Unit 3 Review

15-110 – Monday 11/09

Schedule

- Unit Recap
- MapReduce
- Pipelining
- RSA
- Encryption Efficiency

Unit Recap

Unit 3 Goals

In Unit 3, we discussed two primary ideas:

How to **scale up** computing that involves enormous data or inefficient algorithms.

How to support **computing across multiple devices**.

Scaling Up Computing

First, we discussed how to use concepts of **concurrency** and **parallelization** to run programs more efficiently.

We discussed different **levels of concurrency** that support parallelized work on a single computer.

We also talked about how it's **difficult to design** parallelized algorithms, but that it grows easier if you use **algorithmic patterns** to split up either the data or the algorithm itself.

Computing Across Multiple Devices

We also discussed a level of concurrency and bridges multiple computers - **distributed computing**. This allows certain algorithms (like **MapReduce**) to efficiently compute enormous amounts of data.

We learned the core elements of the largest distributed computing network, **the internet**, and discussed how data can be quickly and safely sent across this network.

We also discussed how to make distributed computing **private and secure** through **authentication** and **encryption**.

Next Unit

Our next unit will focus on something different: how to use **computer science as a tool** in other disciplines.

We'll focus specifically on three useful methods that intersect with the field of computer science: **data analysis**, **simulation**, and **machine learning**.

Our final unit will then briefly touch on how computer science interacts with the world at large.

MapReduce

Big Idea

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

For example: it might take a long time to process 10 million data points. But if you break them up into 1,000 datasets (each with 10,000 data points), and send each dataset to its own computer, you can get the results 1,000 times faster!

As long as you have many many computers to distribute work across, you can reduce 'n' down to constant size.

MapReduce Pattern

The purpose of the MapReduce pattern is to make it easier to split up the data and combine the results back together.

The **mapper** takes a small dataset and returns an answer. We run many, many mappers across many computers.

The **reducer** takes a bunch of the mapper results and combines them into one result. We can have one or multiple reducers.

The **collector** does all the management work- it splits up the data, sends it to computers, collects the results, puts them together, and sends them to the reducer. The collector is the most complex part, but it is **standardized** across different problems - you don't need to re-implement it every time!

MapReduce Example - Searching the Internet

How would MapReduce be applied to something like a search request? **Break up the data**.

1. **Collector** divides all the indexed websites on the internet into many many small groups, sends those groups to many many computers

2. Each computer runs the **mapper** on its set of websites, returns a dictionary mapping possible results to confidence ratings

3. **Collector** merges all the result dictionaries into a mega-dictionary, which it sends to a computer

4. The computer runs the **reducer** to find the most-confident results, and returns them in a sorted list

Pipelining

Big Idea

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

For example: to run an algorithm that takes ten steps on 10 million data points, break up the ten steps across ten different processes. Each processor gets a data point from the previous step, runs its step, and passes the data point on to the next step.

This makes the entire algorithm **ten times faster,** assuming each step takes the same amount of time.

Pipelining and Shared Resources

Pipelining only gives a constant-time improvement, whereas MapReduce can pseudo-improve the function family. Why bother with pipelining?

This process is incredibly useful when dealing with **shared resources**. If an algorithmic step requires the use of a shared resource, and that resource can't be parallelized, MapReduce won't work - it can't duplicate the resource.

Pipelining can delegate work on that shared resource to a single process. This ensures that there is no wasted time where the resource isn't being used.

Pipelining Example

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

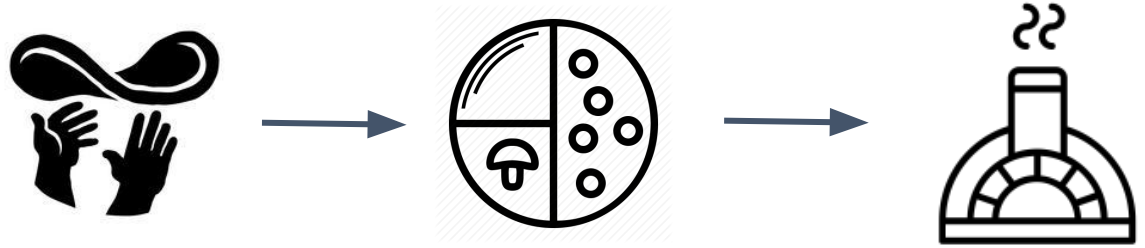
1. Flatten the dough
2. Apply the toppings
3. Bake in the oven

If we have infinite ovens, we can parallelize with MapReduce (have many cooks each make a few pizzas independently) for optimal parallelization.

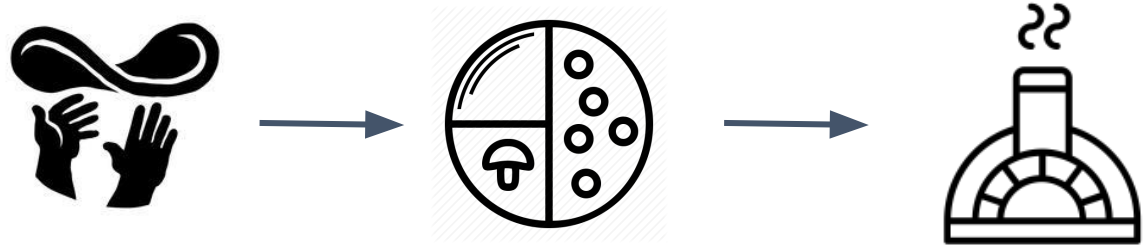
But if the number of ovens is **limited**, we may want to use pipelining instead (have some cooks focus on dough, others on toppings, and others on the limited number of ovens). This also helps for tasks that require prep time to set up - maybe for flattening, the cook only has to clean the counter and flour it once.

Perfect MapReduce - 4 workers, 3 time-steps

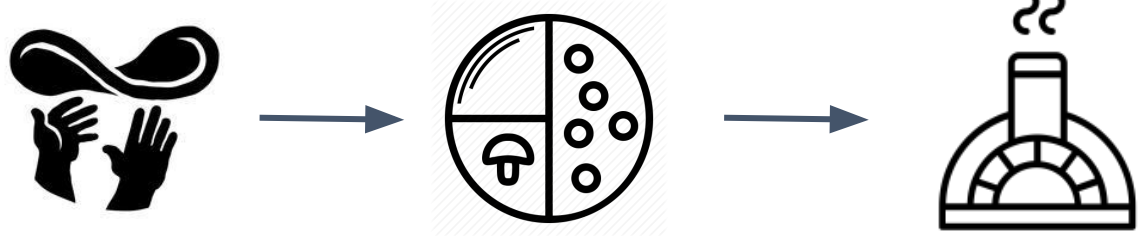
Worker 1:



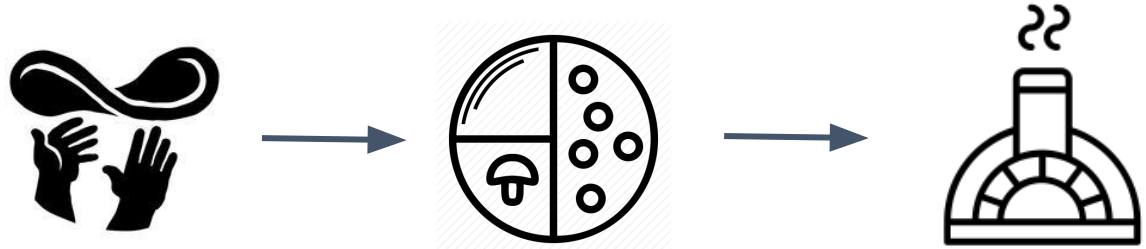
Worker 2:



Worker 3:

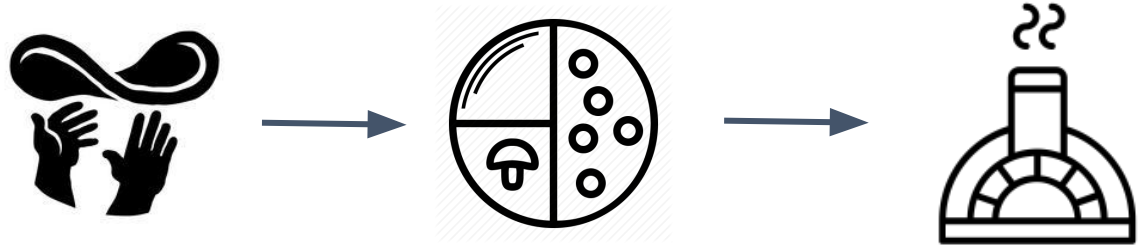


Worker 4:

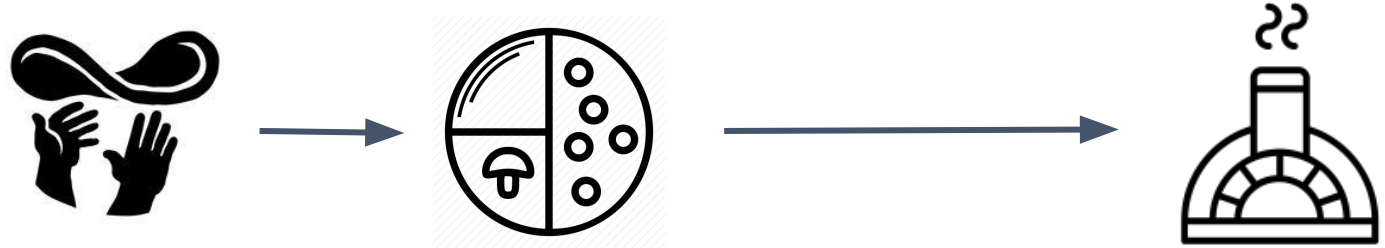


Limited MapReduce - 4 workers, 6 time-steps

Worker 1:

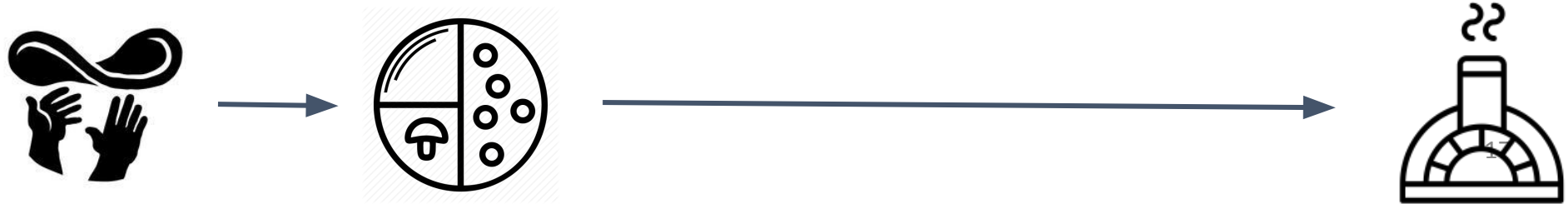


Worker 2:

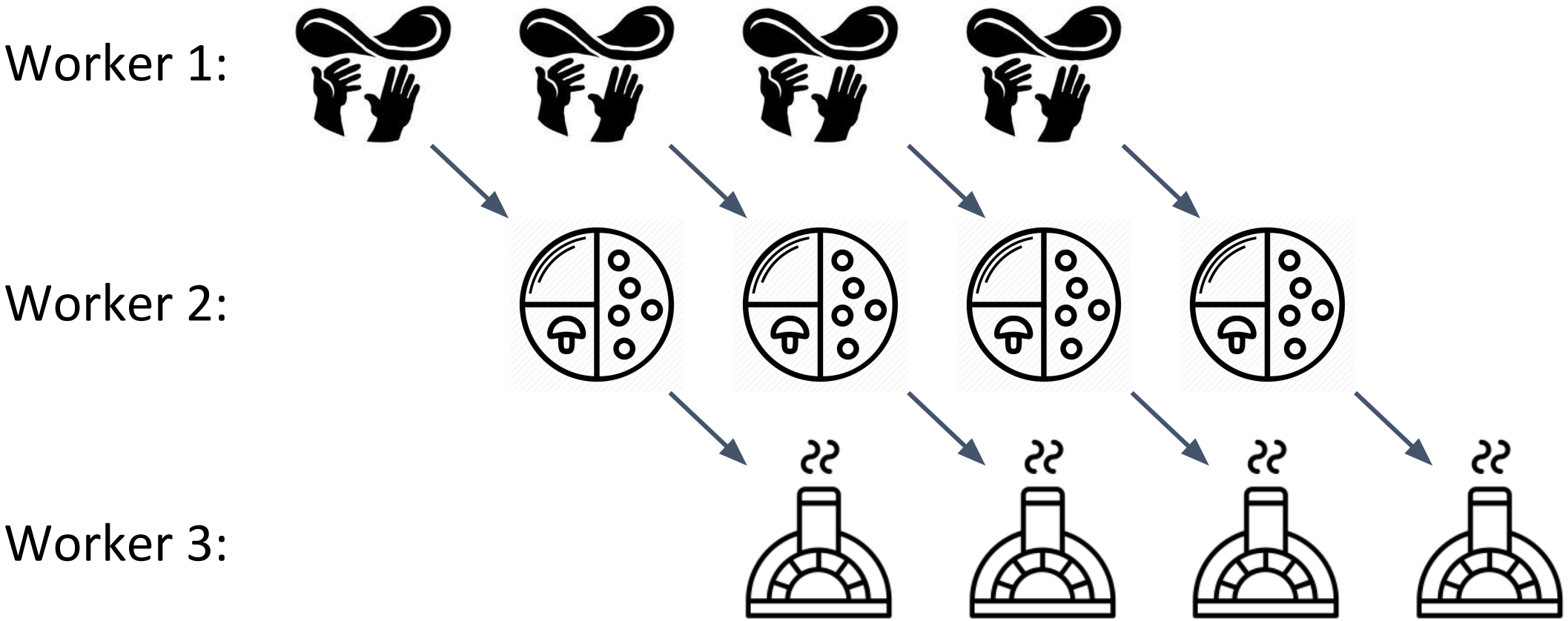


Worker 3:

Worker 4:



Pizza Pipelining - 3 workers, 6 time-steps



RSA Encryption

Overview of RSA Encryption

- Named after its inventors: Rivest, Shamir, and Adleman
- Used in lots of Internet infrastructure, such as https: protocol.
- RSA is an example of an asymmetric encryption scheme:
  - a public key is used to encrypt a message
  - a private key must be used to decrypt the message

Communication With Asymmetric Encryption

Alice wants to send a message to Bob. How can she do it securely?

1. Alice looks up Bob's public key.
2. Alice encrypts the message using Bob's public key.
3. Alice transmits the message to Bob.
    - Carol is eavesdropping on the communication, but only sees ciphertext.
4. Bob receives the encrypted message.
5. Bob uses his private key to decrypt the ciphertext, recovering the plaintext.
6. Bob reads the plaintext.

How to Generate Public and Private Keys

Pick two really large prime numbers p and q.

Calculate $n = p \times q$

Calculate d and e from n, p, and q using a hairy formula (Wikipedia).

The tuple (d,n) is your private key. Keep it secret!

The tuple (e,n) is your public key. Tell the world.

Encrypting a Message

Let x denote the bytes of your message treated as a really large integer.

Get your intended recipient's public key (e,n).

Calculate $c = x^e \bmod n$. That is the encrypted message.

Decrypting a Message

Start with the ciphertext bytes c, treated as a really large integer.

Recall your private key (d,n) that is paired with the public key the sender used to encrypt the message.

Calculate a $x = c^d \bmod n$. That is the plaintext message.

Why Does This Work?

The method for calculating d and e from n, p, and q ensures that:

$$(x^e)^d \bmod n = x$$

For encryption and decryption we rely on the fact that:

$$(x^e \bmod n)^d \bmod n = (x^e)^d \bmod n = x$$

Encryption Efficiency

Efficiency of Encryption/Decryption/Cracking

A good encryption scheme should provide efficient algorithms for encryption and decryption, but have no tractable algorithm for cracking the code.

Consider Caesar's cipher, which moves each letter by a constant number of positions c. (Caesar himself used c=3).

- Encryption is O(n) where n is the length of the message. Great!
- The keyspace is of size 26, which is constant. So we can crack the code by trying every possible key, which takes O(1) time. Disaster!

Efficiency of RSA

The RSA algorithm uses integer multiplication and modulus operations, which are at worst $O(n^2)$. Faster algorithms exist.

RSA allows for keys to be as long as we like. An k-bit key gives a keyspace of size 2^k, so trying every key is $O(2^k)$ which is intractable.

Given the public key (e,n), if we could factor n and recover p and q, then we could calculate the private key (d,n) and decrypt the message.

But factoring numbers is an NP-hard problem: no polynomial time algorithm for prime factorization exists (as far as we know).

Is RSA Really Secure?

If we discover that P = NP, we'll be able to do prime factorization efficiently, and RSA will no longer be secure.

Quantum computers can factor numbers efficiently using Shor's algorithm. But no one has built a quantum computer with enough bits to break modern RSA encryption, which uses 1024 or 2048 bit keys.

Lots of people are working on quantum computing. Stay tuned...