Lesson 6: Parallel Architectures

Phillip J. Windley, Ph.D.
CS462 – Large-Scale Distributed Systems

Lesson 06: Parallel Architectures

Contents

Introduction

Parallel architectures are a sub-class of distributed computing where the processes are all working to solve the same problem.

There are different kinds of parallelism at various levels of computing. For example, even though you might write a program as a sequence of instructions, the compiler or CPU may make changes to it at compile time or run time so that some operations happen in parallel or in a different order. This is called *implicit parallelism*.

Explicit parallelism, the kind we care about in this lesson, occurs when the programmer is aware of the parallelism and designs the processes to operate in parallel.

This lesson will explore some of the basic concepts in parallel computing, look at some of the theoretical limitations, and focus on a specific kind of parallel programming called MapReduce.

Lesson Objectives

After completing this lesson, you should be able to:

1. Understand the different taxonomies for classifying parallel computing
2. Explain Amdahl's Law in simple terms
3. Explain the reasons why communication and synchronization limit good parallel performance
4. Describe the advantages of hierarchical architectures
5. Explain how MapReduce works.
6. Use MapReduce to solve a data processing problem.

Reading

Read the following:

- Parallel computing from Wikipedia (https://en.wikipedia.org/wiki/Parallel_computing)
- Introduction to Parallel Architectures from MIT Open CourseWare starting at minute 20:52 (http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-189-multicore-programming-primer-january-iap-2007/lecture-notes-and-video/l3-introduction-to-parallel-architectures/)
- MapReduce: Simplified Data Processing on Large Clusters (PDF) (http://research.google.com/archive/mapreduce-osdi04.pdf) by Jeffrey Dean and Sanjay Ghemawat

Classifying Parallel Architectures

Parallel architectures are a specific kind of distributed system.

Concurrency and Parallelism

Concurrency and parallelism are related concepts, but they are distinct.

Concurrent programming happens when several computations are happening in over-lapping time periods. Your laptop, for example, seems like it doing a lot of things at the same time—just look at the process list—even though there are only 1, 2, or 4 cores. So, we have concurrency without parallelism.

At the other end of the spectrum, the CPU in your laptop is carrying out pieces of the same computation in parallel to speed up the execution of the instruction stream.

Parallel Computation's Space

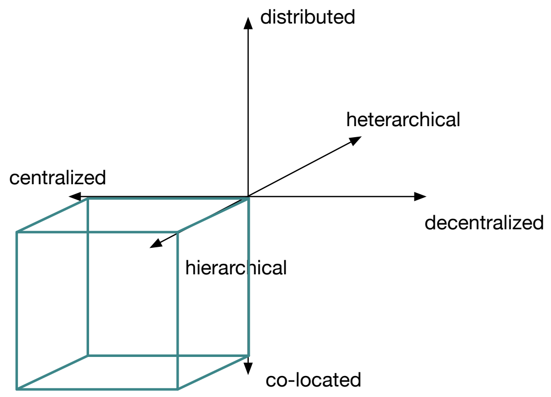
Parallel computing occupies a unique spot in the universe of distributed systems.

Parallel computing is *centralized*—all of the processes are typically under the control of a single entity.

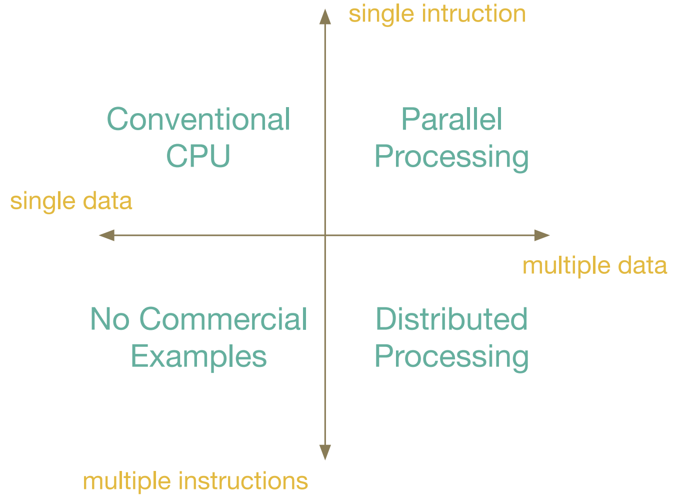
Parallel computing is usually *hierarchical*—parallel architectures are frequently described as grids, trees, or pipelines.

Parallel computing is *co-located*—for efficiency, parallel processes are typically located very close to each other, often in the same chassis or at least the same data center.

These choices are driven by the problem space and the need for high performance.



Flynn’s Taxonomy



Michael Flynn classified processing on the basis of instructions and data.

A conventional CPU has a single stream of instructions acting on a single stream of data.

Distributed processing of the type we’ll discuss for the rest of the course has multiple streams of instructions acting on multiple streams of data.

Parallel processing has single instruction stream acting on multiple streams of data, sometimes referred to as SIMD.

Amdahl's Law

Amdahl's Law describes the potential speed-up from parallel processing.

Performance Limitations

Adding additional resources doesn't necessarily speed up a computation. There's a limit defined by Amdahl's Law.

The basic idea of Amdahl's law is that a parallel computation's maximum performance gain is limited by the portion of the computation that has to happen serially which creates a bottleneck.

And there's almost always a serial portion: scheduling, resource allocation, communication, and synchronizing to name a few.

For example, if a computation that takes 20 hours on a single CPU has a serial portion that takes 1 hour (5%), then Amdahl's law shows that no matter how many processors you put on the task, the maximum speed up is 20x.

Consequently, after a point, putting additional processors on the job is just wasted resource.

George Amdahl

Calculating the Limits

The performance limits of an algorithm can be calculated using a simple formula.

If *P* is the proportion of the program that can be parallelized, then *(1 - P)* is the serial portion. The speed up that can be achieved with *N* processors is given by the following formula:

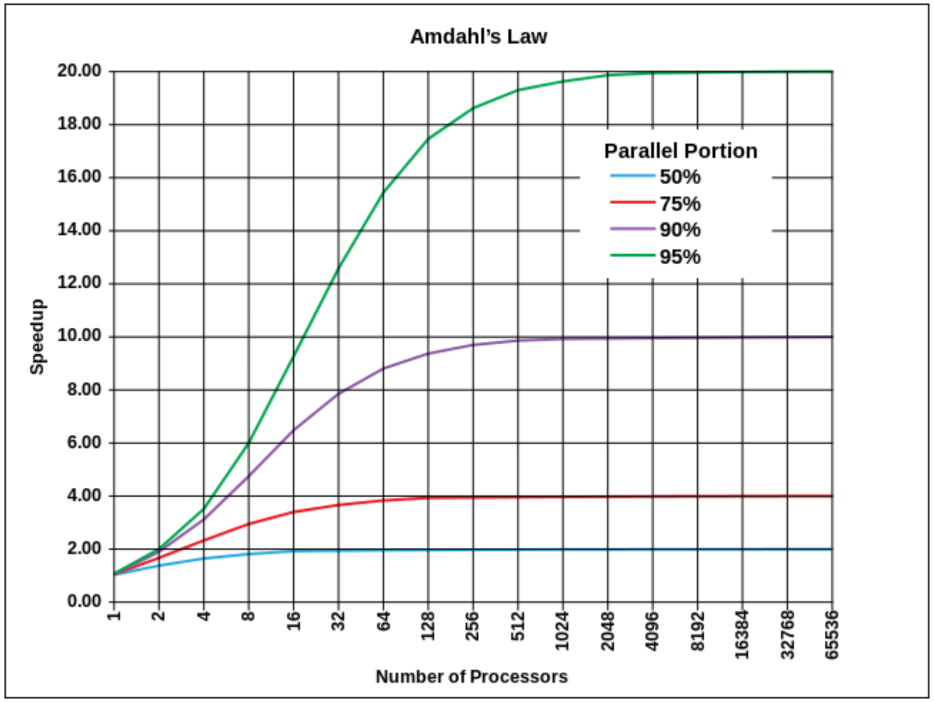
$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

As *N* goes to infinity, this formula becomes *1/(1 – P)* giving us the maximum possible speed up. For example if *P* is *.9* then *S(N) = 10* showing the maximum speed up is 10x.

This performance limit applies to hiring short order cooks as much as it does to computing. Adding more resources doesn't always improve performance. The serial portion of a process—whether a computation or a restaurant– is the bottleneck.

Consequently, much of parallel programming consists of finding algorithms with the largest *P* possible.

Picturing the Limits



As you can see from the graph, the limits are asymptotes that quickly reach their maximum.

Achieving 95% parallelism is hard and many algorithms fall short of that, so most parallel computations don't make use of as many threads as you might expect.

Problems with very high values of *P* are known as *embarrassingly parallel* problems.

Performance Factors

Communication and synchronization are the roadblocks to good parallel performance.

Process Interaction

Except for embarrassingly parallel algorithms, the threads in a parallel computation need to communicate with each other. There are two ways they can do this.

Shared memory – the processes can share a storage location that they use for communicating. Shared memory can also be used to synchronize threads by using the shared memory as a semaphore.

Messaging – the processes communicate via messages. This could be over a network or special-purposes bus. Networks for this use are typically hyper-local and designed for this purpose.

Consistent Execution

The threads of execution for most parallel algorithms must be coupled to achieve *consistent execution*.

Obviously, parallel threads of execution communicate to transfer values between processes. Parallel algorithms communicate not only to calculate the result, but to achieve *deterministic* execution.

For any given set of inputs, the parallel version of an algorithm should return the same answer each time it is run as well as returning the same answer a sequential version of the algorithm would return.

Parallel algorithms achieve this by locking memory or otherwise sequencing operations between threads. This communication overhead, as well as the waiting required for sequencing impose a performance overhead.

As we saw in our discussion of Amdahl's Law, these sequential portions of a parallel algorithm are the limiting factor in speeding up execution.

Hierarchical Architectures and MapReduce

MapReduce is a popular parallel architecture for data processing

Pipelines

You're probably familiar with Unix *pipes* and *filters*. Unix commands that read from the standard input and write to the standard output are called filters. You can use the pipe symbol (|) to string filters together to accomplish a specific task.

For example, `cat` prints a file to standard out, `tr` transliterates characters, `sort` sorts lines of text, and `uniq` removes duplicate lines.

You can use them together to print all the unique words in a file of text called chap1.txt as follows:

```
cat chap1.txt | tr ' ' '\n' | sort | uniq
```

Unix doesn't necessarily run these filters in parallel, but many pipelines could be run concurrently. (This one has a significant limitation on parallel execution, can you spot it?)

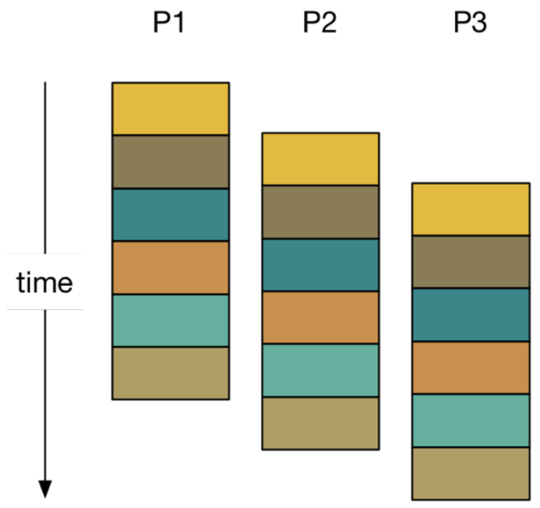
The linear structure of the pipeline defines the coupling between processes.

Parallel Pipelines

When pipelined processes are executed in parallel, we get a picture that looks like the diagram to the right.

With 3 processes, we spend the first 2 steps filling the pipeline and the last 2 steps emptying it. The longer the stream of uninterrupted steps for each processes, the more efficient the pipeline becomes.

Looping that can't be unwound and put in the pipeline as well as branching interrupt the smooth flow of operations that the pipeline needs to process, making it less efficient.



Functional Models

Functional programming languages have long touted reduced coupling and the resultant ease in mapping to parallel algorithms.

Functional languages have higher-order functions that can control the application of other functions to data structures. Two of the most frequently used are

- `map(f, l)` – a function that takes another function, `f`, and a list, `l`, and returns a new list where each member is the result of applying `f` to the corresponding member of `l`. It's easy to see how map() can be parallelized.
- `reduce(f, l)` – a function that applies a function `f` to combine members of a list `l` resulting in a single value.

For example, the following increments the numbers in a list and then returns the sum:

```
reduce(+, map(inc, [21,56,34,75]))
```

Of course, `map()` and `reduce()` don't have to operate on just numbers.

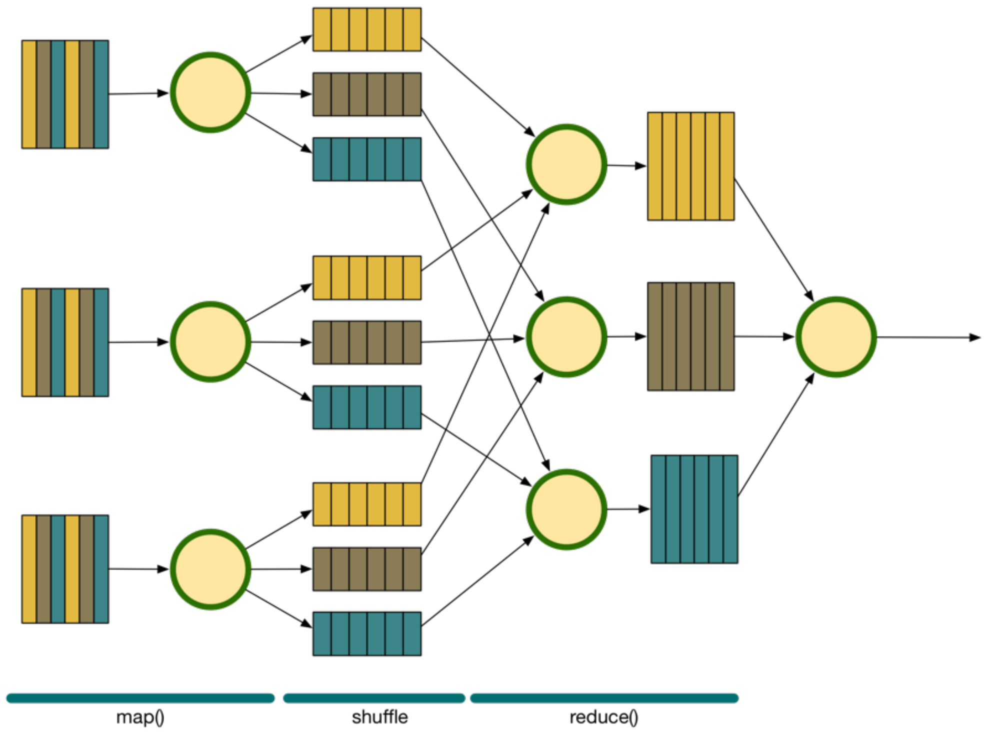
MapReduce

By structuring the interactions to use `map()` and `reduce()` operators, a MapReduce processing system provides an easy way to create efficient parallel algorithms for a variety of operations.

While not every problem can be written in terms of `map()` and `reduce()` operators, many so-called *big data* problems can be. Consequently MapReduce has become very popular in recent years as programmers look for reliable ways to quickly process large data sets using commodity processors and networking hardware.

MapReduce is a parallel processing model inspired by the `map()` and `reduce()` operators of functional programming.

MapReduce: Shuffling and Grouping



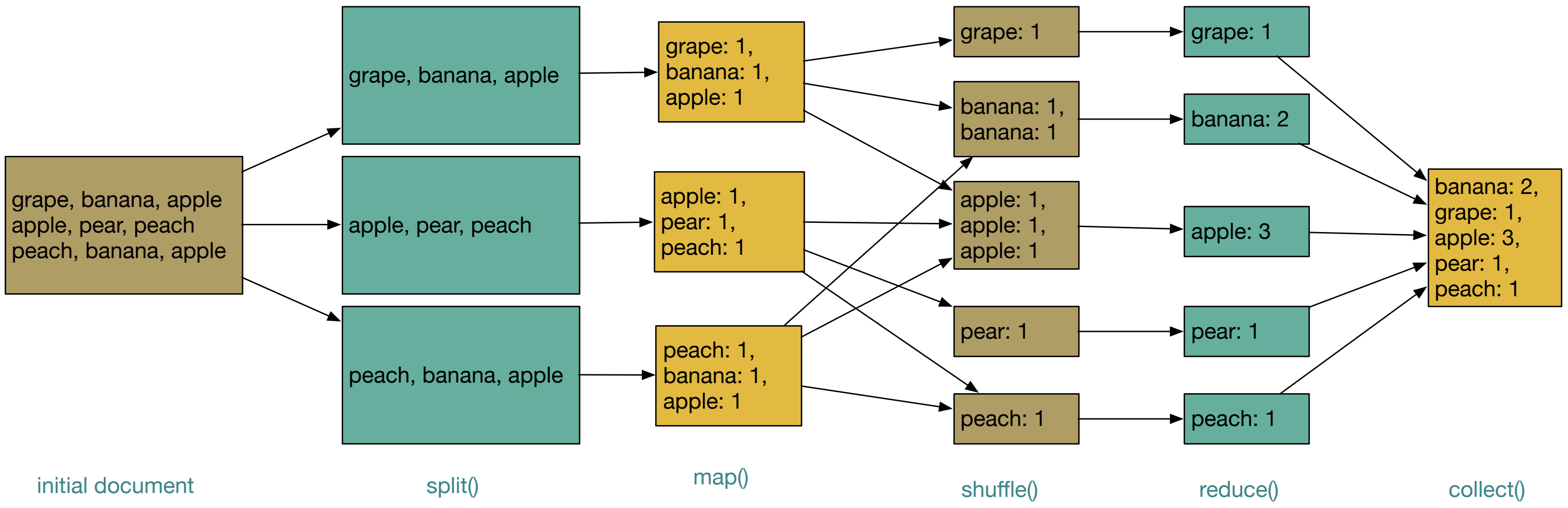
MapReduce doesn’t just operate on plain lists, but on lists of key-value pairs.

MapReduce uses a shuffle or grouping step to collect all the values with the same key. Shuffling is very fast because of the use of the key to group like items.

Programmers supply the code for the `map()` and `reduce()` operators, but the shuffle and master control processes are part of the MapReduce system.

MapReduce Example: Counting Word Occurances

The following diagram shows how MapReduce can be used to count the occurances of words in a document. `Split()`, `shuffle()`, and `collect()` are all very low cost , *O(n)*, operations that are provided by the MapReduce System. The programmer supplies the `map()` and `reduce()` functions.



MapReduce Example: Count URL Access Frequency

Suppose we have a big, unordered list of URLs that have been accessed from our system over some period of time.

We can use MapReduce to count how many times each has been accessed by first using `map()` to create a key-value pair of each URL and the number 1 and `reduce()` to sum up all the 1's for each URL.

The shuffle step makes grouping all of the tuples with the same key (URL) very fast. The programmer merely supplies a simple program to create a key-value pair for the mapping stage and a counting routine for the reduce stage.

```
[<url0>, <url1>,…,<urln>]

Map()
[{<url0>, 1},
 {<url1>, 1},
 …,
 {<urln>, 1}]

Reduce()
[{<url>, <count>},
 {<url>, <count>},
 …]
```

MapReduce Example: Inverted Index

Suppose we have a bunch of documents and we want an index of which words appear in each document.

We can create such an index with MapReduce by serializing the words in each document (possible with another MapReduce process) and then creating a key-value pair with each word and the document identifier.

The reduce stage then accumulates a list of all the document identifiers for each word.

```
doc₀:[<word₀>, <word₁>,…,<wordₙ>]

Map()
[{<word₀>, docₓ},
 {<word₁>, doc_y},
 …,
 {<wordₙ>, doc_z}]

Reduce()
[{<word₀>, [<docₓ>, doc_z]},
 {<word₁>, [<doc_a>, doc_b]},
 …]
```

Conclusion

Summary & Review

Credits

Summary and Review

Parallel processing is a large area of study. You could devote your entire career to understanding and using parallel processing to solve problems. This lesson is merely an introduction to the field.

We began by classifying parallel processing, both in our own distributed systems space as well as understanding it in terms of Flynn's taxonomy.

Amdahl's Law presents a fundamental limitation to the performance speed up we can get with a parallel algorithm. This law has wide applicability in and out of parallel processing.

Communication and synchronization pose fundamental problems for parallel processing that we cannot get around. Every parallel algorithm must meet the consistent sequencing requirement. The coupling that results slows down parallel computation.

MapReduce is way of structuring parallel computations that eases the burden on the programmer for setting up the parallel computation and standardizes the interactions between processes. MapReduce has become very popular for processing large data sets on commodity hardware and networks.

Credits

Photos and Diagrams:

- Photo of Gene Amdahl from Wikipedia (https://commons.wikimedia.org/wiki/File:Amdahl_march_13_2008.jpg), CC BY-SA 3.0
- Amdahl's Law graph (https://commons.wikimedia.org/wiki/File:AmdahlsLaw.svg), CC BY-SA 3.0

All other photos and diagrams are either commonly available logos or property of the author.