CS435 Distributed systems

Continued

PARALLEL PROCESSING (B)

Dr. Basit Qureshi

PhD FHEA SMIEEE MACM

www.drbasit.org

TOPICS

- Fork Join with Divide and Conquer
- Speedup and Amdahl's Law
- MapReduce for distributed parallel processing

Most of the content in these set of slides is based on "A Sophomoric Introduction to Shared-Memory Parallelism and Concurrency", Dan Grossman, online notes, version Feb 2012

CS435 Distributed systems

FORK JOIN WITH DIVIDE AND CONQUER

Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

KEY CONCEPTS: WORK AND SPAN

Analyzing parallel algorithms requires considering the full range of processors available

- We parameterize this by letting $\mathbf{T_P}$ be the running time if **P** processors are available
- We then calculate two extremes: work and span

Work: $\mathbf{T_1}$ → How long using only 1 processor

- Just "sequentialize" the recursive forking

Span: $\mathbf{T_\infty}$ → How long using infinity processors

- The longest dependence-chain
- Example: $O(\texttt{log}\ n)$ for summing an array
  - Notice that having $> n/2$ processors is no additional help
- Also called "critical path length" or "computational depth"

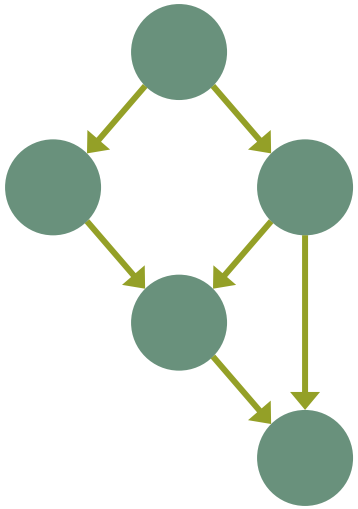
THE DAG

A program execution using **`fork`** and **`join`** can be seen as a DAG

- Nodes: Pieces of work
- Edges: Source must finish before destination starts

A fork "ends a node" and makes two outgoing edges

- New thread
- Continuation of current thread



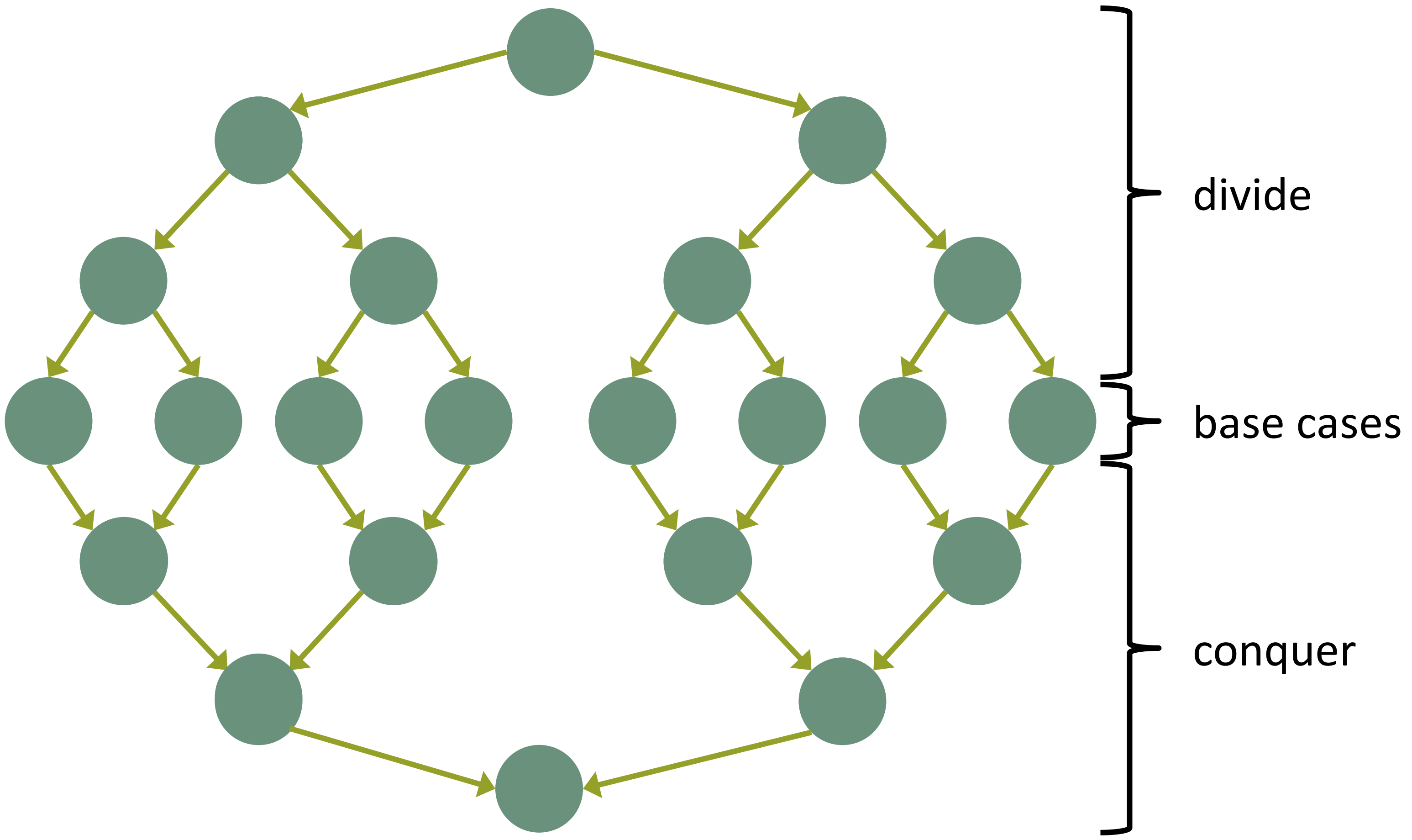
A join "ends a node" and makes a node with two incoming edges

- Node just ended
- Last node of thread joined on

OUR SIMPLE EXAMPLES

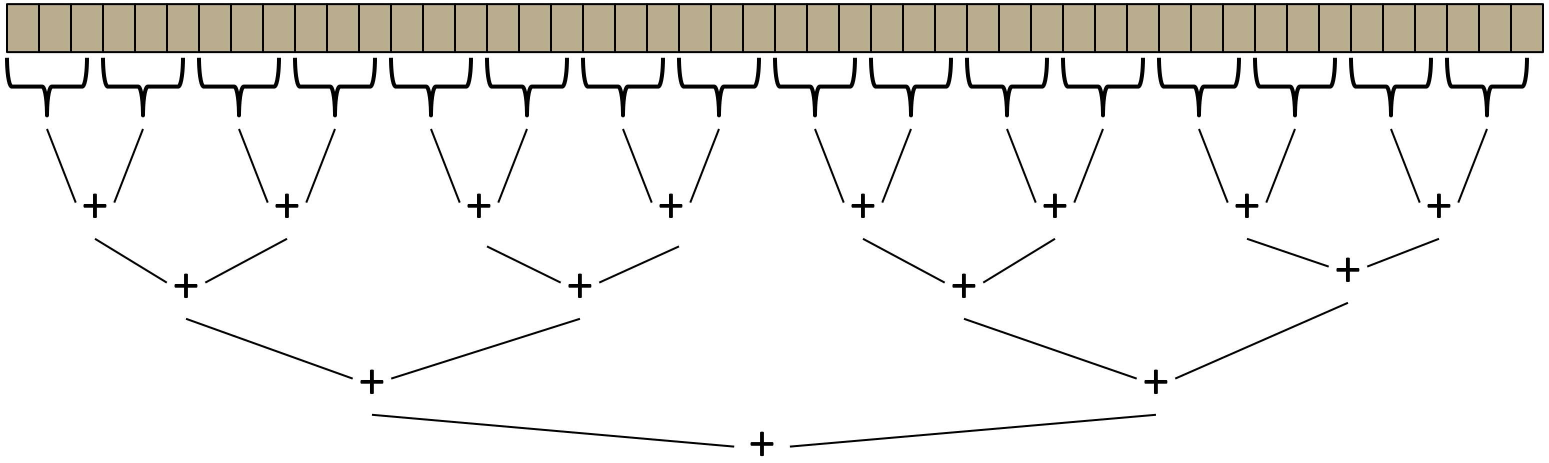
`fork` and `join` are very flexible, but divide-and-conquer use them in a very basic way:

- A tree on top of an upside-down tree



WHAT ELSE LOOKS LIKE THIS?

Summing an array went from $O(n)$ sequential to $O(\texttt{log}\ n)$ parallel (*assuming* ***a lot*** *of processors and very large n*)



Anything that can use results from two halves and merge them in $O(1)$ time has the same properties and *exponential speed-up* (in theory)

EXAMPLES

- Finding Maximum or minimum element in array with large n.
- Finding a value (e.g. 17) in a array with large n
- Counts (e.g., # of strings that start with a vowel)
  - Base case?

MORE INTERESTING DAGS?

Of course, the DAGs are not always so simple (and neither are the related parallel problems)

Example:

- Suppose combining two results might be expensive enough that we want to parallelize each one
  - Parallelize the base case itself??

REDUCTIONS

Such computations of this simple form are common enough to have a name: reductions (or reduces?)

- Produce single answer from collection via an associative operator
  - Examples: max, count, leftmost, rightmost, sum, …
- Recursive results don't have to be single numbers or strings and can be arrays or objects with fields
  - Example: Histogram of test results
- But some things are inherently sequential
  - How we process **`arr[i]`** may depend entirely on the result of processing **`arr[i-1]`**

MAPS AND DATA PARALLELISM

A map operates on each element of a collection independently to create a new collection of the same size

- No combining results
- For arrays, this is so trivial some hardware has direct support (often in graphics cards)

Canonical example: Vector addition

```
int[] vector_add(int[] arr1, int[] arr2){
  assert (arr1.length == arr2.length);
  result = new int[arr1.length];
  FORALL(i=0; i < arr1.length; i++) {
    result[i] = arr1[i] + arr2[i];
  }
  return result;
}
```

MAPS IN FORKJOIN FRAMEWORK


```
class VecAdd extends RecursiveAction {
  int lo; int hi; int[] res; int[] arr1; int[] arr2;
  VecAdd(int l,int h,int[] r,int[] a1,int[] a2){ … }
  protected void compute(){
    if(hi – lo < SEQUENTIAL_CUTOFF) {
      for(int i=lo; i < hi; i++)
        res[i] = arr1[i] + arr2[i];
    } else {
      int mid = (hi+lo)/2;
      VecAdd left = new VecAdd(lo,mid,res,arr1,arr2);
      VecAdd right= new VecAdd(mid,hi,res,arr1,arr2);
      left.fork();
      right.compute();
      left.join();
    }
  }
}
static final ForkJoinPool fjPool = new ForkJoinPool();
int[] add(int[] arr1, int[] arr2){
  assert (arr1.length == arr2.length);
  int[] ans = new int[arr1.length];
  fjPool.invoke(new VecAdd(0,arr.length,ans,arr1,arr2);
  return ans;
}
```

MAPS AND REDUCTIONS

Maps and reductions are the "workhorses" of parallel programming

We often use maps and reductions to describe parallel algorithms

- Programming them then becomes "trivial" with a little practice (like how for-loops are second-nature to you)

DIGRESSION: MAPREDUCE ON CLUSTERS

You may have heard of Google’s "map/reduce"

Perform maps/reduces on data using many machines

- The system takes care of distributing the data and managing fault tolerance
- You just write code to map one element and reduce elements to a combined result

MAPS AND REDUCTIONS ON TREES

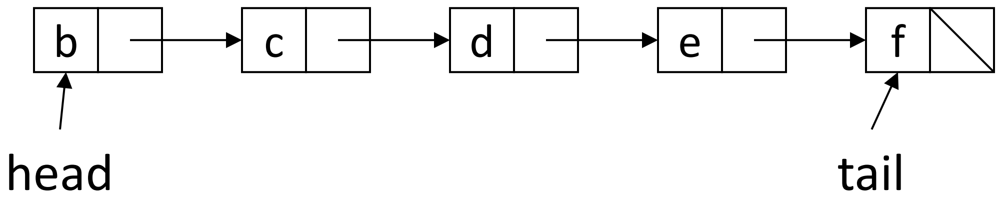
Work just fine on balanced trees

- Divide-and-conquer each child
- Example: Finding the minimum element in an unsorted but balanced binary tree takes $O(\mathtt{log}\ n)$ time given enough processors

MAPS AND REDUCTIONS ON LINKED LISTS

Can you parallelize maps or reduces over linked lists?

- Example: Increment all elements of a linked list
- Example: Sum all elements of a linked list



Once again, data structures matter!

For parallelism, balanced trees generally better than lists so that we can get to all the data exponentially faster **$O(\log n)$ vs. $O(n)$**

- Trees have the same flexibility as lists compared to arrays (i.e., no shifting for insert or remove)

ANALYZING ALGORITHMS

Like all algorithms, parallel algorithms should be:

- Correct
- Efficient

For our algorithms so far, their correctness is "obvious" so we'll focus on efficiency

- Determine asymptotic bounds
- Analyze the algorithm without regard to a specific number of processors
- The key "magic" of the ForkJoin Framework is getting expected run-time performance **asymptotically optimal** for the available number of processors

CONNECTING TO PERFORMANCE

Recall: $\mathbf{T_P}$ = run time if **P** processors are available

We can also think of this in terms of the program's DAG

Work = $\mathbf{T_1}$ = sum of run-time of all nodes in the DAG

- One processor/thread does everything
- $O(n)$ for simple maps and reductions

Span = $\mathbf{T_\infty}$ = run-time of most-expensive path in DAG

- Infinite army of processors/threads do everything; Still has to wait for earlier results
- $O(\log n)$ for simple maps and reductions

SOME MORE TERMS

Speed-up on **P** processors: $\mathbf{T_1 / T_P}$

Perfect linear speed-up: If speed-up is **P** as we vary **P**

- Means we get full benefit for each additional processor as in doubling **P,** halves running time
- This is usually our goal
- *Hard to get (sometimes impossible) in practice*

Parallelism is the maximum possible speed-up: $\mathbf{T_1/T_\infty}$

- At some point, adding processors won't help

Parallel algorithms is about decreasing span without increasing work too much

OPTIMAL T_P: THANKS FORKJOIN LIBRARY

So we know $\mathbf{T_1}$ and $\mathbf{T_\infty}$ but we want $\mathbf{T_P}$ (e.g., **P**=4)

So an *asymptotically* optimal execution would be:

$$\mathbf{T_P = \mathit{O}((T_1 / P) + T_\infty)}$$

The ForkJoin Framework gives an *expected-time guarantee* of asymptotically optimal!

DIVISION OF RESPONSIBILITY

Our job as ForkJoin Framework users:

- Pick a good parallel algorithm and implement it
- Its execution creates a DAG of things to do
- *Make all the nodes small(ish) and approximately equal amount of work*

The framework-writer's job:

- Assign work to available processors to avoid idling
- Keep constant factors low
- Give the expected-time optimal guarantee assuming framework-user did his/her job

$$T_P = O((T_1 / P) + T_\infty)$$

EXAMPLES: $T_P = O((T_1 / P) + T_\infty)$

Algorithms seen so far (e.g., sum an array):

If $\mathbf{T_1} = O(n)$ and $\mathbf{T_\infty} = O(\texttt{log}\ n)$

→ $\mathbf{T_P} = O(n/\mathbf{P} + \texttt{log}\ n)$

Suppose instead:

If $\mathbf{T_1} = O(n^2)$ and $\mathbf{T_\infty} = O(n)$

→ $\mathbf{T_P} = O(n^2/\mathbf{P} + n)$

Of course, these expectations ignore any overhead or memory issues

CS435 Distributed systems

AMDAHL'S LAW

Dr. Basit Qureshi

PhD FHEA SMIEEE MACM

www.drbasit.org

AMDAHL'S LAW

In practice, much of our programming typically has parts that parallelize well

- Maps/reductions over arrays and trees

And also parts that don't parallelize at all

- Reading a linked list
- Getting/loading input
- Doing computations based on previous step

AMDAHL'S LAW

Let ***work*** (time to run on 1 processor) be 1 unit time

If **S** is the portion of execution that cannot be parallelized (Serial), then we can define T_1 as:

$$\mathbf{T_1 = S + (1\text{-}S)/1 = 1}$$

If we get perfect linear speedup on *the parallel portion*, then we can define T_P as:

$$\mathbf{T_P = S + (1\text{-}S)/P}$$

Thus, the overall speedup with **P** processors is (Amdahl's Law):

$$\mathbf{T_1 / T_P = 1 / (S + (1\text{-}S)/P)}$$

And the parallelism (infinite processors) is:

$$\mathbf{T_1 / T_\infty = 1 / S}$$

AMDAHL'S LAW

Amdahl's Law: $\mathbf{T_1 / T_P = 1 / (S + (1\text{-}S)/P)}$

$\mathbf{T_1 / T_\infty = 1 / S}$

Suppose 33% of a program is sequential:

- Then a billion processors won't give a speedup over 3

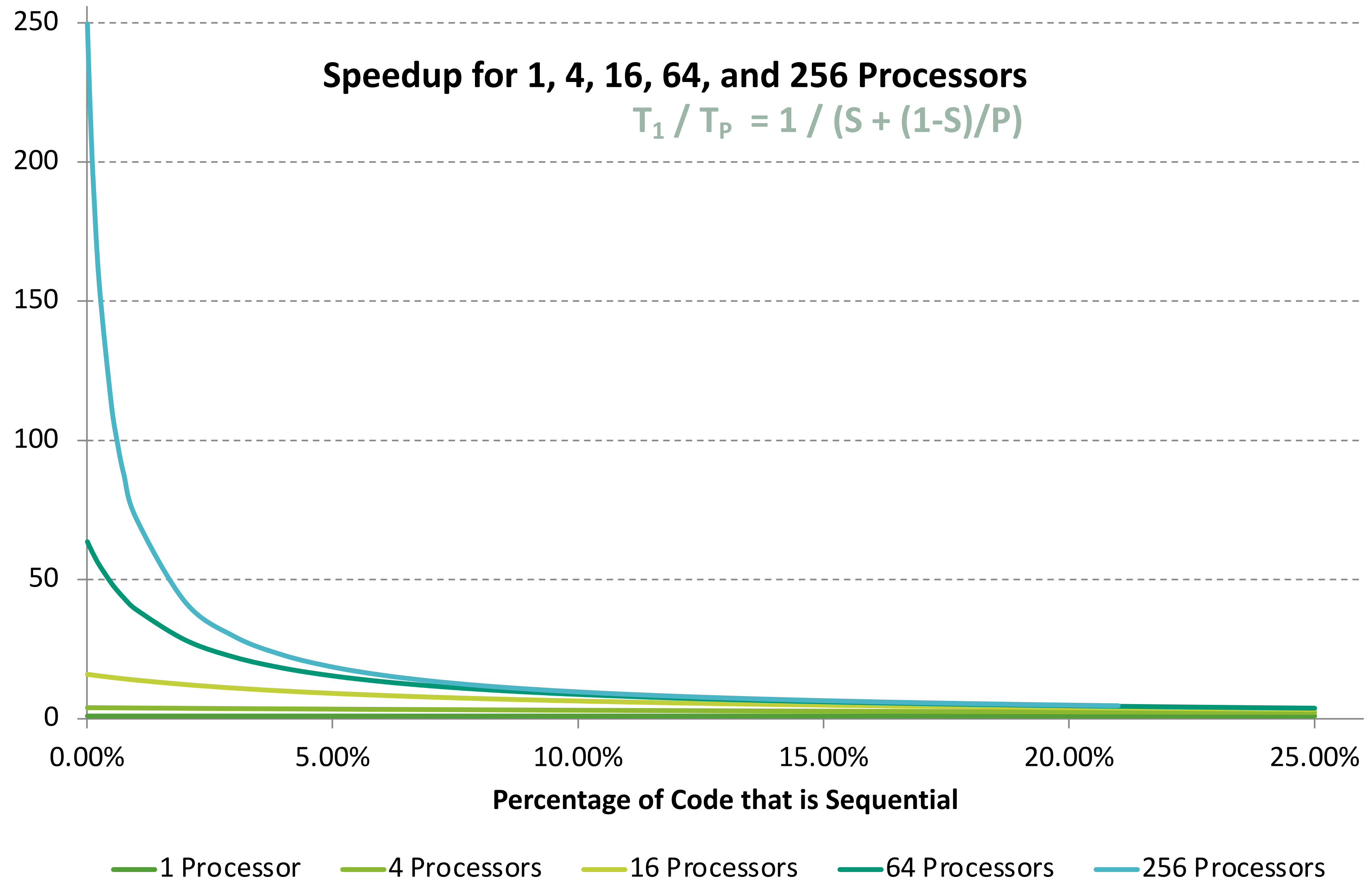
Suppose you miss the good old days (1980-2005) where 12 years or so was long enough to get 100x speedup

- Now suppose in 12 years, clock speed is the same but you get 256 processors instead of just 1
- For the 256 cores to gain ≥100x speedup, we need

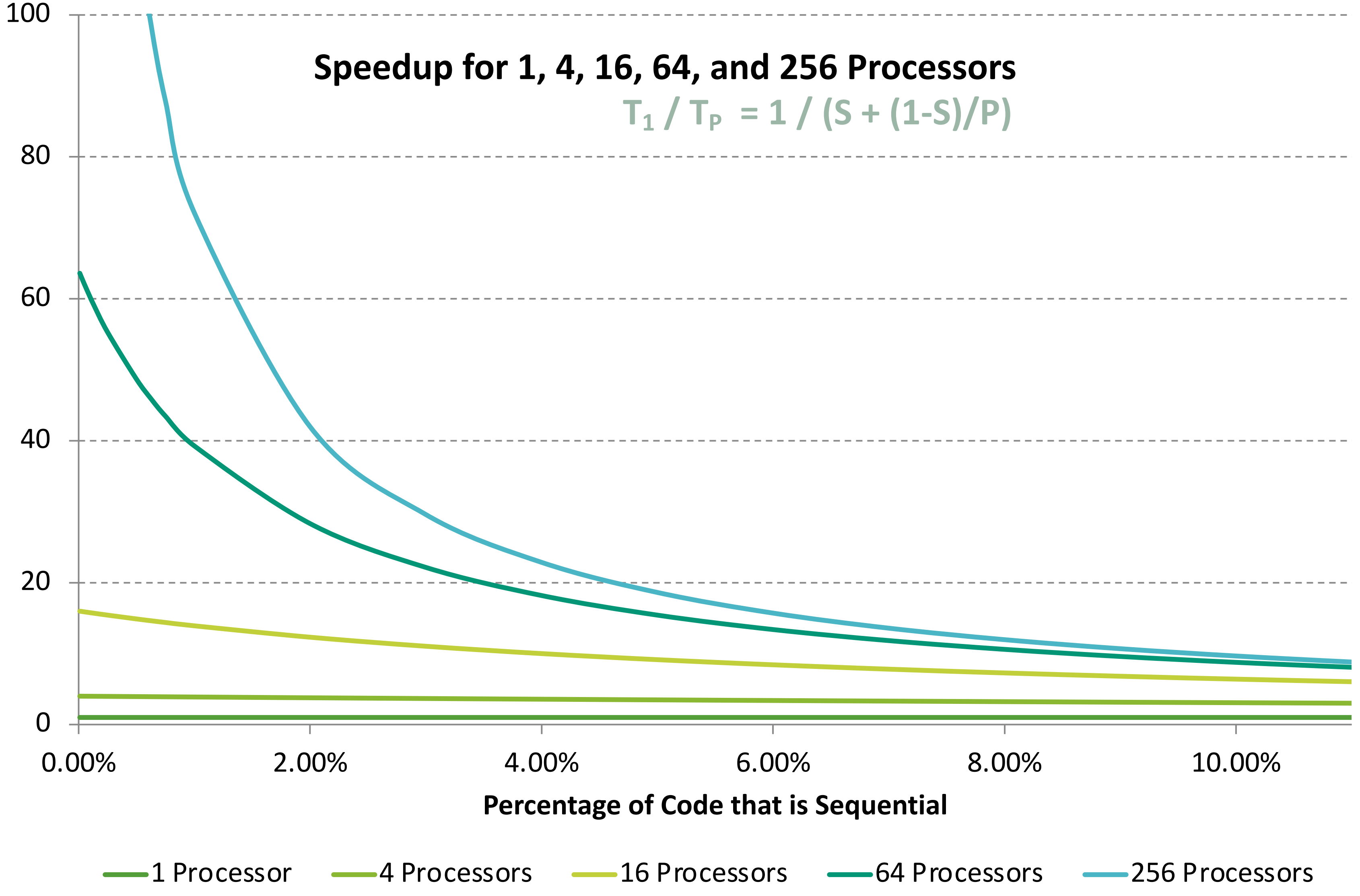
$$100 \leq 1 / (\mathbf{S} + (1\text{-}\mathbf{S})/256)$$

Which means $\mathbf{S} \leq .0061$ or 99.4% of the algorithm must be perfectly parallelizable!!

A PLOT YOU HAVE TO SEE



A PLOT YOU HAVE TO SEE (ZOOMED IN)



ALL IS NOT LOST

Amdahl's Law is 

- Doesn't mean additional processors are worthless!!

We can always search for new parallel algorithms

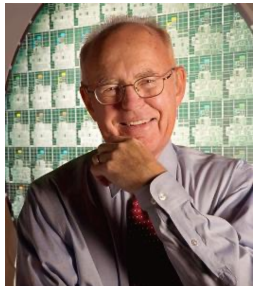
- We will see that some tasks may seem inherently sequential but can be parallelized

We can also change the problems we're trying to solve or pursue new problems

- Example: Video games/CGI use parallelism
  - But not for rendering 10-year-old graphics faster
  - They are rendering more beautiful(?) monsters

A FINAL WORD ON MOORE AND AMDAHL

Although we call both of their work laws, they are very different entities



Moore’s "Law" is an *observation* about the progress of the semiconductor industry:

- Transistor density doubles every ≈18 months

Amdahl’s Law is a mathematical theorem

- Diminishing returns of adding more processors

Very different but incredibly important in the design of computer systems

ASSIGNMENT 1

Due: Thursday, this week

CS435: Distributed Systems (242)

Assignment 1: A Parallel Word Frequency Calculator

Due: Thursday, this week **Weight (3 points)**

deepseek

Into the unknown

DeepSeek is an advanced AI model that excels in understanding and generating human-like text by leveraging deep learning techniques, context, and semantic relationships. It goes beyond basic word patterns to deliver more coherent and contextually accurate responses.

A word frequency calculator analyzes text to determine how often each word appears, helping models understand word patterns and distributions. This is key for generating coherent and contextually appropriate responses. By prioritizing high-frequency words, which are often more relevant, DeepSeek improves its ability to predict word combinations and produce natural-sounding text. Unlike simpler models, DeepSeek leverages advanced deep learning techniques, considering context, syntax, and semantics, going beyond basic word patterns.

For large datasets, calculating word frequency across multiple files can be time-consuming. This assignment/task involves creating a parallel word frequency calculator in Java using the Fork/Join framework. The program will read multiple text files from a directory and compute word frequencies, using parallel processing to speed up calculations across files.

The skeleton file assignment1.java given to you has:

a. A **main** method contains TO DO tasks. Follow the previous tutorial to complete the TO DO tasks. The files are read by the threads in parallel. The compute method in the **ComputeFrequency** class will compute the frequency values for each text file. The main method will output those values to the console (serial and not parallel).

b. A class **ComputeFrequency** is given to implement the Fork/Join Task: This class will represent a task that can be submitted to the Fork/Join pool. It will take in an array with text file names; a starting index and ending index to specify which files to process. It will then apply the recursive fork, where for each forked task compute method is called. The compute method implements the word frequency computation logic.

Download the datasets to test your work

- Dataset 1: 12 text files of varying size
- Dataset 2: 2000 text files of varying size

To verify the correctness of your program, the following are the frequencies of keywords "the" and "and" in the Datasets:

	frequencies	
Keyword	**the**	**and**
Dataset1	166	66
Dataset2	31519	12072

CS435 Distributed systems

MAP REDUCE FRAMEWORK

Most of the content in these set of slides is based on Paul Krzyzanowski Distributed Systems course taught at Rutgers University

Dr. Basit Qureshi

PhD FHEA SMIEEE MACM

www.drbasit.org

WHAT ABOUT DISTRIBUTED PARALLEL PROCESSING?

Suppose you need to perform some computation on a huge amount of data (1 petabyte = 1 million Gibabyte)

- Even small amounts of processing can add up
  - Break the workload in small chunks. Each chunk takes 1 MB.
  - Assume each subtask takes 100ms for 1MB chunk
  - 1 billion chunks!
  - 100ms per data item × 1 billion items = 1157 days of computation!

WHAT ABOUT DISTRIBUTED PARALLEL PROCESSING?

- Suppose you need to perform some computation on a huge amount of data (1 petabyte = 1 million Gibabyte)
  - Solution?
    - Break the work up so lots of computers can work on just parts of the data
      - Split the workload among 10,000 computers ⇒ 2.7 hours of computation
  - Put the data on a file server?
    - More data than you can fit on one system
    - Disk bandwidth will be an issue
      - if you read an SSD at 500 MB/s, it becomes a bottleneck on the network
    - Shared bandwidth
      - 10,000 systems will get data at < 5KB/s

- We need to distribute the workload and the data

WHAT ABOUT DISTRIBUTED PARALLEL PROCESSING?

- Work with a Distributed Systems to solve the problem!
- Issues!!!
  - Split the data in smaller chunks (Shards)
  - Allocate chunks to processes
  - Remotely control the processes to run on the servers
  - Partition the work among processes
  - Assign processes to servers, allocate data chunks
  - Lookout for communication problems
  - Lookout for failure
  - Manage and re-start failed processes
  - Process and collect the results from different processes running on different servers
- Map Reduce!
  - A workhorse for distributed batch processing

Complex!

MAPREDUCE

"*MapReduce is a programming model and an associated implementation for processing and generating large data sets*"

- Programming model
  - Abstractions to express simple computations
- Library
  - Takes care of the gory stuff: Parallelization, Fault Tolerance, Data Distribution and Load Balancing

MAPREDUCE

- Master/worker approach
  - Master
    - initializes data set and splits it according to # of workers
    - Sends each worker a sub-array of data
    - Receives the results from each worker
  - Worker
    - Receives a sub-array from master
    - Performs processing
    - Sends results to master

MAPREDUCE

- Created by Google in 2004 Jeffrey Dean and Sanjay Ghemawat
- Inspired by LISP
  - Map(function, set of values)
    - Applies function to each value in the set

    (map 'length '(() (a) (a b) (a b c))) ⇒ (0 1 2 3)
  - Reduce(function, set of values)
    - Combines all the values using a binary function (e.g., +)

    (reduce #'+ '(1 2 3 4 5)) ⇒ 15

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple compu-

https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

MAPREDUCE

- Framework for parallel computing
- Programmers get simple API
- Don’t have to worry about handling
    - Parallelization
    - Data distribution
    - Load balancing
    - Fault tolerance
    - Monitoring

User can process huge amounts of data (terabytes and petabytes) on thousands of processors

MAPREDUCE

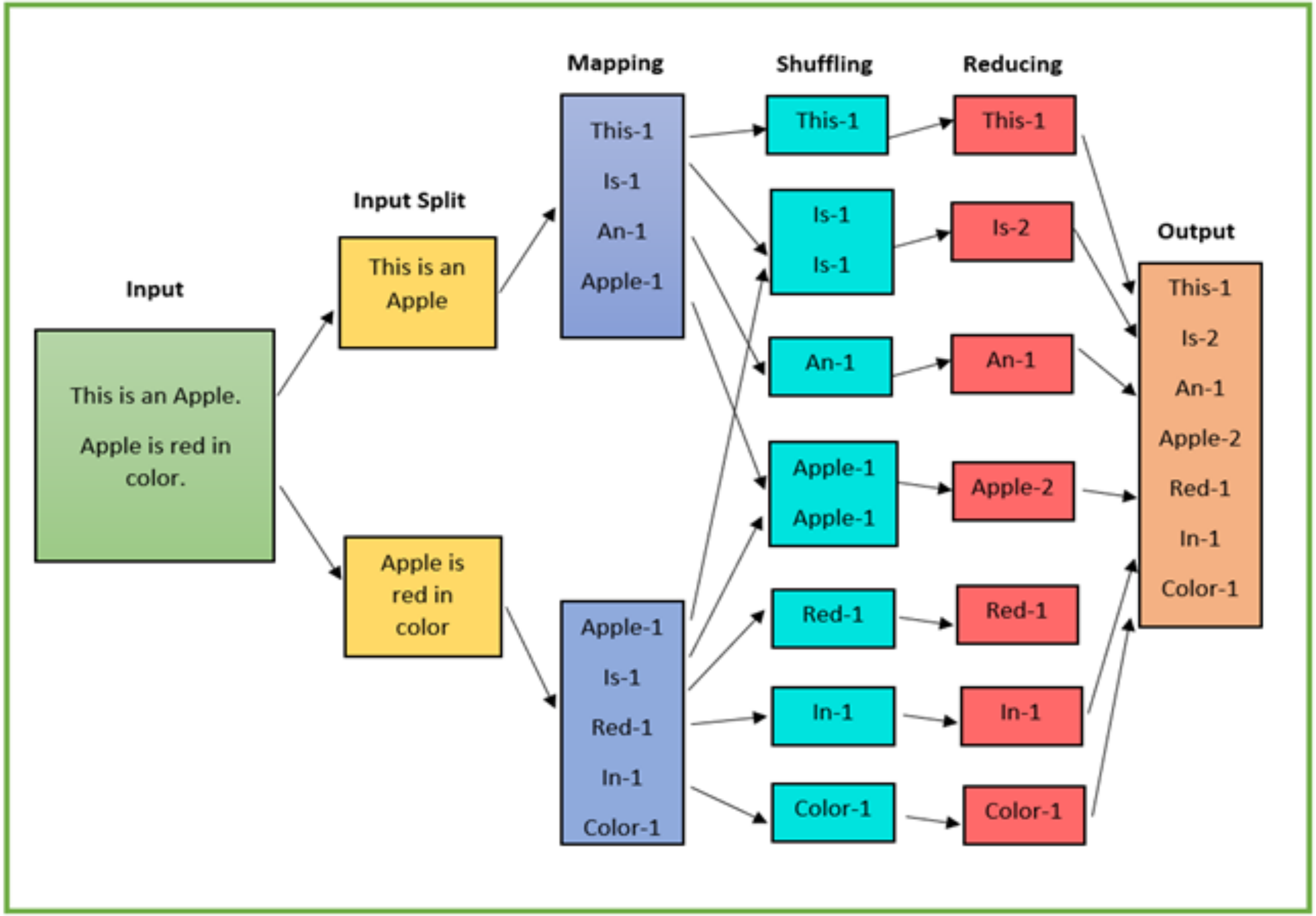
Who works with MapReduce?

- Apache Hadoop MapReduce
  - Most common Open source implementation
- Amazon Elastic: Runs Hadoop on Amazon EC2
- Microsoft Azure HDInsight
- Google Cloud MapReduce for App Engine

MAPREDUCE

- Map:
  - Grab the relevant data from the source
  - User function gets called for each chunk of input
  - Spits out (key, value) pairs
- Reduce:
  - Aggregate the results
  - User function gets called for each unique key with all values corresponding to that key

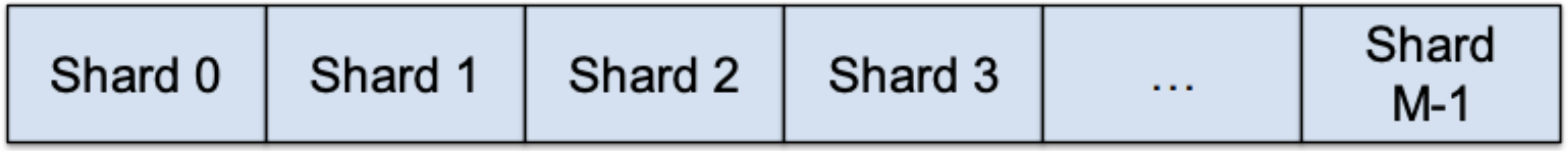
MAP REDUCE: 7-STEP PROCESS



MAPREDUCE

Step 1: Split

- Split input files into chunks (shards/splits). Size depends on the file system (typically 128MB)



Input data

Divided into *M* shards (splits)

MAPREDUCE

Step 2: Fork processes

- Start up many copies of the program on a cluster of machines
  - One master: scheduler & coordinator
  - Lots of workers
- Tasks
  - Map: each works on a shard
  - Reduce: each works on intermediate files
  - Partitions, Maps and Reduce tasks are defined by the users

MAPREDUCE

Step 3: Each Map task

- Reads contents of the input shard
- Parses key/value pairs out of the input data
- Passes each pair to a user-defined map function
  - Produces intermediate key/value pairs
  - These are buffered in memory

- MapReduce supports multiple types of files stored in various locations

MAPREDUCE

Step 4: Create Intermediate files

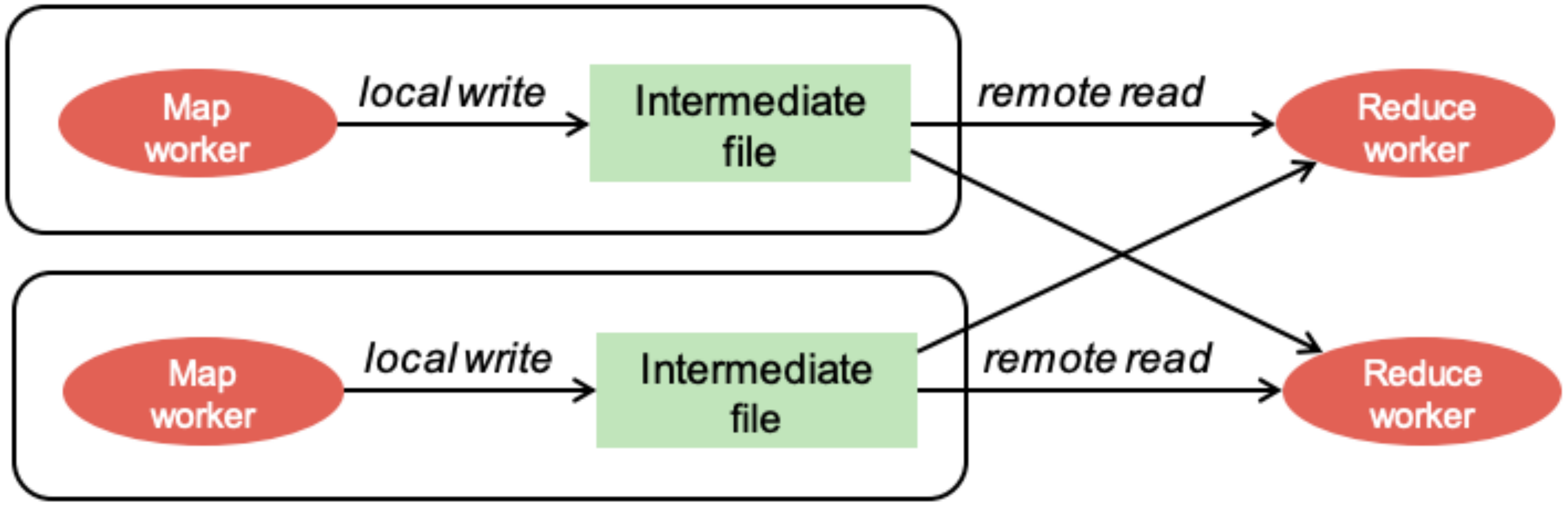
1. Intermediate key/value pairs produced by the user's map function buffered in memory and are periodically written to the local disk
   - Partitioned into *R* regions by a partitioning function
2. Notifies master when complete
   - Passes locations of intermediate data to the master
   - Master forwards these locations to the reduce worker
3. Map key-value data will be processed by Reduce workers
   - The user's Reduce function will be called once per unique key generated by Map.
4. We first need to group all the (key, value) data by keys and decide which Reduce worker processes which set of keys
   - The Reduce worker will later sort the values within each keys

Default function to identify a reduce worker: hash(key) mod R

MAPREDUCE

Step 5: Reduce: Shuffle

- Reduce worker is notified by the master about the location of intermediate files for its partition
  - Shuffle: Uses Remote Procedure Calls (RPCs) to read the data from the local disks of the map workers
  - Sort: When the reduce worker gets all the (key, value) data for its partition from all workers
    - It sorts the data by the keys
    - All occurrences of the same key are grouped together



MAPREDUCE

Step 6: Reduce: Sort

- The sort phase grouped data by keys
  - This makes it easy to identify all the values from all the map workers that are associated with each key
- The user's Reduce function is given the key and the set of intermediate values for that key

< key, (value1, value2, value3, value4, …) >

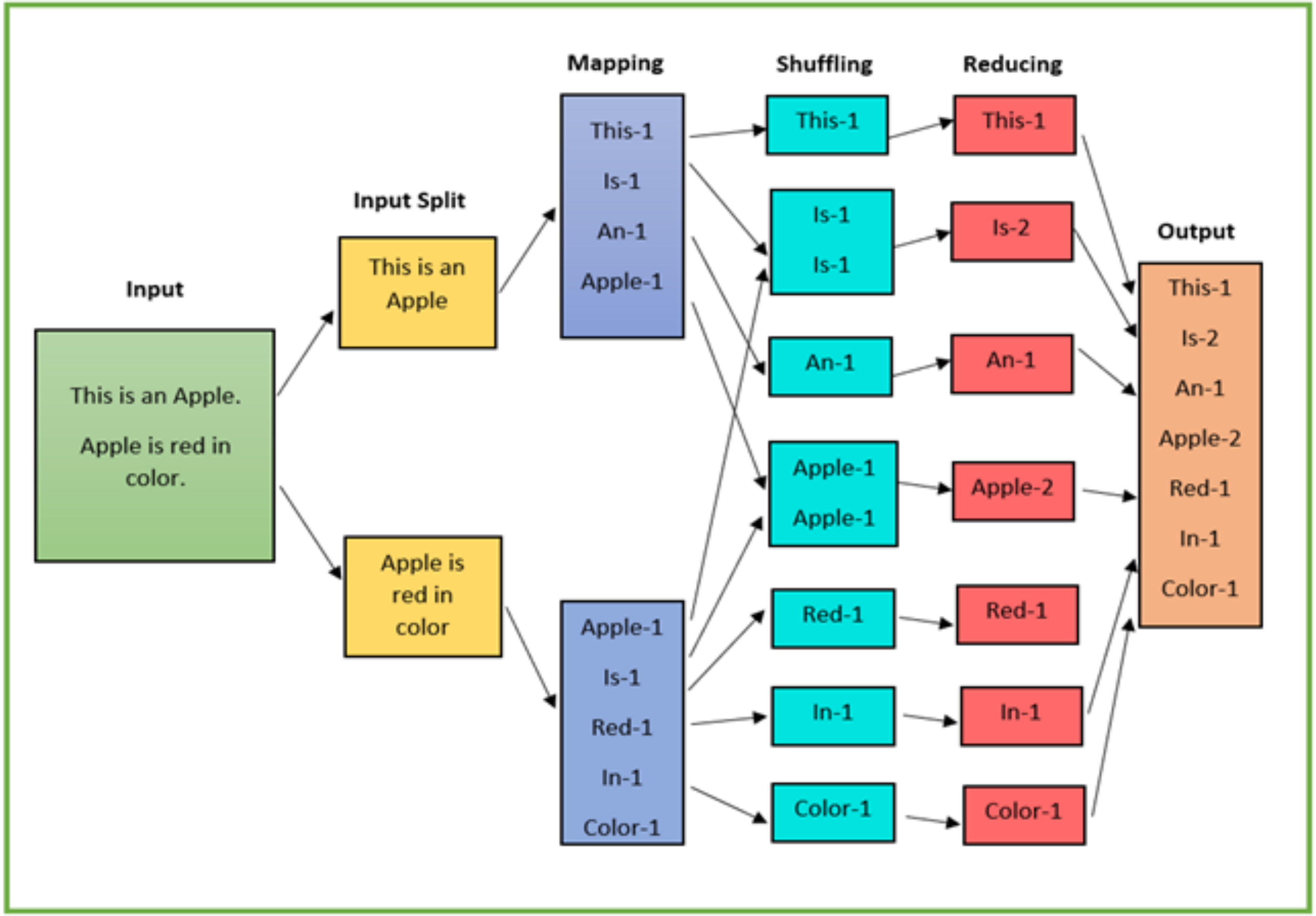
- The output of the Reduce function is appended to an output file

MAPREDUCE

Step 7: Return

- When all map and reduce tasks have completed, the master wakes up the user program
- The MapReduce call in the user program returns and the program can resume execution
- Output of MapReduce is available in R output files

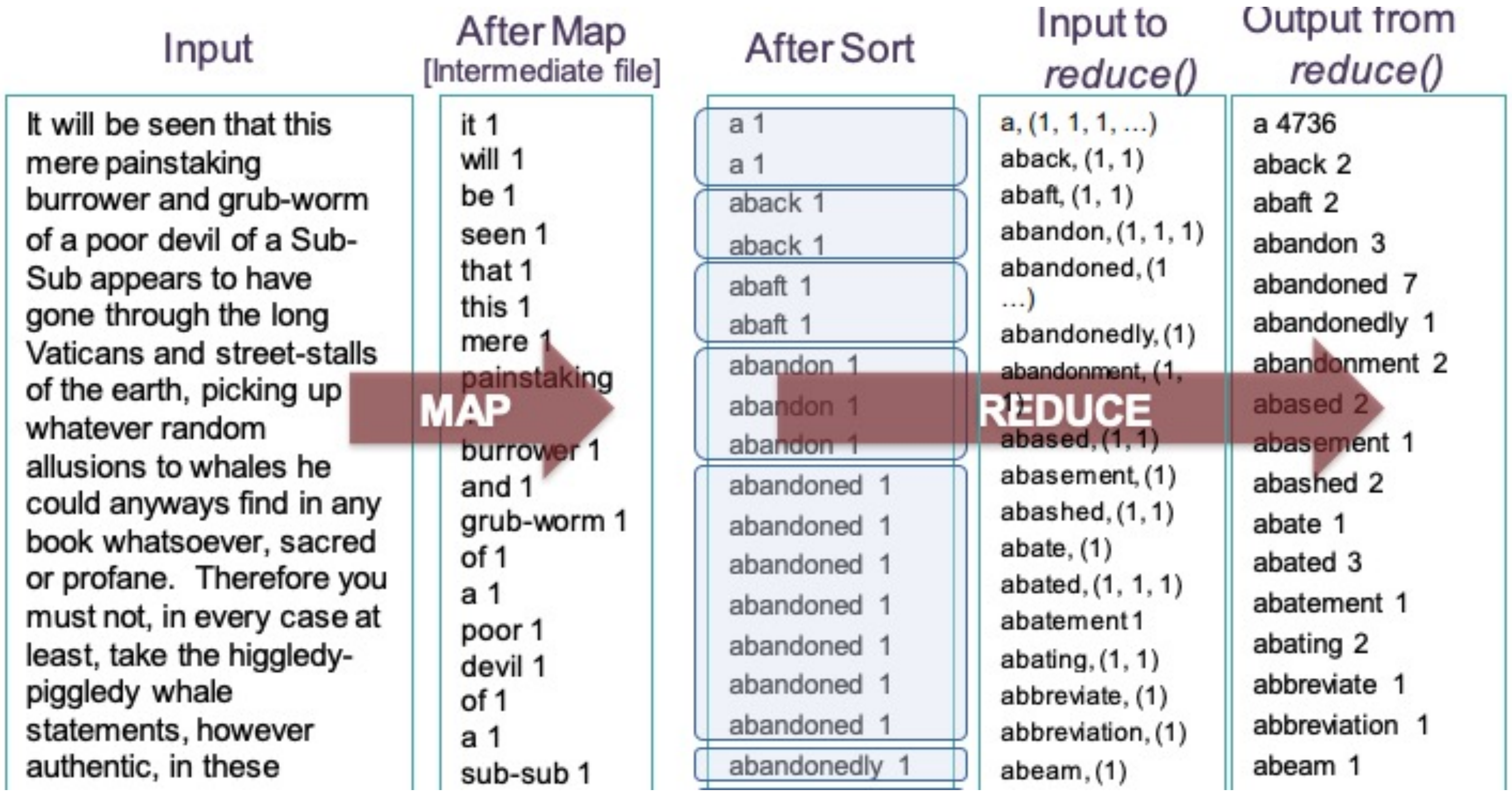
MAP REDUCE: EXECUTION FLOW EXAMPLE



MAPREDUCE: EXAMPLES

WordCount

Count the # occurrences of each word in a collection of documents



MAPREDUCE: EXAMPLES

```
map(String input_key, String input_value):
    // input_key: document name
    // input_value: document contents
    for each word w in input_value:
      EmitIntermediate(w, "1");
```

<“Sam”, “1”>, <“Apple”, “1”>, <“Sam”, “1”>,
<“Mom”, “1”>, <“Sam”, “1”>, <“Mom”, “1”>,

```
  reduce(String output_key, Iterator intermediate_values):
    // output_key: a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
      result += ParseInt(v);
    Emit(AsString(result));
```

<“Sam” , [“1”,”1”,”1”]>, <“Apple” , [“1”]>,
<“Mom” , [“1”, “1”]>

“3”
“1”
“2”

MAPREDUCE: EXAMPLES

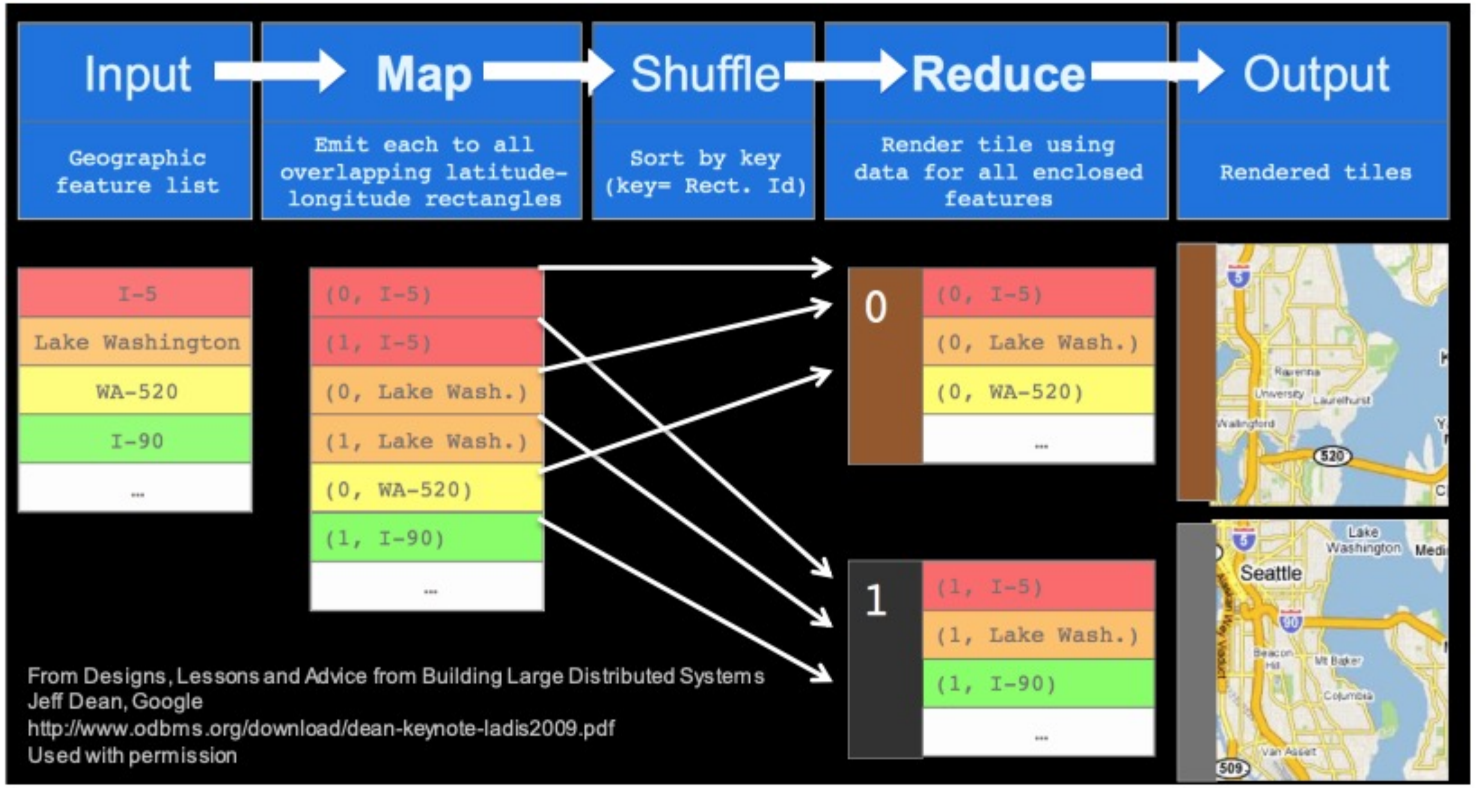
Web crawlers

- Search for words in lots of documents
  - Map: emit a line if it matches a given pattern
  - Reduce: just copy the intermediate data to the output
- Find the count of each URL in web logs
  - Map: process logs of web page access; output
  - Reduce: add all values for the same URL
- Find the frequency of each URL in web logs
  - Run 1: just count total URLs
  - Run 2: just like URL count but now we stored total_urls
- Find where page links come from
  - Map: output for each link to target in a page source
  - Reduce: concatenate the list of all source URLs associated with a target Output

MAPREDUCE: EXAMPLES

Other examples:

- Stock performance summary – Find average daily gain of each company from 1/1/2010 – 12/31/2020
- Average salaries in regions – Show zip codes where average salaries are in the ranges:(1) < $100K (2) $100K … $500K (3) > $500K



MAPREDUCE: EXAMPLES

Other examples:

- Social Media: Companies like **Facebook** and **Twitter** employ MapReduce for tasks such as user analytics, trend analysis, and recommendation systems.

- Genomic Data Processing: The genomics field utilizes MapReduce to process and analyze large volumes of genetic data for research and healthcare purposes.

- Log Processing: Log files generated by systems, servers, and applications can be efficiently processed and analyzed using MapReduce for debugging and monitoring.

- Natural Language Processing: In NLP tasks, MapReduce is used to process and analyze text data, such as sentiment analysis, topic modeling, and language translation.

MAPREDUCE

Benefits

- Fault Tolerance
  - Master pings each worker periodically
    - If no response is received within a certain time, the worker is marked as failed
    - Map or reduce tasks given to this worker are reset back to the initial state and rescheduled for other workers
- Locality
  - Input and Output data comes from the filesystem
    - MapReduce (often) runs on chunkservers
    - Keep computation close to the files if possible

MAPREDUCE

In practice

- MapReduce was used to process webpage data collected by Google's crawlers. Determine the site's PageRank.
  - It took 8 hours for a run!!
  - Results were moved to search servers
  - This was done continuously
  - Now: Can’t wait for 8 hours delay. The dynamic web changes!
- Most data is not stored as simple files
  - B-trees, tables, SQL databases, memory-mapped key-values
- We don't usually use textual data: it's slow & hard to parse
    - Most I/O gets encoded with Protocol Buffers

MAPREDUCE

- Batch-oriented
  - Not suited for near-real-time processes
  - Cannot start a new phase until the previous has completed
    - Reduce cannot start until all Map workers have completed
  - Suffers from “stragglers”
    - workers that take too long (or fail)
  - This was done continuously
  - MapReduce is still useful but there are also other options

SUMMARY

- Fork Join with Divide and Conquer
- Speedup and Amdahl's Law
- MapReduce for distributed parallel processing