Parallel Programming

15-110 – Monday 10/25

Announcements

- Hw4 was due **today**

- Check5/Hw5 released
  - Note that Check5 only has a written component – no programming part

- **Code Review #2!** Look for a Piazza post today/tomorrow morning with instructions to sign up for a slot
  - **IMPORTANT:** to give TAs enough time to prepare, we're closing sign-ups for slots on **Friday 6pm**, not an hour before each slot begins. Sign up early!

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 **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 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 us), but we will discuss common problems and algorithms in the field.

Difficulties in Parallelization

Difficulty of Design

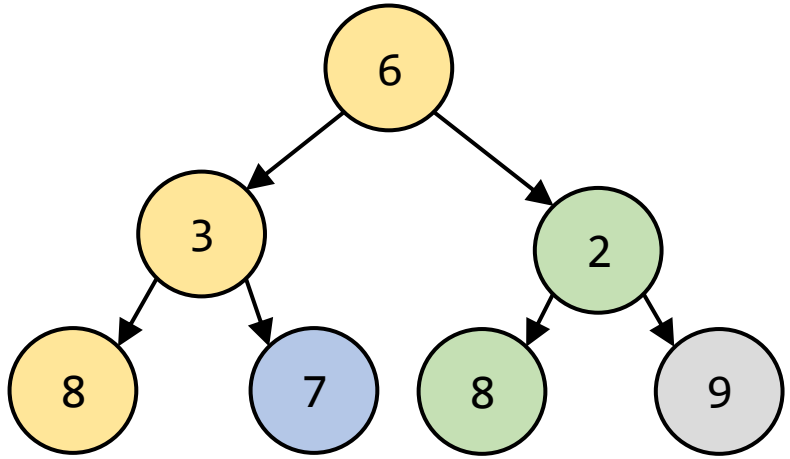
Parallel programming is more difficult than regular programming because it forces us to think in new ways and adds new constraints to the problems we try to solve.

First, we must 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!

Summing a Tree Concurrently

Let's start with a simple example. We showed in class how to 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. What if we do it concurrently?

We make up to 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, but 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.



How many **time-steps** does this take? Consider the original core, which does the most steps. This will only do one call per level of the tree; if the tree is balanced, that's log n levels. **Concurrent tree-summing is O(log n)!**

Making Loops Concurrent

It's easy to make recursive problems like tree-summing 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 update a resource 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.

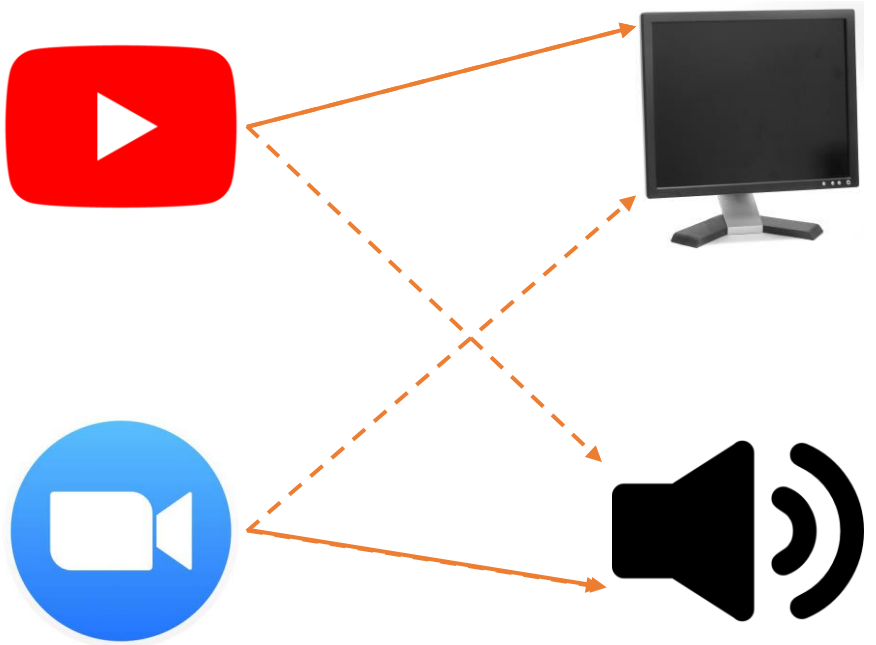
Sidebar: if we want two programs to use a resource simultaneously, we usually use a third program to combine the actions together, and that third program is the one that accesses the resource. For example, if you listen to music while watching a lecture recording, your computer **mixes** the two audio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

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**.

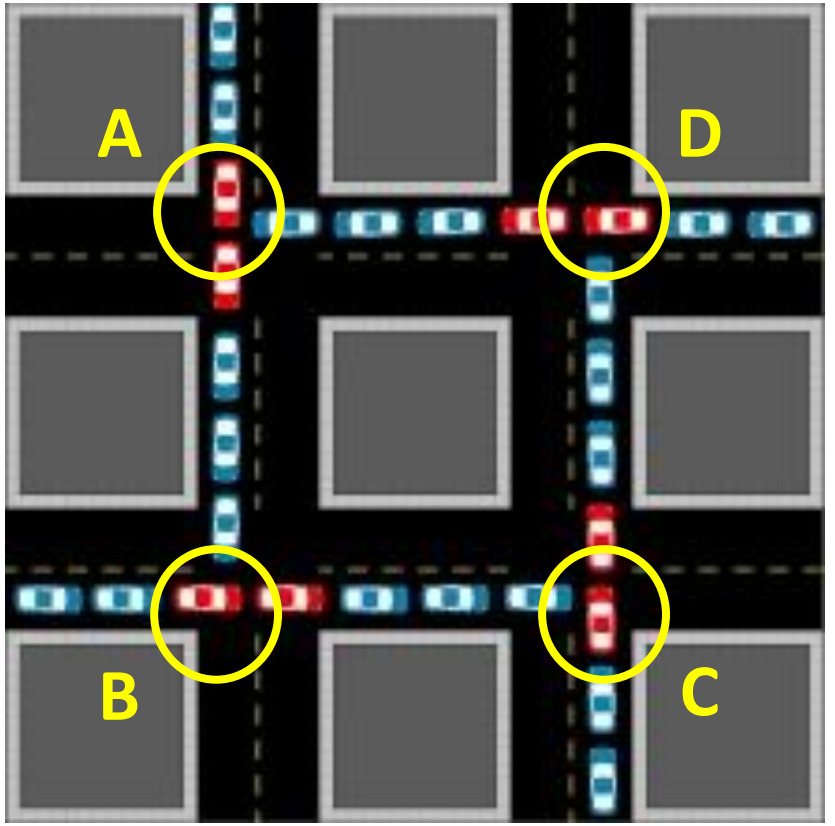


Deadlock Definition

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.

In the example to the right, each direction of traffic needs two of the intersection spots, but only has one. All four directions are blocked as a result.

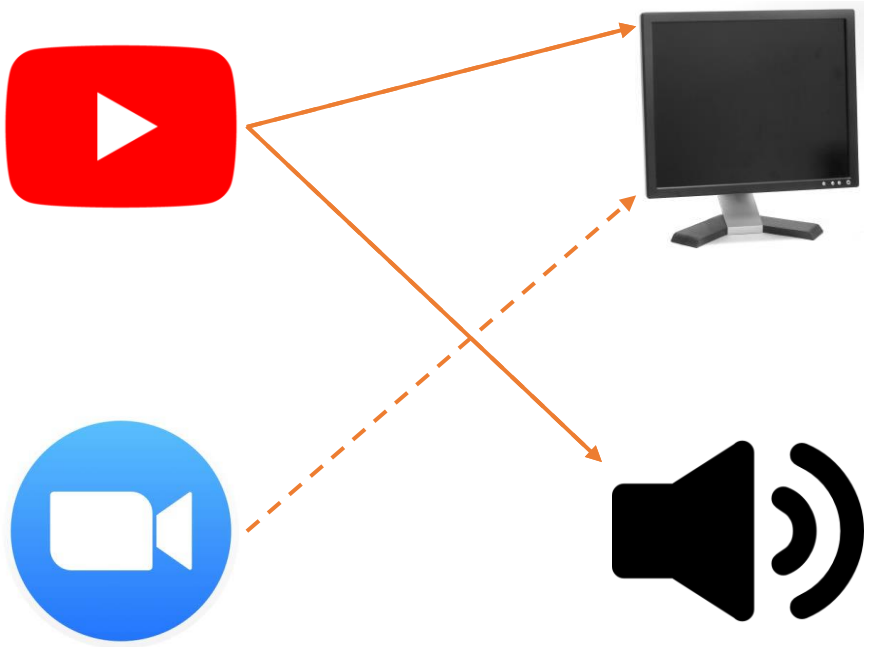


Fix Deadlock With Ordered Resources

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.



Some Processes Need to Communicate

We can't always guarantee that the processes running concurrently on a computer are independent. If a single program is split into multiple tasks that run concurrently instead, those tasks might need to share partial results as they run. They'll need a way to **communicate** with each other.

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. **Example:** in tree-summing, a core will need to wait for both calls to finish before it can sum the results.

Pipelining and MapReduce

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

The core idea behind pipelining is that you can parallelize an algorithm by splitting up the **algorithm** into a series of **consecutive steps**.

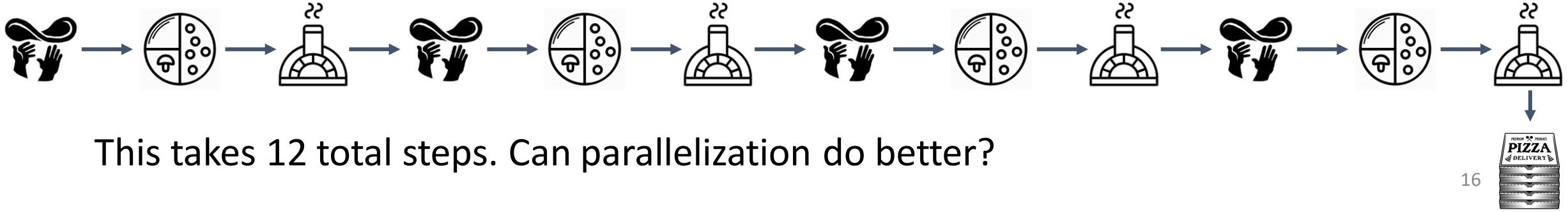
The core idea behind MapReduce is that you can parallelize an algorithm by splitting up the **data** into **many many small parts**.

Message Passing Example: Line Cooking

Let's introduce our two algorithms 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. Can parallelization do better?

Pipelining

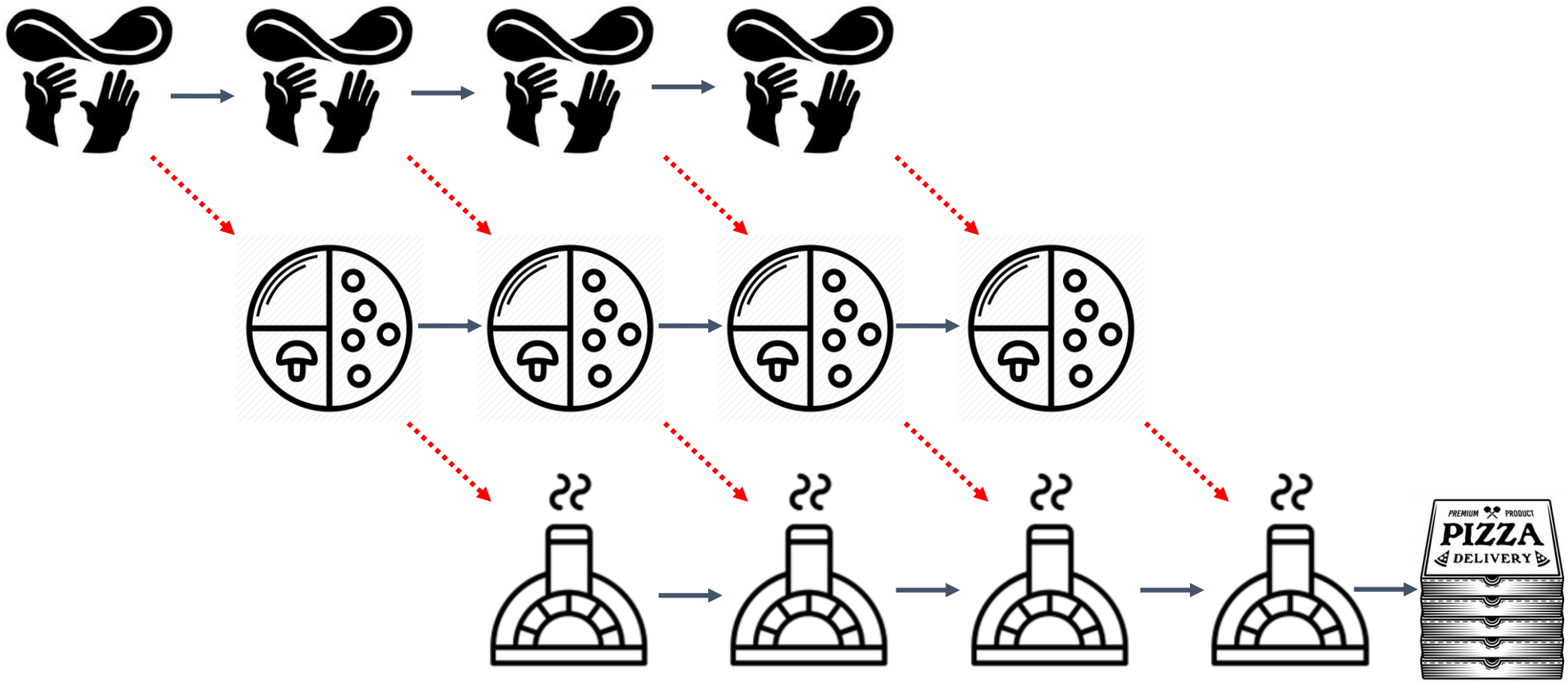
Pipelining Definition

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 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 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.

Pizza Pipelining - 3 workers, 1 oven, 6 time-steps



Each worker has **one task**. #1 flattens dough, #2 arranges toppings, #3 bakes in the oven.
There are still 12 total steps, but there are only 6 **time-steps**.

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 applying toppings until the dough has been flattened.

Additionally, the length of time that the pipelining process takes **depends on the longest step**. 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 step to conclude.

Benefits of Pipelining

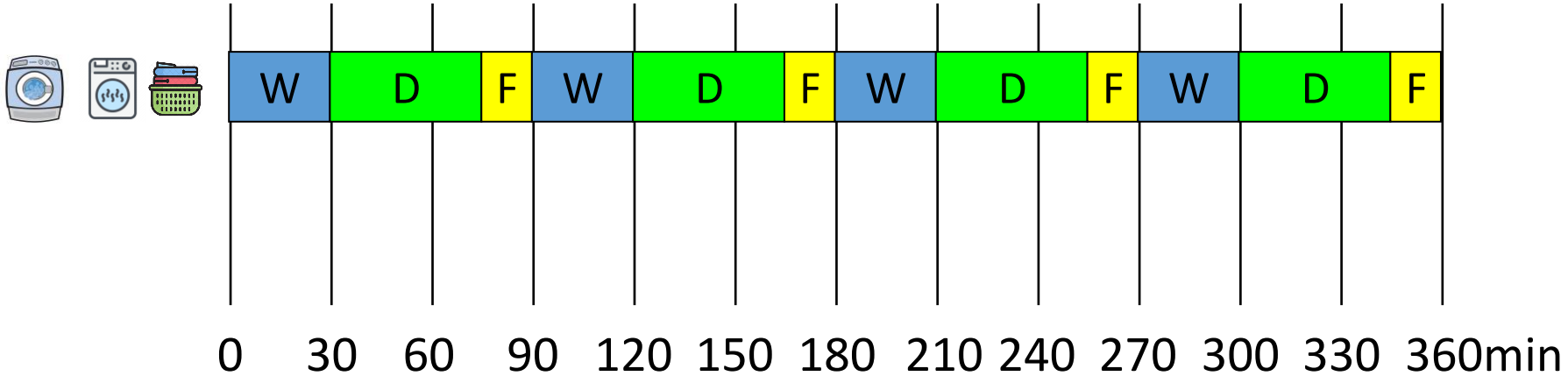
Pipelining is most useful when the number of shared resources is **limited**. For example, in pizza-making we may have only one oven; using pipelining ensures that we are constantly making use of the oven without wasting time.

Pipelining is also useful for tasks that require setup time, but then can run many times without further setup - maybe for flattening, the cook only has to clean the counter and flour it once.

Another 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; folding [F] takes 15.

If you don't use pipelining and wait until a load of laundry is folded before starting the next one, doing four loads of laundry takes **six hours**.

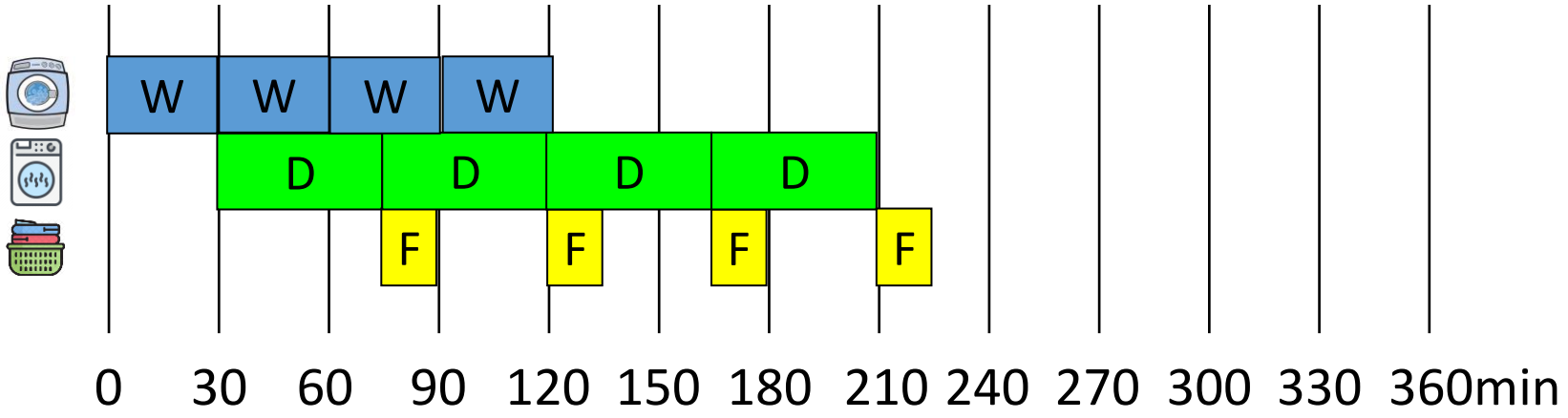


Example: Laundry With Pipelining

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!

[In reality, you alternate between these tasks and the machines do the work; you just start the machines. So the **machines** are the workers in this scenario.]



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.

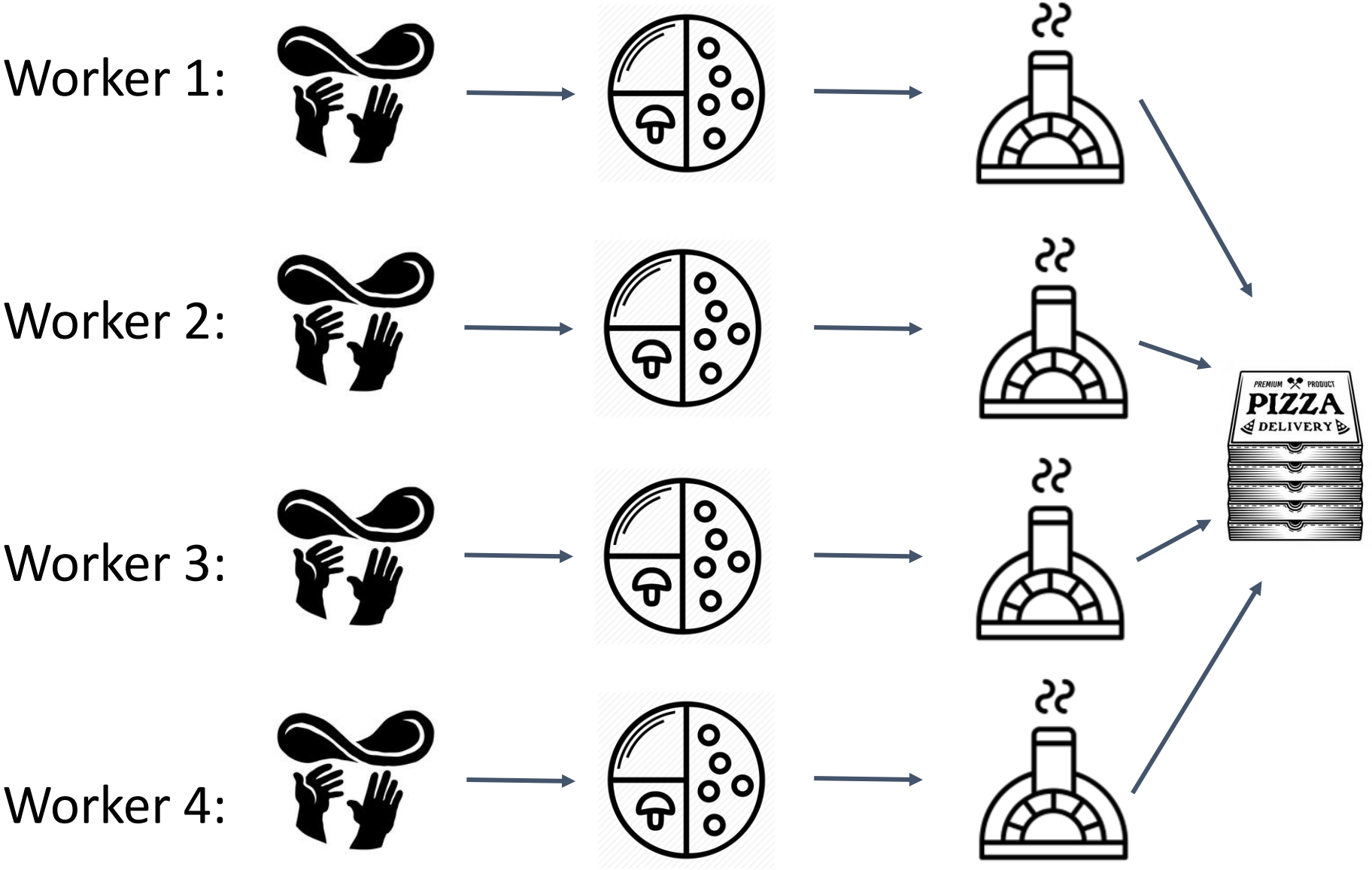
MapReduce

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.

This is a really effective approach if you have a lot of cores to work with (like in distributed computing). It's also a great approach for any problem over **big data** – that is, giant data sets that take far too long to process sequentially.

MapReduce - 4 workers, 4 Ovens, 3 time-steps



Each worker makes one pizza instead of doing one task repeatedly.

If we have infinite ovens and infinite workers, we can make as many pizzas as we want in just 3 time-steps!

Making MapReduce Algorithms

The MapReduce approach is simple enough that we can discuss how to build algorithms that actually use it.

A MapReduce algorithm is composed of three parts.

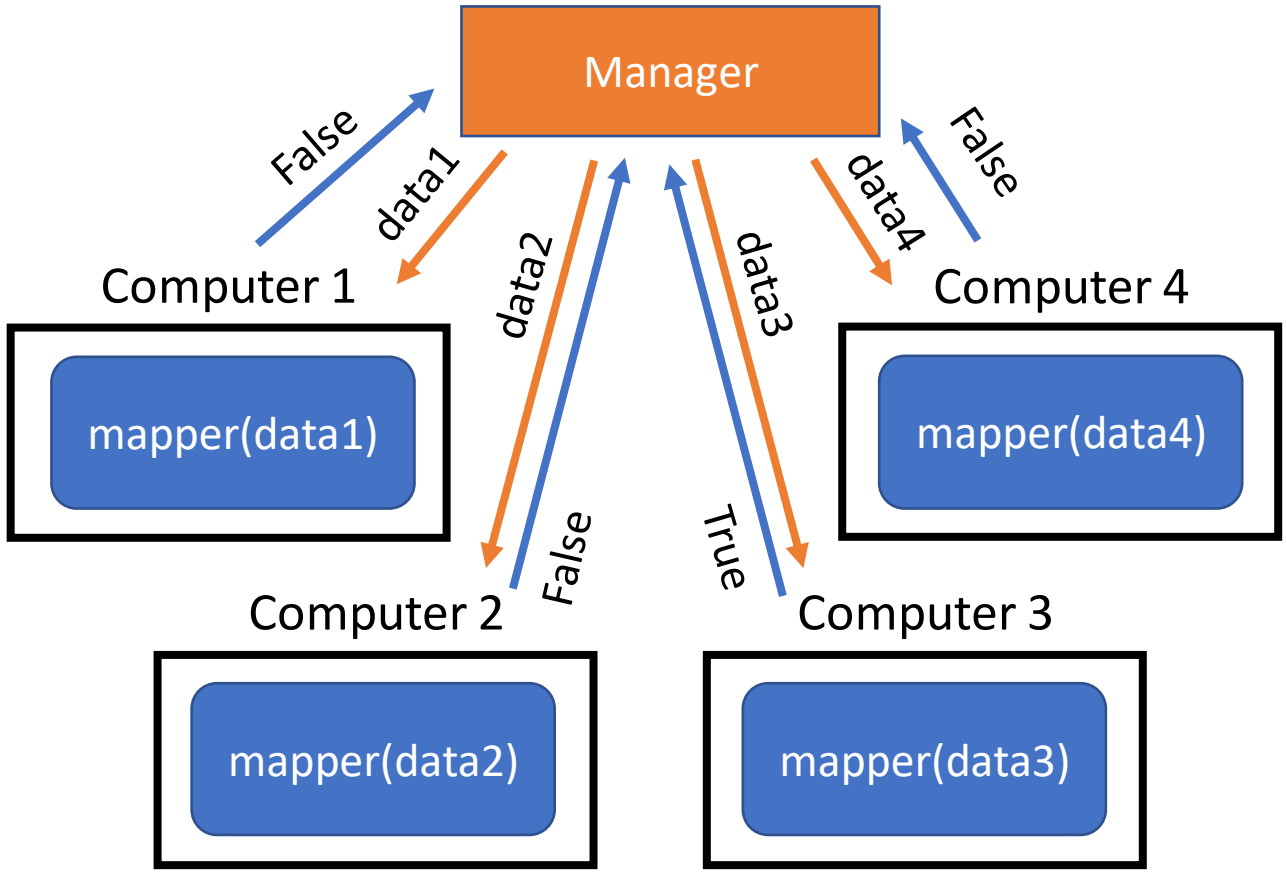
- The **mapper** takes a piece of data, processes it, and finds a partial result
- The **reducer** takes a set of results and combines them together
- The **manager** moves data through the process and outputs the final result
  - Splits up data, sends to mappers, get results back
  - Combines results together, sends to the reducer
  - Gets the final result, outputs it

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, the **manager** divides the book into many small parts- maybe one page per part. It sends 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 **manager**.

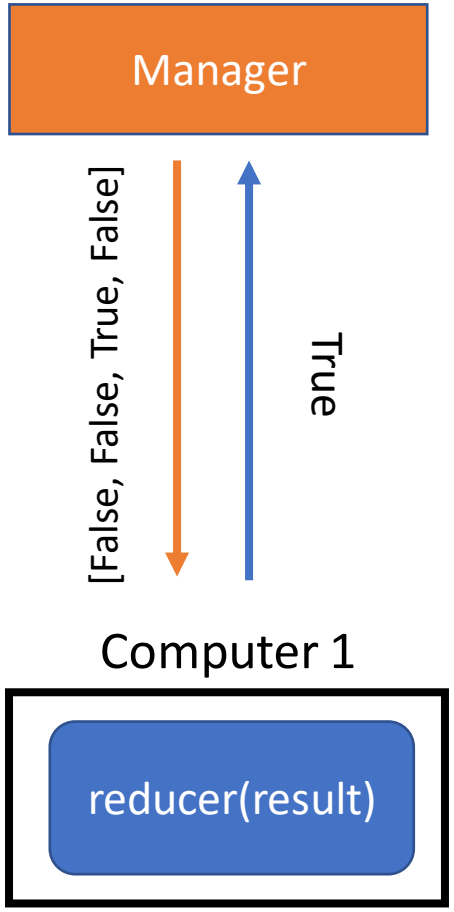


MapReduce Examples: Search – Reducer

Once all the mappers have returned their results the **manager** 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 send `True` back to the **manager** if any of them are `True`.



Coding MapReduce

We've provided a version of the MapReduce manager 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.

It's hard to tell that the system uses multiprocessing, but we can print out partial work to show what's happening. You need to end the process (by clicking the 'Terminate and restart the interpreter' button) to see what was printed in the individual calls.

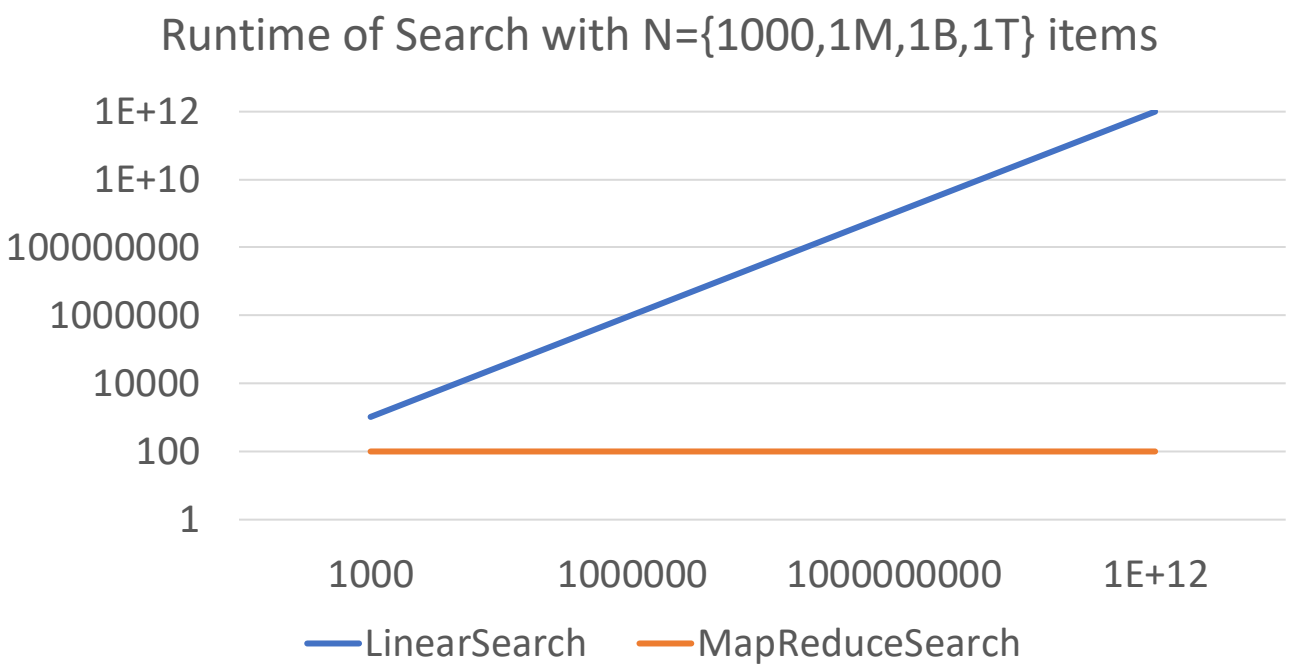
```
# Assume the page is in a file
def mapper(filename, target):
  # don't worry about reading/cleaning files
  # yet - we'll get there soon!
  text = cleanFile(readFile(filename))
  words = text.split(" ")
  for i in range(len(words)):
    word = words[i]
    if word == target:
      print("file", f, "found on word #", i)
      return True
  print("file", f, didn't find it")
  return False

# If the word is on any page, return True
def reducer(lst):
    print("reducer is checking", 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 1 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, the **manager** breaks 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 **manager** takes all those counts and puts them in a list.

The **reducer** takes the list of numbers and returns their sum.

Activity: MapReduce the Class (if time)

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

The instructor (the **manager**) will break the room into groups of 10-20 people each. Designate one person in each group as the notetaker (the **mapper**). That person must tally how many people in the group are in each of the 7 CMU schools (CIT, CFA, Dietrich, Heinz, MCS, SCS, and Tepper).

When the mappers are done, the notetakers will pass their papers to the instructor (the **manager**), who will pass them off to someone else (the **reducer**) to combine into one final tally.

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 **parallelized algorithms** for distributed computing

Feedback: https://bit.ly/110-f21-feedback