

---

# COMP 322: Principles of Parallel Programming

Vivek Sarkar  
Department of Computer Science  
Rice University  
vsarkar@rice.edu



# Summary of Last Lecture

---

- HJ abstract execution metrics
- Hands-on examples with the following constructs
  - forall
  - async statement
  - finish statement
  - async expression with `future<T>` return value
  - force operation on `future<T>`

# Programming Assignment #1: Generalized Scan

---

- **Assignment**

- Implement Generalized Scan library in HJ for general problem size  $N$  (need not be a power of two)
  - Library should support clients that implement `init()`, `combine()`, `accum()`, `scanGen()` methods
- Test the correctness of your implementation on two applications of Generalized Scan with input int arrays of size  $N = 10, 100, 1000$ 
  - Partial Sums
  - Index of Last Occurrence (pg. 124)
- Measure and report the ideal parallelism in your algorithm using HJ's abstract execution metrics for  $N = 10, 100, 1000$ 
  - Add 1 as a "local op" for each + operation in Partial Sums
  - Add 1 as a "local op" for each max operation in Last Occurrence

# Programming Assignment #1: Generalized Scan (contd)

---

- **Submission should include**
    - A brief report (in Word/PDF format) summarizing the design of your library, and the abstract metrics obtained for the two applications
    - A tar/zip file containing your source code including test programs, test input, and test output
  - **Grading criteria**
    - 30% for report (includes clarity of writing & explanations)
    - 40% for code correctness
    - 30% for abstract metrics
  - **Deadline**
    - Your submission should be sent by email to the TA, [cs20@rice.edu](mailto:cs20@rice.edu), by 5pm on Tuesday, Oct 6<sup>th</sup> (3 weeks from Sep 15<sup>th</sup>)
    - Late submissions will be penalized at 10% day
  - **Other**
    - This assignment should be done individually. You may discuss class material with other students, but your solution should be your own.
-

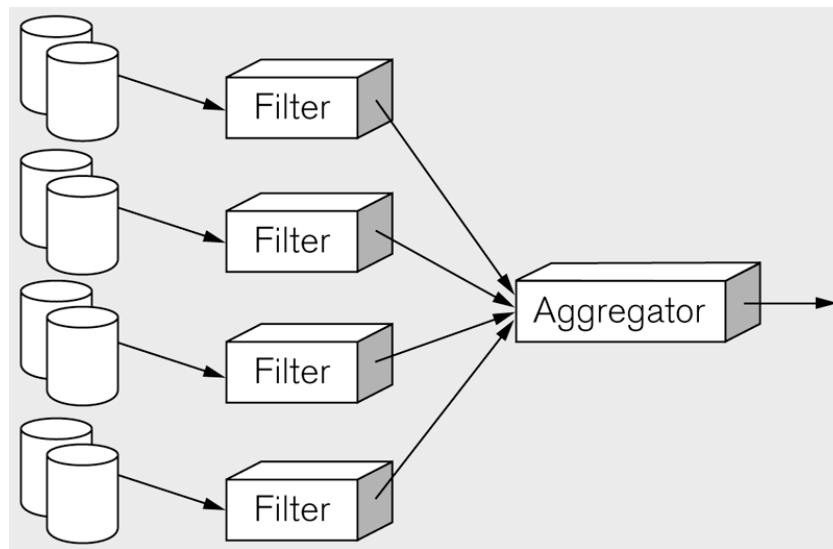
# Acknowledgments

---

- Course text: “Principles of Parallel Programming”, Calvin Lin & Lawrence Snyder
  - Includes resources available at <http://www.pearsonhighered.com/educator/academic/product/0,3110,0321487907,00.html>
- “Data Analysis with MapReduce”, John Mellor-Crummey, COMP 422 Lecture 24, Spring 2009
- “MapReduce: Simplified Data Processing on Large Clusters,” [Jeffrey Dean](#) and [Sanjay Ghemawat](#). 6th Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.
  - <http://labs.google.com/papers/mapreduce.html>
- Introduction to Parallel Programming with MapReduce, Google Code University
  - <http://code.google.com/edu/parallel/mapreduce-tutorial.html>
- Seminar presentation on “MapReduce Theory and Implementation” by Christophe Bisciglia et al. Summer 2007
  - <http://code.google.com/edu/submissions/mapreduce/llm3-mapreduce.ppt>
- Hadoop Map/Reduce Tutorial
  - [http://hadoop.apache.org/core/docs/r0.19.1/mapred\\_tutorial.html](http://hadoop.apache.org/core/docs/r0.19.1/mapred_tutorial.html)
- “Interpreting the Data: Parallel Analysis with Sawzall”, Rob Pike, Sean Dorward, Robert Griesemer, Sean Quinlan, Scientific Programming journal, 2005
  - <http://labs.google.com/papers/sawzall.html>

# MapReduce Framework

- Pioneered by Google --- extension of Generalized Reduce (Chapter 5) applied to data-center-scale distributed systems
  - A leaf in the reduce tree may correspond to one or more disks in a distributed file system
  - Support by an interpreted procedural language, Sawzall
- Filter = computation specified by map function
- Aggregator = computation specified by reduce function



Source: Figure 10.6 from textbook

# Putting Data-Center-Scale in Perspective

---

“Although it has been deployed for only about 18 months, Sawzall has become one of the most widely used programming languages at Google. There are over two thousand Sawzall source files registered in our source code control system and a growing collection of libraries to aid the processing of various special-purpose data sets. Most Sawzall programs are of modest size, but the largest are several thousand lines long and generate dozens of multi-dimensional tables in a single run.

One measure of Sawzall's utility is how much data processing it does. We monitored its use during the month of March 2005. During that time, on one dedicated Workqueue cluster with 1500 Xeon CPUs, there were 32,580 Sawzall jobs launched, using an average of 220 machines each. While running those jobs, 18,636 failures occurred (application failure, network outage, system crash, etc.) that triggered rerunning some portion of the job. The jobs read a total of  $3.2 \times 10^{15}$  bytes of data (2.8PB) and wrote  $9.9 \times 10^{12}$  bytes (9.3TB) (demonstrating that the term “data reduction” has some resonance). The average job therefore processed about 100GB. The jobs collectively consumed almost exactly one machine-century.”

--- “Interpreting the Data: Parallel Analysis with Sawzall”, Rob Pike, Sean Dorward, Robert Griesemer, Sean Quinlan, *Scientific Programming journal*, 2005

---

# Motivation: Large Scale Data Processing

---

- Want to process terabytes of raw data
  - documents found by a web crawl
  - web request logs
- Produce various kinds of derived data
  - inverted indices
    - e.g. mapping from words to locations in documents
  - various representations of graph structure of documents
  - summaries of number of pages crawled per host
  - most frequent queries in a given day
  - ...
- Input data is large
- Need to parallelize computation so it takes reasonable time
  - need hundreds/thousands of CPUs
- Need for fault tolerance

# Solution: MapReduce Programming Model

---

- Inspired by **map** and **reduce** primitives in lisp
  - mapping a function **f** over a sequence **x y z** yields **f(x) f(y) f(z)**
  - reduce function combines sequence elements using a binary op
- Many data analysis computations can be expressed as
  - applying a **map** operation to each logical input record
    - produce a set of intermediate (key, value) pairs
  - applying a **reduce** to all intermediate pairs with same key
- Simple programming model using an application framework
  - user supplies **map** and **reduce** operators
  - messy implementation details handled by the framework
    - parallelization
    - fault tolerance
    - data distribution
    - load balance

# Example: Count Word Occurrences

---

## Pseudo Code

**map(String input\_key, String value):**

*// input\_key: document name*

*// value: document contents*

**for each** word *w* **in** value:

**EmitIntermediate**(*w*, "1"); *// count 1 for each occurrence*

**reduce(String output\_key, Iterator values):**

*// output\_key: a word*

*// values: a list of counts*

**int** result = 0;

**for each** *v* **in** values:

result += **ParseInt**(*v*);

**Emit**(**AsString**(result));

# Applying the Framework

---

- Fill in a **MapReduceSpecification** object with
  - names of input and output files
  - map and reduce operators
  - optional tuning parameters
- Invoke the **MapReduce** framework to initiate the computation
  - pass the specification object as an argument

# What about Data Types?

---

- Conceptually, map and reduce have associated types
  - map (k1, v1) → list(k2, v2)
  - reduce(k2, list(v2)) → list(v2)
- Input keys and values
  - drawn from a different domain than output keys and values
- Intermediate keys and values
  - drawn from the same domain as output keys and values
- Google MapReduce C++ implementation
  - passes strings to all user defined functions: simple and uniform
  - have user convert between strings and appropriate types
- Java-based implementations
  - Use Object instead of String
  - Define framework as interface or super-class

# MapReduce Examples

---

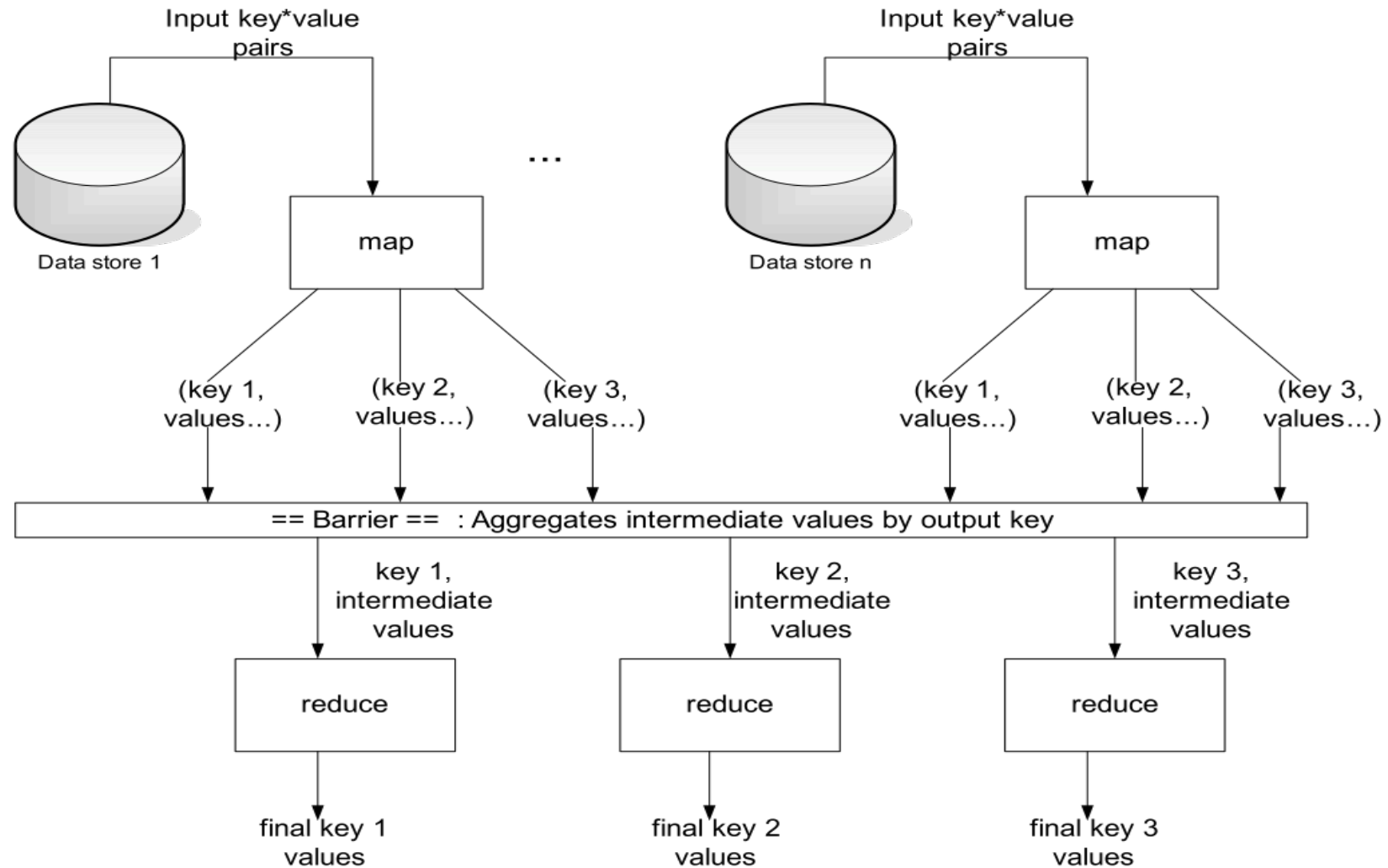
- **Distributed grep**
  - **map**: emits a line if it matches a supplied pattern
  - **reduce**: identity function - copies intermediate data to the output
- **Count of URL access frequency**
  - **map**: processes logs of web page requests, outputs a sequence of  $\langle \text{URL}, 1 \rangle$  tuples
  - **reduce**: adds together all values for the same URL and emits a  $\langle \text{URL}, \text{total count} \rangle$  pair
- **Reverse web-link graph**
  - **map**: outputs  $\langle \text{target}, \text{source} \rangle$  pairs for each link to a target URL found in a page named source
  - **reduce**: concatenates the list of all source URLs associated with a given target URL
    - emits the pair:  $\langle \text{target}, \text{list of sources} \rangle$

# Implementation Considerations

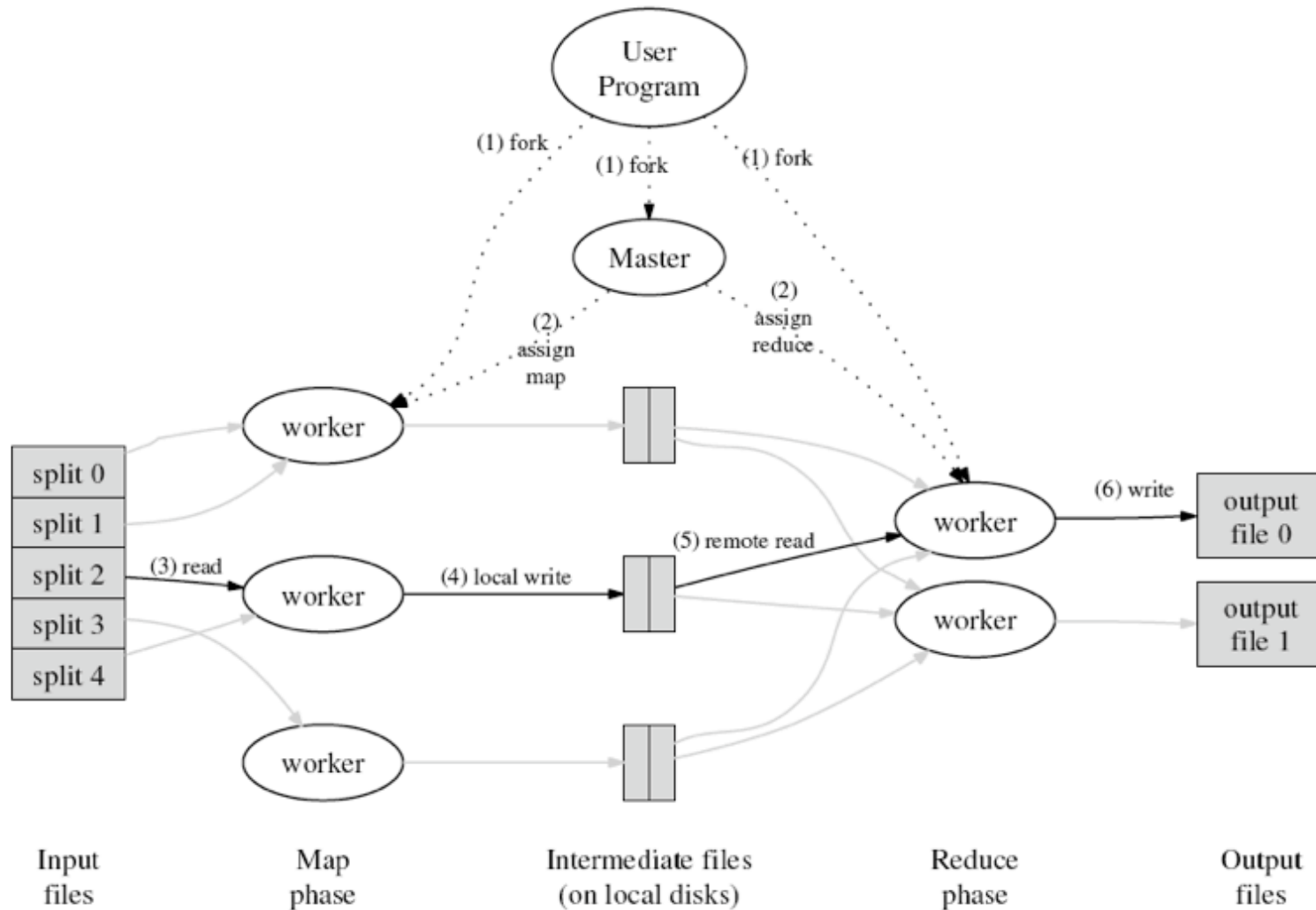
---

- Many different implementations are possible
- Right choice depends on environment, e.g.
  - small shared memory machine
  - large NUMA multiprocessor
  - larger collection of networked machines
- Google environment
  - dual-processor x86, Linux, 2-4GB memory
  - commodity network: 100Mb/1Gb Ethernet per machine
  - thousands of machines: failure common
  - storage:
    - inexpensive disks attached directly to machines
    - distributed file system to manage data on these disksreplication provides availability and reliability on unreliable h/w

# Logical Overview of Execution



# Execution Overview



# Execution Overview Details

---

## 1. Initiation

- MapReduce library splits input files into  $M$  pieces
  - typically 16-64 MB per piece (controllable by a parameter)
- starts up copies of the program on a cluster (1 master + workers)

2. Master assigns  $M$  map tasks and  $R$  reduce tasks to workers

3. Worker assigned a map task reads contents of input split

- parses key/value pairs out of input data and passes them to map
- intermediate key/value pairs produced by map: buffer in memory

4. Periodically write pairs to local disk; partition into  $R$  regions; pass locations of buffered pairs to master for reducers

5. Reduce worker uses RPC to read intermediate data from remote disks; sorts pairs by key

6. Iterates over sorted intermediate data; calls reduce; appends output to final output file for this reduce partition

7. When all is complete, user program is notified

---

# Master Data Structures

---

- For each map and reduce task, store
  - state (idle, in-progress, completed)
  - identity of worker machine (for non-idle tasks)
- For each completed map task
  - store locations and sizes of R intermediate files produced by map
  - information updated as map tasks complete
  - pushed incrementally to workers that have in-progress reduce tasks

# Task Granularity

---

- Divide map phase into  $M$  pieces; reduce phase into  $R$  pieces
- Ideally,  $M$  and  $R$  much larger than number of worker machines
- Dynamically load balance tasks onto workers
- Upon failure
  - the many map tasks performed by a worker can be distributed among other machines
- How big are  $M$  and  $R$ ?
- Master
  - makes  $O(M + R)$  scheduling decisions
  - keeps  $O(M * R)$  state in memory - in practice, 1 byte per pair

## Task granularity in practice

— often use  $M = 200K$ ,  $R = 5K$ , using 2000 worker machines

# Fault Tolerance: Worker Failure

---

- **Detecting failure**
  - master pings worker periodically
  - if no answer after a while, master marks worker as failed
- **Coping with failure**
  - any map tasks for worker reset to *idle* state; may be rescheduled
  - worker's completed map tasks re-execute on failure
    - data on local disk of failed worker is inaccessible
    - any reducers attempting to notified of the re-execution

## Fault tolerance in action

- network maintenance on a cluster caused groups of 80 machines at a time to become unreachable for several minutes
- master simply re-executed work for unreachable machines and continued to make forward progress

# Master Failure

---

- Master could periodically write checkpoints of master data structures
- If master dies, another could be recreated from checkpointed copies of its state
- In practice
  - only a single master
  - failure would be rare
  - implementation currently aborts MapReduce if master fails
  - client could check this condition and retry the computation

# Coping with “Stragglers”

---

- Problem: a slow machine at the end of the computation could stall the whole computation
- When a MapReduce is nearing completion, schedule backup executions of in-progress tasks

## Backup task execution in practice

- significantly reduces time for large MapReduce computations
- a sorting example took 44% longer without backup tasks

# Combiner

---

- When there is significant repetition in intermediate keys
  - e.g. instances of <the, 1> in word count output
- it is useful to partially merge data locally before sending it across the network to a reducer
- Combiner
  - function executed on each machine that performs a map task
  - typically the same code as the reducer
- Significantly speeds up “certain classes of MapReduce operations”
  - Like accum() function in Generalized Reduce interface

# MapReduce at Google

---

- Large-scale machine learning
- Clustering problems for Google News, Froogle
- Extraction of data to produce popular queries (Zeitgeist)
- Extracting properties of web pages
  - e.g. geographical location for localized search
- Large-scale graph computations
- Large-scale indexing
  - 2004: indexing crawled documents
    - data size > 20 TB
    - runs indexing as a sequence of 5-10 MapReduce operations
  - experience
    - applications smaller than ad-hoc indexing by 4x
    - readily modifiable because programs are simple
    - performance is good: keep conceptually distinct things separate rather than forcing them together
    - indexing is easy to operate: e.g. fault tolerant; easy to improve performance by adding new machines

# Performance Anecdotes I

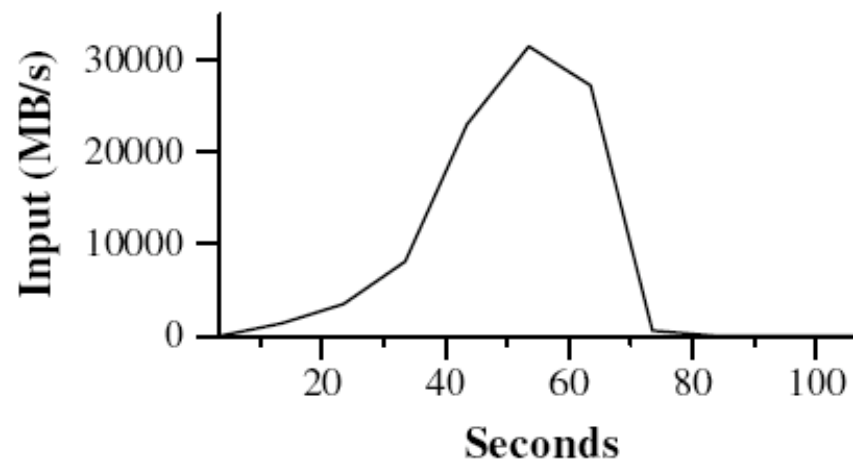
---

- **Cluster**
  - ~1800 nodes
    - 2 2GHz Xeon, 4GB memory, 2 160GB IDE drives, 1Gb Ethernet
  - network 2-level tree-shaped switched Ethernet
    - ~100-200Gbps aggregate bandwidth at the root
- **Benchmarks executed on a roughly idle cluster**

grep: scan through  $10^{10}$  100-byte records (~1TB) for a relatively rare 3-character pattern

—split input into 64MB pieces,  $M=15000$ ,  $R = 1$  (one output file)

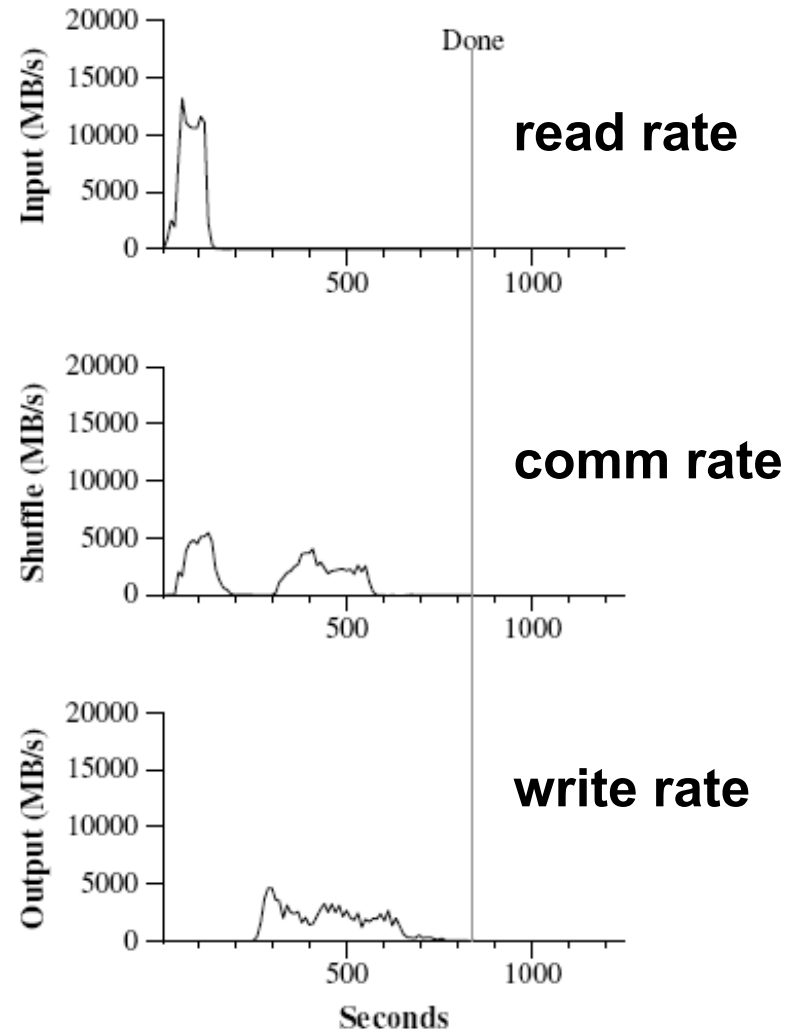
—time = ~150 seconds



# Performance Anecdotes II

sort  $10^{10}$  100-byte records  
(~1TB)

- consists of < 50 lines of user code
- split input into 64MB pieces,  $M=15000$ ,  $R=4000$
- partitioning function uses initial bytes to put it into one of  $R$  pieces
- input rate higher than shuffle or output rate: on local disk
- output phase makes 2 replica for availability
- time = 891 seconds



# MapReduce is a Success

---

- **Reasons for its success**
  - easy even for users lacking experience with parallel systems
    - insulates user from complexity
      - parallelization, fault tolerance, locality opt., load balancing
  - large variety of computations expressible using MapReduce
    - sorting, data mining, machine learning, etc.
  - implementation scales to large commodity clusters
    - makes efficient use of thousands of machines
- **Lessons**
  - restricting programming model simplifies tackling
    - parallelization, fault tolerance, distribution
  - network bandwidth is a scarce resource
    - locality optimizations to save network bandwidth are important
      - read data from local disk; write intermediate data to local disk
  - redundant execution
    - reduces impact of slow machines, machine failures, data loss

# Full “Word Count” Example: Map

---

```
#include "mapreduce/mapreduce.h"

class WordCounter : public Mapper {
public:
    virtual void Map(const MapInput& input) {
        const string& text = input.value();
        const int n = text.size();
        for (int i = 0; i < n; ) {
            // Skip past leading whitespace
            while ((i < n) && isspace(text[i])) i++;
            // Find word end
            int start = i;
            while ((i < n) && !isspace(text[i])) i++;
            if (start < i) Emit(text.substr(start,i-
start),"1");
        }
    }
};

REGISTER_MAPPER(WordCounter);
```

# Full “Word Count” Example: Reduce

---

```
#include "mapreduce/mapreduce.h"

class Adder : public Reducer {
  virtual void Reduce(ReducerInput* input) {
    // Iterate over all entries with the
    // same key and add the values
    int64 value = 0;
    while (! input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
    }
    // Emit sum for input->key()
    Emit(IntToString(value));
  }
};

REGISTER_REDUCER(Adder);
```

# Full “Word Count” Example: Main Program

```
#include "mapreduce/mapreduce.h"

int main(int argc, char** argv) {
    ParseCommandLineFlags(argc, argv);
    MapReduceSpecification spec;

    // Store list of input files into "spec"
    for (int i = 1; i < argc; i++) {
        MapReduceInput* input = spec.add_input();
        input->set_format("text");
        input->set_filepattern(argv[i]);
        input->set_mapper_class("WordCounter");
    }
    // Specify the output files:
    // /gfs/test/freq-00000-of-00100
    // /gfs/test/freq-00001-of-00100
    // ...
    MapReduceOutput* out = spec.output();
    out->set_filebase("/gfs/test/freq");
    out->set_num_tasks(100);
    out->set_format("text");
    out->set_reducer_class("Adder");

    // Optional: do partial sums within map
    // tasks to save network bandwidth
    out->set_combiner_class("Adder");

    // Tuning parameters: use at most 2000
    // machines and 100 MB memory per task
    spec.set_machines(2000);
    spec.set_map_megabytes(100);
    spec.set_reduce_megabytes(100);

    // Now run it
    MapReduceResult result;
    if (!MapReduce(spec, &result)) abort();

    // Done: 'result' structure contains info
    // about counters, time taken, number of
    // machines used, etc.

    return 0;
}
```