hours**.



Example: Laundry With Pipelining

To use pipelining, split the three steps of the laundry process across three workers: the washer, dryer, and folder. Each worker has a lock on the **shared resource**.

With pipelining, four loads of laundry only takes **3 hours and 45 minutes**. Much faster!



Rules for Pipelining

When designing a pipeline, it's important to remember that **each step relies on the step that came before it**. You cannot start drying the laundry until it has finished being washed.

Additionally, the length of time that the pipelining process takes **depends on the longest step**. Since drying takes 45min, the folding must wait for drying to finish before it can start.

Activity: Design a Pipeline

The process of writing a thank-you card has three sequential steps: **Writing** the note [**10min**], **Adding** the address to the envelope [**6min**], and **Stuffing** the envelope [**6min**]. Because you hate writing thank-you cards, you've decided to hire two helpers (your younger siblings) to help with the work.

You need to **write all the notes yourself**, to make sure they're personalized, but you can outsource the other tasks to the helpers once the card has been written

By yourself, you can write 2 full thank-you cards in an hour (plus part of a third). If you use pipelining and the three workers (yourself + two helpers), **how many completed thank-you cards can you make in an hour**?

Hint: try drawing this out the way we drew out the washer/dryer/folder example, but with writer/adder/stuffer as the three roles.

Pipelining in Computer Science

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

Pipelining is often combined with **multiprocessing**, to split the operations being performed across multiple cores. This helps ensure that no core goes unused.

MapReduce

MapReduce Organizes Concurrency

Another popular algorithm for organizing parallelized programs is called **MapReduce**. Instead of breaking up a procedure's steps across different cores, this algorithm takes a **large data set** and breaks up the data itself across the cores.

A MapReduce algorithm is composed of three parts.

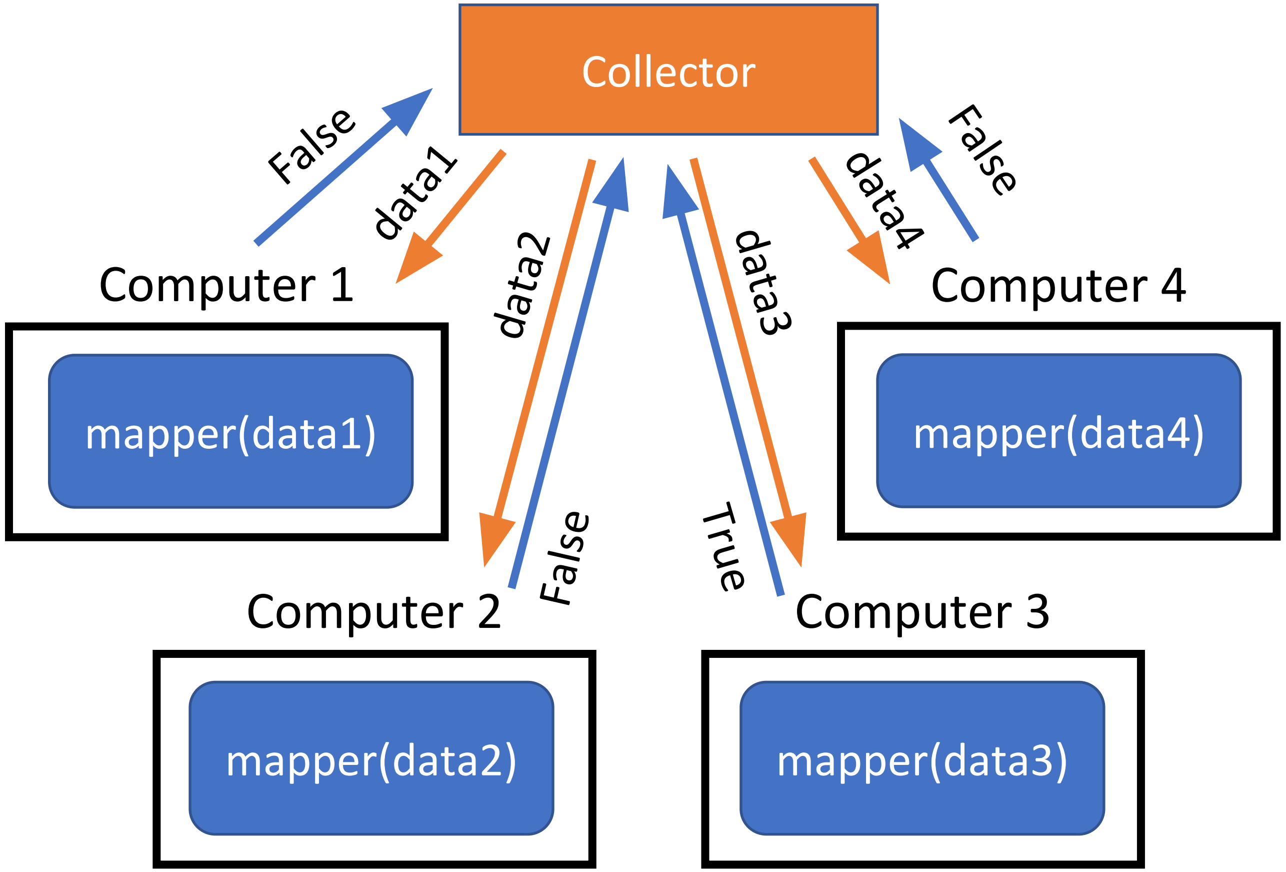
- The **mapper** takes a piece of data, processes it, and finds a partial result
- The **reducer** takes a set of results and combines them together
- The **collector** moves data through the process and outputs the final result
  - We won't program this part; the MapReduce framework does it for us

MapReduce Example: Search – Mapper

Let's say we want to search a book for a specific word. How can we split up this task?

First, divide the book into many small parts- maybe one page per part. Send each page to a different computer.

Each computer runs its copy of the **mapper** on its page. It returns True if it finds the result, and False otherwise. These results are sent back to the collector **collector**.

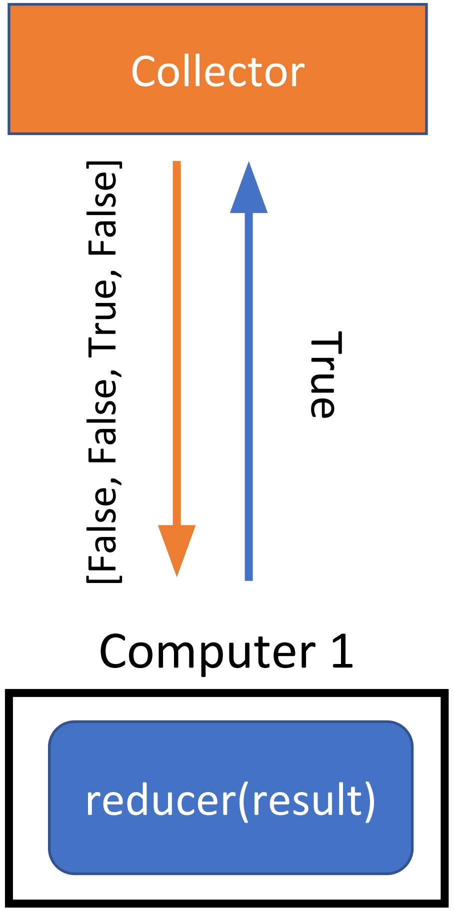


MapReduce Examples: Search – Reducer

Once all the mappers have returned their results, the collector puts them all in a list, and sends that list to the **reducer(s)**. The reducer combines the results together in some way.

There can be more than one reducer if there are lots of results to combine, or if we're checking multiple things (like searching for more than one word). For now, we'll just use one.

Our reducer will check all of the results and return True if any of them are True.



Coding MapReduce

We've provided a version of the MapReduce collector on the course website that uses multiprocessing to run the algorithm on several cores at the same time.

That makes implementing MapReduce easy- we just need to write code for the mapper and the reducer.

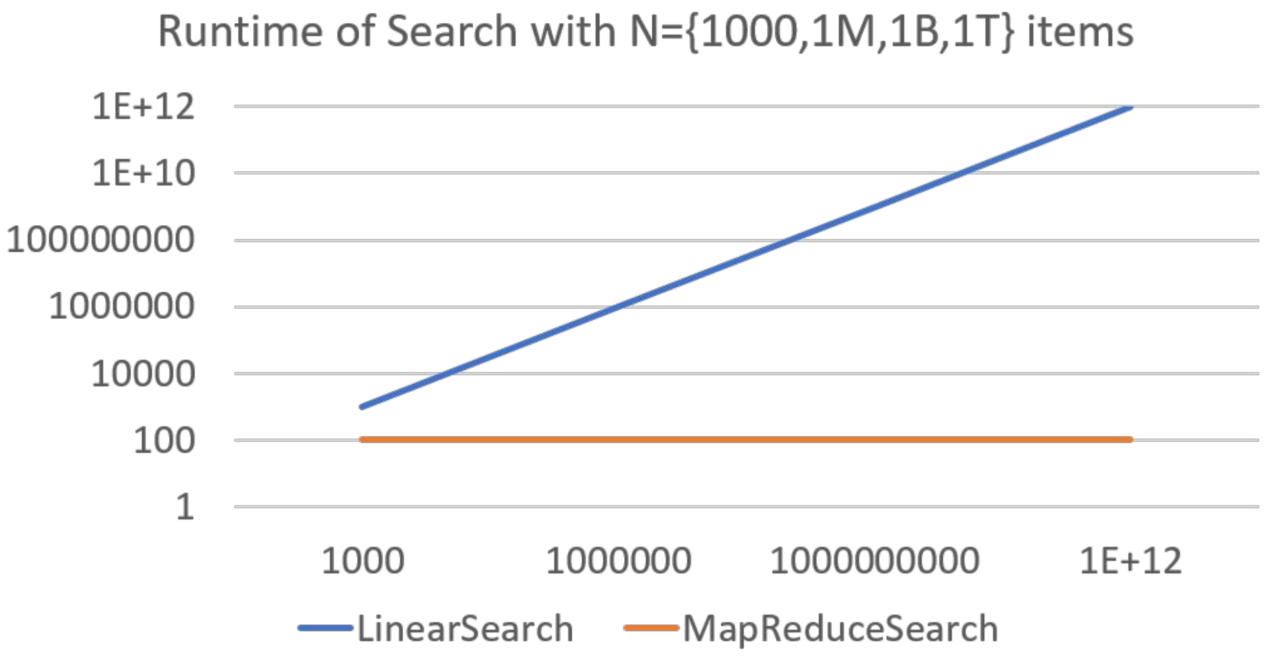
```
# Assume the page is a string
def mapper(s, target):
    words = s.split(" ")
    for word in words:
        if word == target:
            return True
    return False

# If the word is on any page, return True
def reducer(lst):
    for pageResult in lst:
        if pageResult == True:
            return True
    return False
```

MapReduce Efficiency

MapReduce can process huge data sets and get results quickly because it takes a list of length N and breaks it up into **constant-size parts**.

The core assumption is that we have enough computers to make the data pieces really small. If we process 10 million data points with 100,000 computers, each computer only needs to handle 100 data points.

This is similar to the logic behind hashing!



Another Example: Counting

What if we instead wanted to count the number of words across all of Wikipedia?

First, break up the data- maybe each Wikipedia entry goes to a computer.

The **mapper** can take a single page and count all the words on it.

The **reducer** can take a list of numbers and return their sum.

Activity: MapReduce the Class

Let's use MapReduce to determine how many students in 15-110 belong to each school.

We'll assign everyone to a breakout room with ~10 people in it. In the breakout room, designate one person as the notetaker (the **mapper**). That person must tally how many people in the room are in each of the 7 CMU schools (CIT, CFA, Dietrich, Heinz, MCS, SCS, and Tepper).

When we rejoin the main room, the notetaker will report their results to the chat. The instructor will then act as the **reducer** and combine the results.

Learning Goals

- Recognize certain problems that arise while multiprocessing, such as **difficulty of design** and **deadlock**

- Create **pipelines** to increase the efficiency of repeated operations by executing sub-steps at the same time

- Use the **MapReduce pattern** to design and code **parallelized algorithms** for distributed computing

- **Feedback:** https://bit.ly/110-feedback