CS 677: Big Data

Distributed Computation

Lecture 9

Today's Schedule

- Distributed computation overview
- Hardware platforms
  - Supercomputers
  - Beowulf clusters
  - Grid computing, cloud computing
- Computation Engines
  - MPI
  - Bulk Synchronous Parallel
  - MapReduce

Today's Schedule

- **Distributed computation overview**
- Hardware platforms
  - Supercomputers
  - Beowulf clusters
  - Grid computing, cloud computing
- Computation Engines
  - MPI
  - Bulk Synchronous Parallel
  - MapReduce

Distributed Computation

- In our storage systems discussions, we've focused on spreading data out over cluster(s)
- As you can imagine, we also need to take advantage of **distributed computations** to ensure we can process all that data efficiently
- The basic idea behind distributed computations is to **divide and conquer** our problems
  - …but this is a bit different than usual multi-process / multi-core / parallel computing / etc.

Distributed Task Elements

These tasks are concerned with the following:

1. Communication
   - Getting/sharing data, coordinating with other tasks
2. Computation
   - Applying transformations to the data, moving it from *State A* to *State B*
3. Storage
   - Retrieving the initial data state, storing modified state

Principles of Distributed Computation [1/2]

- We need to tailor our tasks for this kind of environment
- Principle 1: **Reduce communication**
  - Communication in a distributed system is latency-prone, even with a fast interconnect
    - **Much** slower than in-memory access
- Principle 2: Break computations into pieces that can **run independently**
  - Try to avoid data/computation dependencies, e.g., Task 3 requires Task 2 to complete, Task 2 requires Task 1…

Principles of Distributed Computation [2/2]

- Principle 3: **Avoid** excessive round trips to **storage**
  - You should never be accessing data more than once, if at all!
    - Indexes can help avoid data that you don't want
  - Writes are even slower than reads
    - Don't write too much intermediate state data to the disk if simply recomputing it will be cheaper

Building It

- What are the features we need for a minimum viable computation engine?
  - Discuss with your neighbor
- …
- Very basic: create a shell script that will ssh to your cluster machines, launch the computation, and then collect the results from stdout
  - This has a long list of shortcomings…

Today's Schedule

- Distributed computation overview
- **Hardware platforms**
  - Supercomputers
  - Beowulf clusters
  - Grid computing, cloud computing
- Computation Engines
  - MPI
  - Bulk Synchronous Parallel
  - MapReduce

CDC 6600

CDC 6600: Tech Specs

- 60-bit CPU, ten 12-bit I/O processors
- 3 megaFLOPS
- Memory: 128K 60-bit words
- Dual video display console
  - Pretty cool: vector system instead of raster
- Storage: 2 MB
  - Could add magnetic drum storage for expansion!
- Yours for ~$10m

Field Trip: IBM 1401

Supercomputing

- Pioneered by Seymour Cray, ~1960s
- Observed that simply making the CPU much faster wasn't all that beneficial
  - Still have to wait for other components to catch up
  - (Assuming the CPU drives everything)
- Instead, Cray designed a system that linked 10 simple computers
  - Each of the 10 PPUs were responsible for shuffling data in and out of memory

System Design

- Original supercomputers used custom hardware to accelerate performance and allow parallelism
- Over time, more off-the-shelf components were used instead
  - Huge leaps in performance of commodity CPUs
- There are still some advantages over a standard cluster:
  - Better interconnects (e.g., Infiniband)
  - Better integration

Top500

- A list of the top 500 supercomputers is available at: https://www.top500.org
- Current #1: *Frontier*
  - 8,730,112 Cores
  - Peak performance exceeding one ExaFlop/s
  - Oak Ridge National Laboratory
- See also: Green500 https://www.top500.org/green500/

Beowulf Clusters

- Over the years, many big computing tasks have migrated away from supercomputing platforms
  - At the same time, supercomputers look more and more like clusters
- "Beowulf" terminology coined in 1994 @ NASA
- Grab a bunch of commodity PCs, install software like OpenMPI, MPICH
  - "Supercomputer" on the cheap!
  - We'll talk about MPI in a minute…

Grid Computing

- Thus far we've focused on communications and data transfer
- **Grid** computing aims to provide processing resources at scale
- Modeled after the electric grid: use resources as you need them
  - Better utilization of hardware between organizations (for instance, universities)

Hardware

- Grids are “super virtual computers” created by combining a large amount of machines
- Connected by commodity/standard networking hardware such as Ethernet
  - May span large geographic regions
- Like a traditional cluster but span across organizations
- Loosely coupled

Making it Work

- This may sound like a recipe for disaster!
  - Extreme heterogeneity
- However, **grid middleware** helps handle the heavy lifting for us
- When launching an application on a grid, we can specify type of resources we need
  - Software libraries, architectures, particular hardware features, etc.

Cloud Computing

- Then cloud computing (***ahem*** Amazon) came along…
- Realizes many goals of the grid computing movement
  - Also makes many of the same mistakes
    - Talk to a grid computing researcher sometime
- Better: **elasticity**
  - Expand and contract your resource pool as needed

Today's Schedule

- Distributed computation overview
- Hardware platforms
  - Supercomputers
  - Beowulf clusters
  - Grid computing, cloud computing
- **Computation Engines**
  - MPI
  - Bulk Synchronous Parallel
  - MapReduce

Message Passing

- Getting back to the point: the basic idea behind parallelism is **divide and conquer**
- To do this, we need to coordinate across processing units in our cluster via **messages**
- We could use **sockets**
  - Who even does that?! 😄
  - Wrong level of abstraction for high performance computing (HPC) applications
  - Every cluster/supercomputer is different

Message Passing Interface

- Message passing is the most common paradigm for programming distributed memory systems
- Processors coordinate their activities by sending messages to each other across the network
  - Infiniband
  - Ethernet
- Message Passing Interface, or just **MPI**, gives us communication primitives to do this

MPI Standard

- There are multiple implementations of MPI that target a single standard
- This allows hardware-specific optimizations: your Cray supercomputer probably ships with its own special version of MPI
  - Knows about the structure of the communication interconnects
- This leads to better performance but also compatibility issues and the usual arguments over the spec

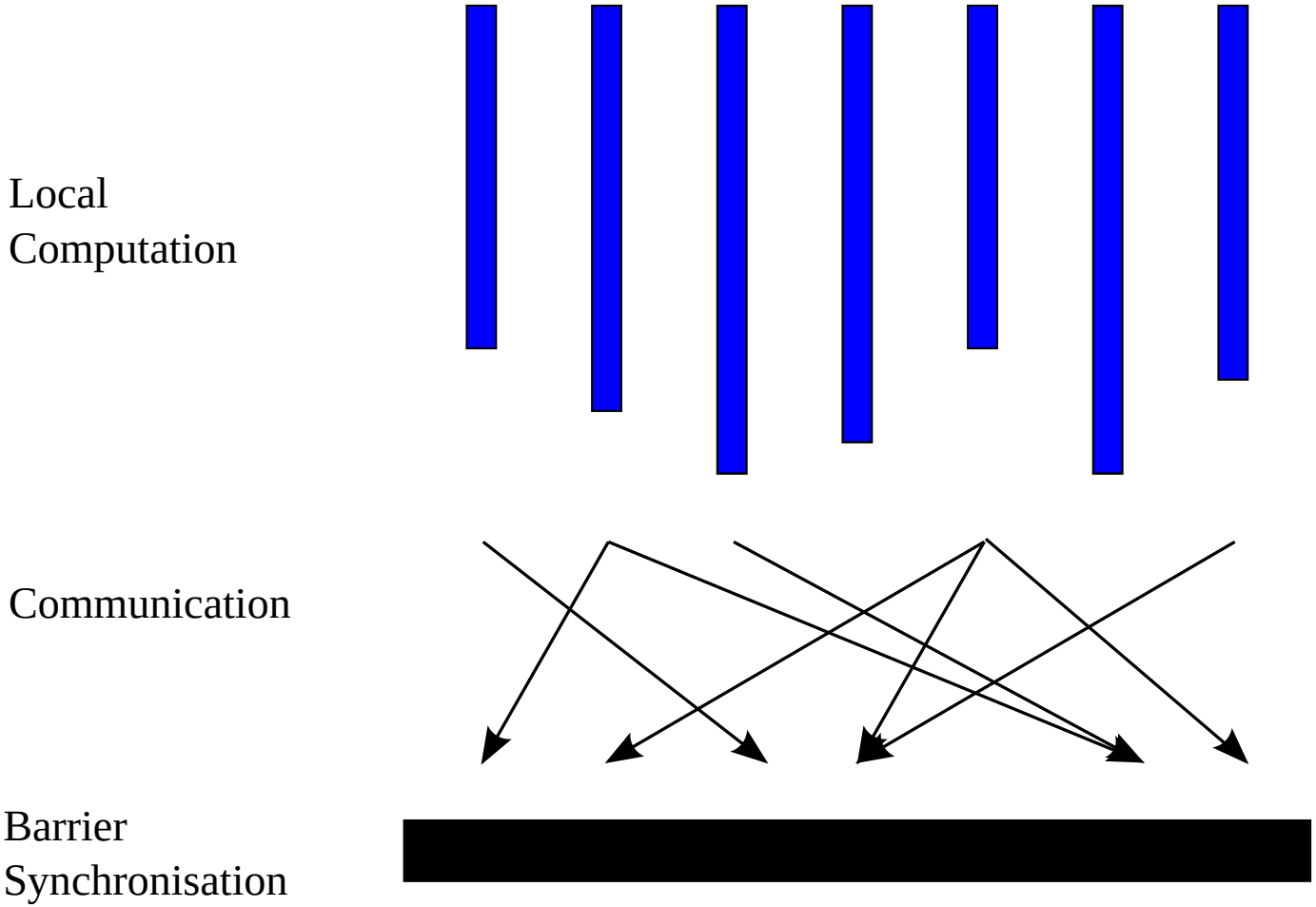
MPI Use Cases

- MPI is great for coordinating supercomputing jobs
- Used extensively for atmospheric modeling, simulations, etc.
- Servicing web requests, working with failures… not so much.
  - Could be decent for batch processing… although, what about the file system?
    - Built on a **pull** model: bring the data to the computation
  - We need other ways of dealing with these problems

Bulk Synchronous Parallel

- Computing paradigm that consists of:
  - Threads
  - Network communication
  - Synchronization
- Somewhat of a precursor to MapReduce

Bulk Synchronous Parallel



BSP: The Trade-Offs

- The good:
  - Tasks execute independently
  - Communication is structured
  - Ultimately, everything gets synched back up!
- The bad:
  - Assumes that nothing needs to happen after the sync step
  - We could potentially build a graph of these computations
  - Synchronization: you're only as fast as your slowest worker

MapReduce

- Distributed computing paradigm that pushes the BSP idea forward
- Two steps:
  - **Map**: filter, sort, produce local summaries, etc.
  - **Reduce**: combine to produce the final result(s)
- Or, *split-apply-combine*
- Based on the map() and reduce() procedures from functional computing

MapReduce Innovations

- Give the user a constrained framework and make them fit their problem to it
  - Parallelism is automatic
  - Fault tolerance can be taken care of by the framework
  - Development time is reduced
- Push computations to the data
  - (or: don't pull data to the computation)
- If you can warp your brain to work with MR, then you save yourself a lot of trouble!

<key, value> pairs

- The bread and butter of any MapReduce application is simple <key, value> pairs
  - Inputs
  - Outputs
- This is restrictive
  - Many times we have to rethink our problem in "MapReduce style"

Map

- Receives an input <k, v> pair
  - From text files: <line_num, line_text>
- Produces an intermediate <k, v> pair
- Intermediate pairs are grouped by the framework and then passed to the reducer over the network
- Usually the map phase sets things up, filters out data, performs pre-processing, etc.
  - There are some **map only** computations that can be expressed without a reduce phase

Reduce

- Receives intermediate <k, v-group> entries (transferred over the network)
- Merges the values together
- Output?
  - You guessed it! <k, v> pairs!

Other Pieces of the Puzzle

- **Shuffling**: transferring data between mappers and reducers
- **Partitioning**: determining which nodes process a given key
- **Sorting**: Keys are sorted to create groups
- **Combining**: pre-sorting groups on the mapper to reduce network communication

Data Types

- Everything is a byte array (aka String)
- Most MapReduce implementations offer some functionality to hide this fact from the developer
- Automatically serializing numbers, for instance
- In Hadoop, you can create your own data types to pass as <k, v> pairs: ***custom writables***

Example Job: Popular URIs

- Map:
  - Read log file
  - Emit <URI, 1> pairs
- Reduce:
  - Add up the counts for each URI
  - Emit <URI, total> pairs

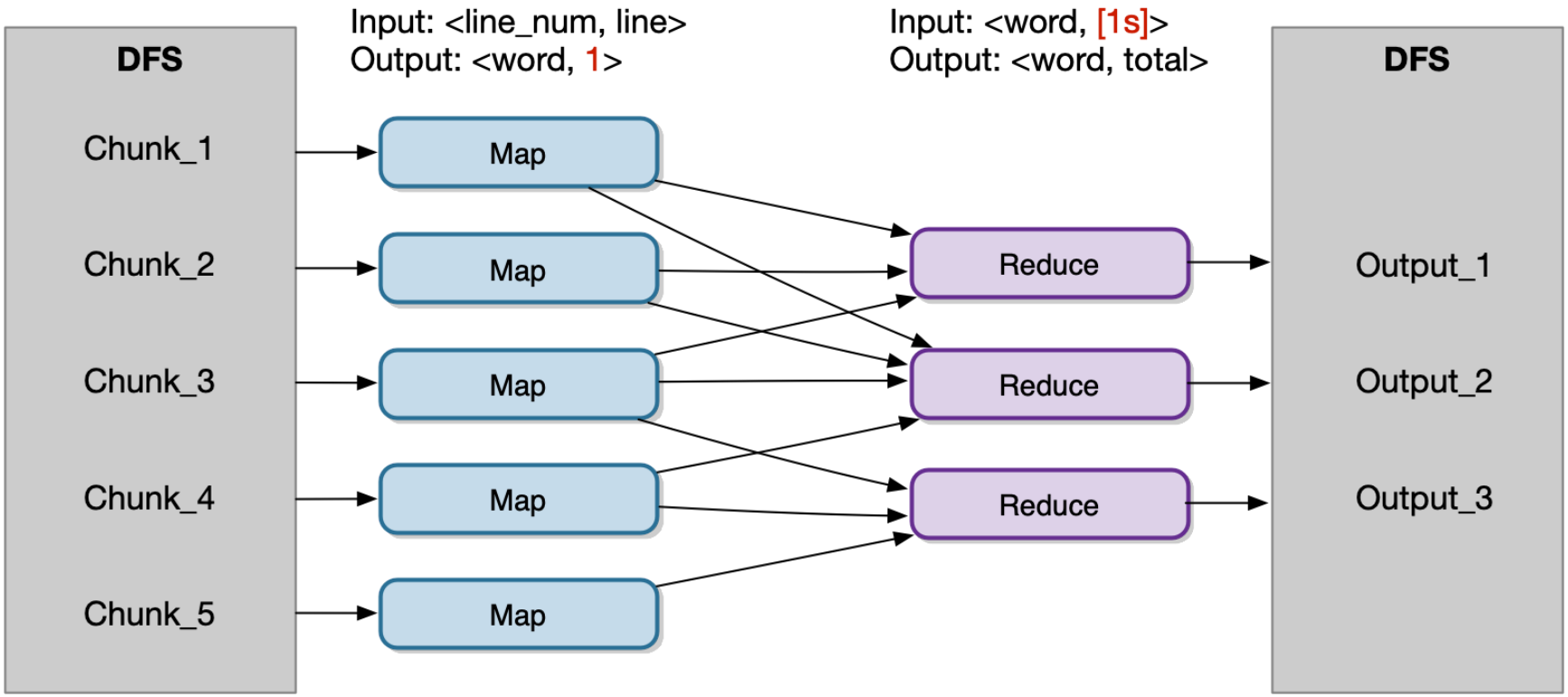
Building an Inverted Index

- Map:
  - Read web page
  - Emit <word, URI>
- Reduce:
  - Sort by URI
  - Emit <word, list(URI 1, URI 2, URI 3, … URI N)>

The "Hello World" of MapReduce

- Since printing "Hello World" on a bunch of machines isn't all that impressive, we need something else
- One of the most common examples: Word Count
- This is a pleasingly parallel job:
  - Break our input file(s) up
  - Grab each occurrence of each word
  - Emit <word, #> tuples
  - Combine the tuples (sort by the key: word)
  - Emit final set of <word, #> tuples

WordCount Data Flow



WordCount: Map

```
public void map(Object key, Text value, Context context)
throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, new IntWritable(1));
    }
}
```

WordCount: Reduce

```
public void reduce(Text key, Iterable<IntWritable> values, Context context)
throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
        sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
}
```

Word Count

- With just this information, we can analyze a lot!
- What are the most common words in our languages?
- How does the frequency of words change over time?
- We can start updating our analysis to think about sentiment, spelling changes, and more

A Few Observations

- The input datatype is assumed to be text by default
- We're emitting (outputting) <word, 1>
  - That's kind of weird, why not anything but <word, 1>?
- There are less reducers than mappers, why not have the same number?
  - ...
- Each mapper writes an output file to the DFS