Cloud Programming

Programming Environment

Dec 12, 2013

Osamu Tatebe

Cloud Computing

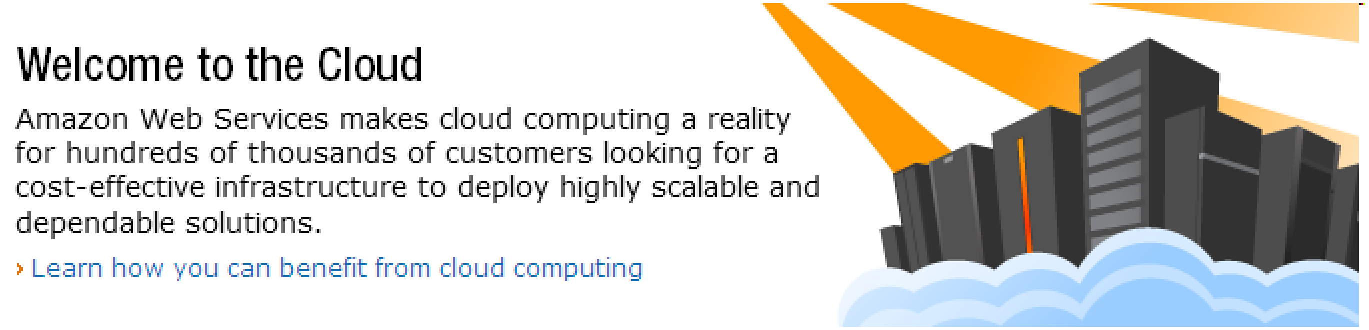
- Only required amount of CPU and storage can be used anytime from anywhere via network
  - Availability, throughput, reliability
  - Manageability
- No need to procure, maintain, and update computers
- Large-scale distributed data processing by MapReduce
  - Loosely coupled data intensive computing
  - Can be a standard parallel language other than MPI

Salesforce.com (1999)

- Provides Customer Relationship Management (CRM) service via network
  - No need to install software and hardware
  - Web interface
    - Outlook, Office, Notes, mobile, offline
  - Customizable
    - By mouse click, or Apex code
  - Multitenant

Amazon Web Services (2002)

- On-demand elastic infrastructure managed by web services
  - Elastic Compute Cloud (EC2)
    - Web service that provides resizable compute capacity
  - Simple Storage Service (S3)
    - Simple web service I/F to store and retrieve data
  - Elastic Block Store (EBS)
    - Block level storage used by EC2 in the same AZ
    - Automatically replicate within the same AZ
    - Point-in-time snapshots can be persisted to S3
- Region and Availability Zone



Amazon CloudFront (2008)

- Web Service for Content Delivery
  - Low latency, high data transfer, no commitments
- Cache copies close to end users
  - US, Europe, Japan, Hong Kong
- No need to maintain web servers
- By default, support peak speeds of 1 Gbps, and peak rates of 1,000 req/sec
- Designed for delivery of “popular” objects
  - Cache poplar objects and remove less poplar objects

Introducing Amazon CloudFront

Distribute your popular content from Amazon S3 around the globe with a single API call. High-performance content delivery is now self-service and pay-as-you-go.

› Learn more

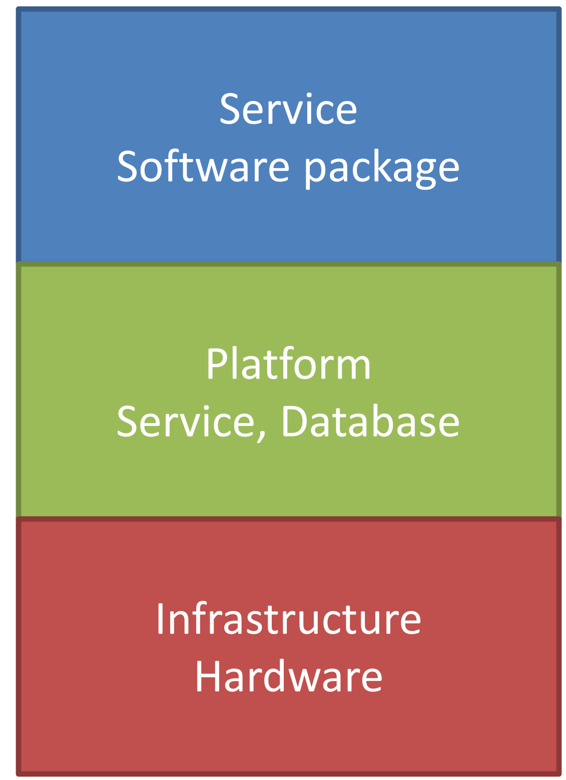
Google App Engine (2008)

- Google provides infrastructure to execute Web apps
  - Python SDK
- Datastore - Distributed data storage service
  - Data objects have a set of properties
  - Objects are retrieved by properties

- Not for large scale data processing

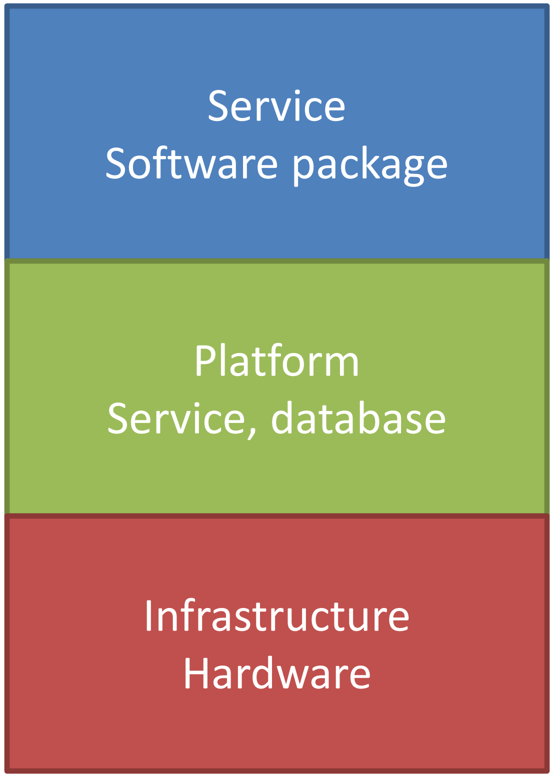
Taxonomy of Cloud

- SaaS (Software as a Service)
  - Google Apps (Gmail, …), CRM
  - Microsoft Online Services
- PaaS (Platform as a Service)
  - Development of Web apps
    - Force.com, Google App Engine
    - Windows Azure
- IaaS (Infrastructure as a Service)
  - Amazon EC2, S3

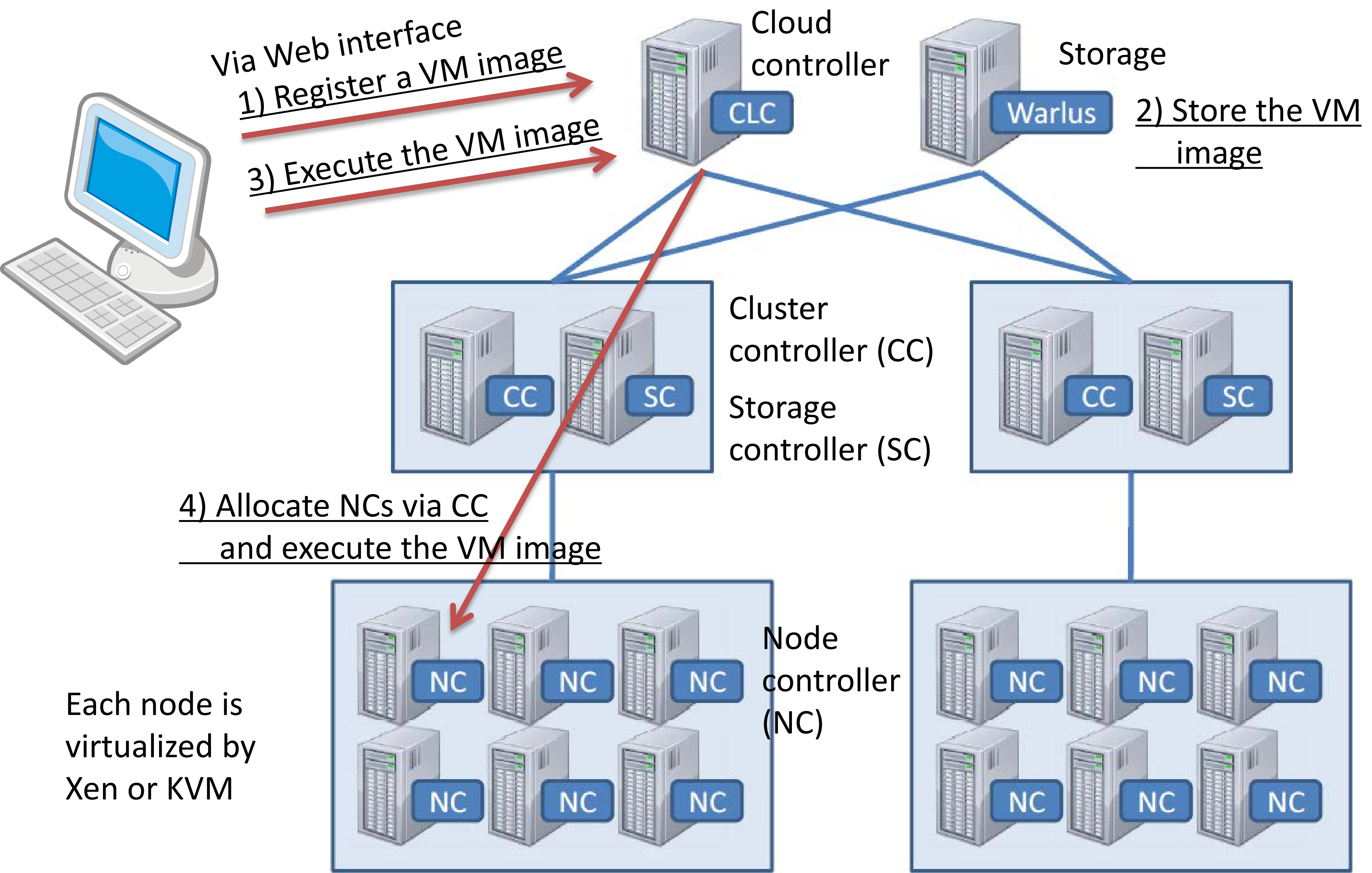


Cloud technology

- SaaS (Software as a Service)
  - Web 2.0
- PaaS (Platform as a Service)
  - Web API
  - Web Service
    - XML, WSDL, SOAP/REST
- IaaS (Infrastructure as a Service)
  - Virtual machine (Xen, KVM)
  - Virtualization of harddisk, storage and network



Example of IaaS: Eucalyptus [2009 Nurmi]



Eucalyptus (2)

- Node controller virtualizes compute node on which VM image is executed (equivalent of EC2)
- Storage Controller virtualizes block device (EBS)
- Warlus virtualizes storage (S3)
- Cloud controller manages the cloud system via Web interface
  - Registers a VM image
  - Allocates a block device
  - Allocates a compute node, execute the VM image, and mount the block device
  - Accesses to storage

Storage system in cloud

- Availability, reliability
- Amazon Web Services
  - S3, EBS
  - Can construct any (file) system that uses block device
    - HDFS (using EBS) for Elastic MapReduce
  - Difficult to construct a system beyond Availability Zone and Region
- Google App Engine
  - Utilize GFS and BigTable
  - Cannot use MapReduce
  - Cannot be geometrically distributed

Summary of cloud computing

- Resources in cloud computing
  - Inexpensive, always available, reliable, high performance
  - Easy to maintain
- Realized by virtualization and web interface
- No need to procure, maintain, and update computers
- If required, more resources can be obtained by cloud

MapReduce (2004)

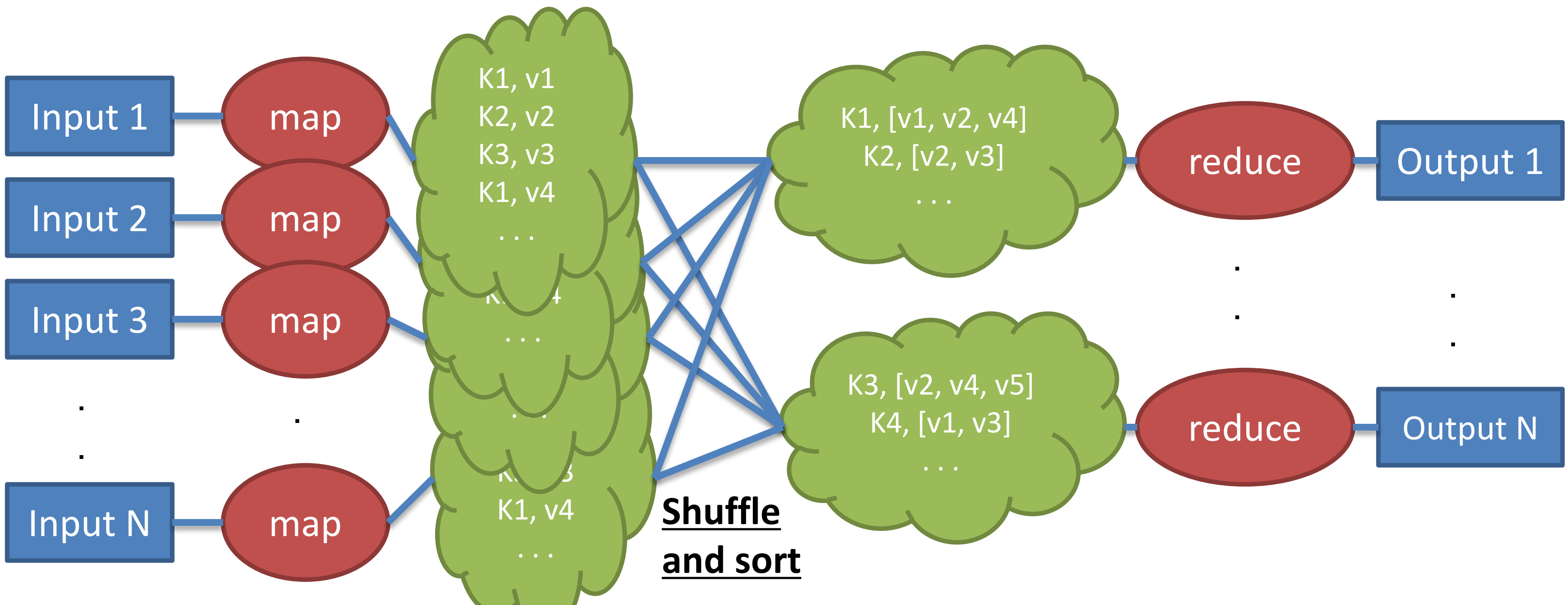
- **Programming model and runtime for data processing on large-scale cluster**
- A user specifies map and reduce functions
- Runtime system does
  - Automatically parallelize
  - Manage machine failure
  - Schedule jobs to efficiently exploit disk and network

Background

- Google requires to process
  - Inverted index
  - Various graph expression of Web documents
  - Number of pages that each host crawls
  - Set of the hottest query in a day
    - from large amount of crawled documents and Web request logs using hundreds to thousands of compute nodes
- Large amount of codes for parallelization, data distribution, error handing are required
- These hide original code for computation

New abstraction (1)

- Describes only required computation
- Runtime library hides complicated processes including <u>parallelization, fault handling, data distribution, load balancing</u>
- Most computation has the following same pattern

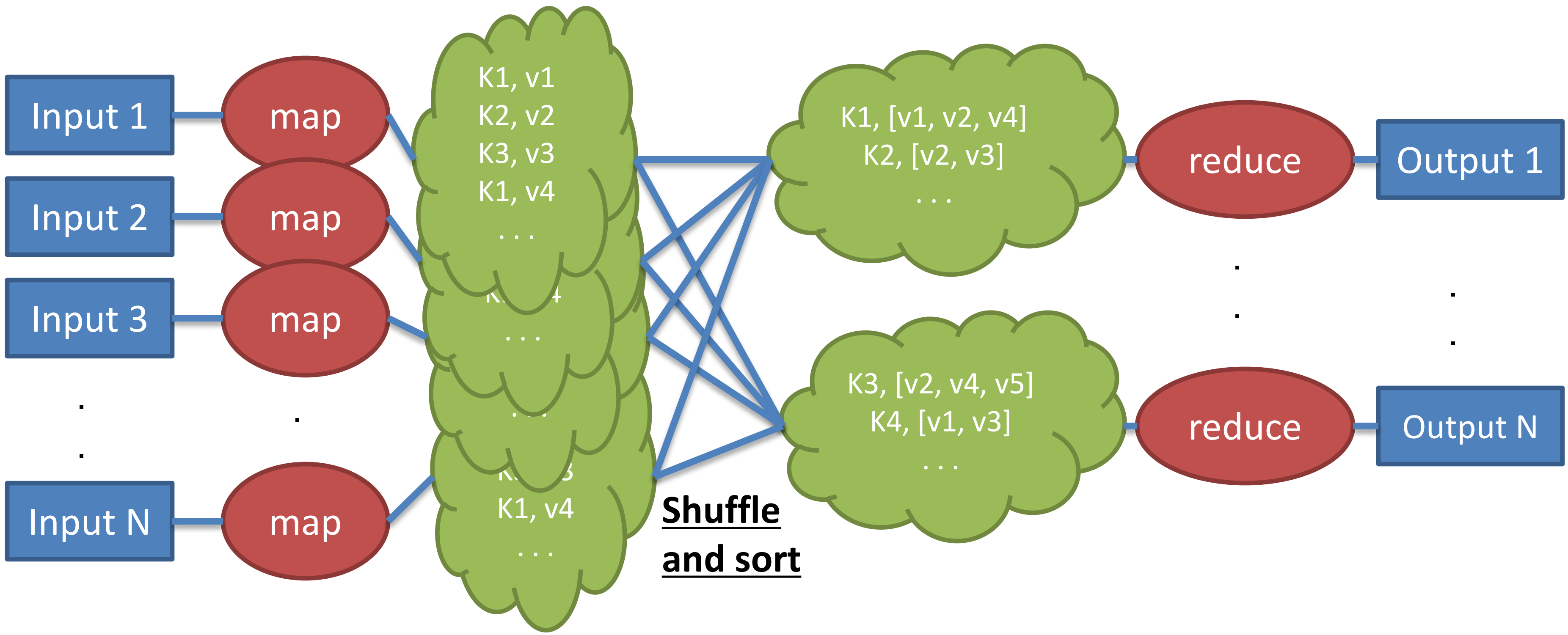


New abstraction (2)

- A functional model of user-supplied map and reduce operations enables
  - Easy parallelization of large-scale computation
  - To run failed tasks again for fault tolerance

- Simple but powerful interface
- It enables high-performance computation on large-scale cluster by auto-parallelization and auto-distribution

Programming model



- Input, output, intermediate data are set of key/value pair
- Map and reduce operations are specified by a user
- Output of map task is sorted by key, and transferred to reduce task

Example: word count

- Map task emits "a word" as a key and 1 as a value
  - (doc, "this is a pen")→(this, 1), (is, 1), (a, 1), (pen, 1)
- Reduce task sums a list of values [1 1 … 1] of each key
  - (this, [1 1 1 1]), (is, [1 1 1]), . . .→(this, 4), (is, 3), . . .

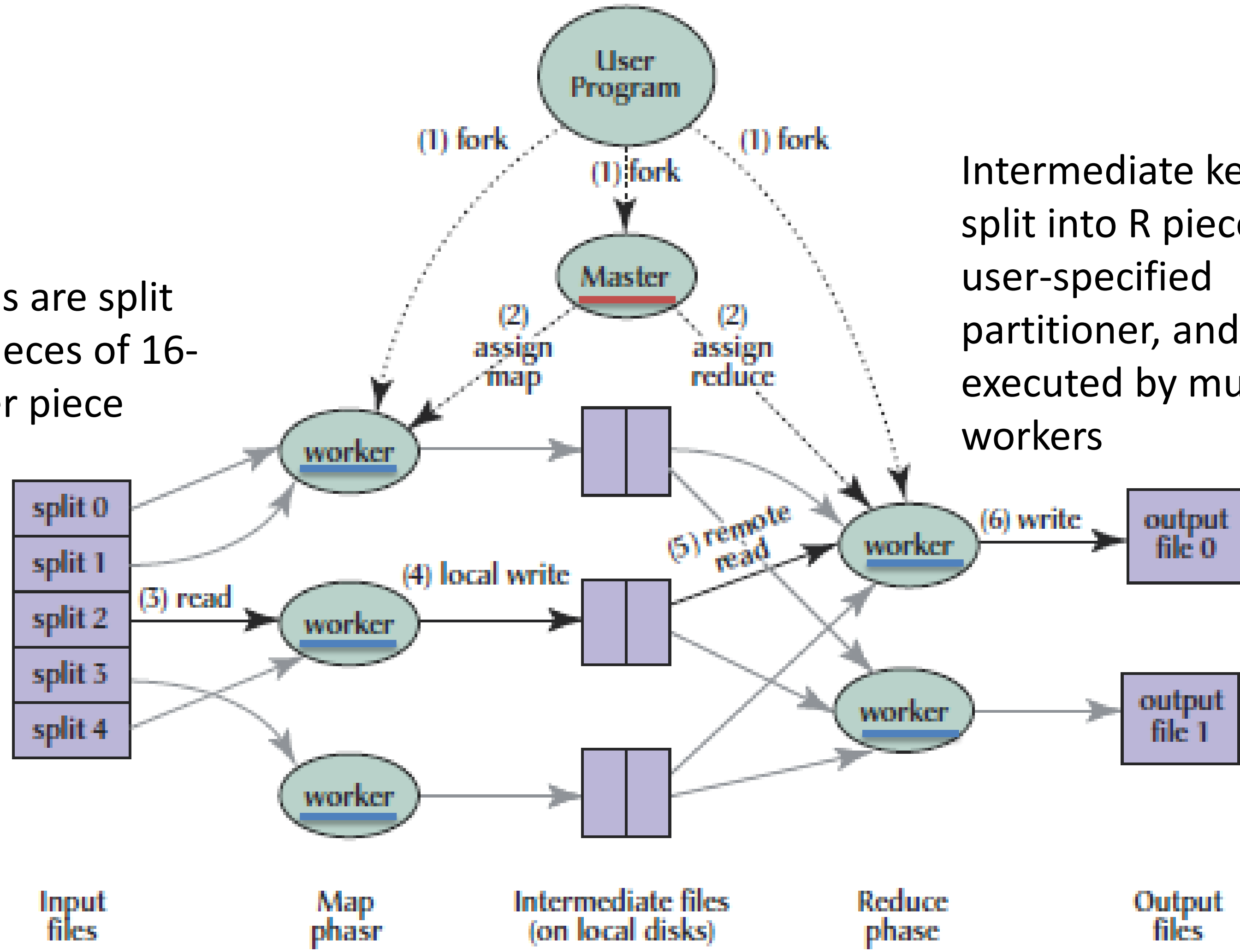
Pseudocode for word count

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

reduce(String key, Iterator values):
// key: a word
// values: a list of counts
int result = 0;
for each v in values:                // sum all counts for each word
    result += ParseInt(v);
Emit(AsString(result));
```

Execution overview

Input files are split into M pieces of 16-64MB per piece



Intermediate keys are split into R pieces by user-specified partitioner, and executed by multiple workers

Fault tolerance

- Indispensable when using hundreds to thousands of nodes
- Handling worker failures
  - The master pings every workers periodically
    - If no response is received from a worker in a certain amount of time, the master marks the worker as failed
  - Any map tasks completed by the worker, any map task or reduce task in progress on a failed worker are re-scheduled
    - Output of map task is stored to a local disk. If the node fails, the output cannot be read.
    - Output of reduce task is stored to a shared file system, which can be read after the worker failure
- Handling master failure
  - It is possible by checkpoint/restart mechanism, however, the master failure is not often since it is a single master

Locality

- Network bandwidth is a relatively scarce resource in PC cluster
- Input data is stored in Google file system (GFS)
  - The file data is stored on the local disks of the worker nodes
  - Each file is divided into 64MB blocks. 3 copies of each block are stored on different machines
- Master takes the location information of the input files into account and attempts to schedule a map task
  - on a machine that contains a replica of the corresponding input data
  - Or, on a machine that is on the same network switch
- <u>Most input data is read locally and consumes no network bandwidth</u>

Task Granularity

- Let be *M* map tasks and *R* reduce tasks
- *M*, *R* >> #workers is ideal
  - Improves dynamic load balancing
  - Speeds up recovery when a worker fails
- Practical bounds of *M* and *R*
  - Implementation issue: master must make O(*M*+*R*) scheduling decisions and keep O(*M***R*) state in memory
  - In practice, *M* is chosen so that each individual task is 16MB to 64MB of input data
  - *R* is a small multiple of # worker machines
    - Typical example, *M* = 200,000 and *R* = 5,000 using 2,000 worker machines

Backup tasks

- A straggler, a machine that takes an unusually long time to complete, causes that the total time lengthens
  - A bad disk may slow its read performance from 30MB/s to 1MB/s
  - Other tasks may be scheduled on the machine, which causes competition for CPU, memory, local disk or network bandwidth
- Master schedules backup executions of the remaining *in-progress tasks* when the MapReduce operation is close to completion
  - The task completes whenever either execution completes
- This mechanism can be tuned so that it increases the used computational resources by no more than a few percent
- Sort example: 44% longer to complete when this is disabled

Refinements

- User-specified partitioning function for determining the mapping of intermediate key values to the *R* reduce tasks
- Ordering guarantees of intermediate key/value pairs
- User-specified combiner functions
  - For doing partial combination of generated intermediate values with the same key within the same map task
  - To reduce the amount of intermediate data that must be transferred across the network
- Custom input and output types
- A mode for execution on a single machine for simplifying debugging and small-scale testing
- http server function to monitor the execution

Environment of performance evaluation

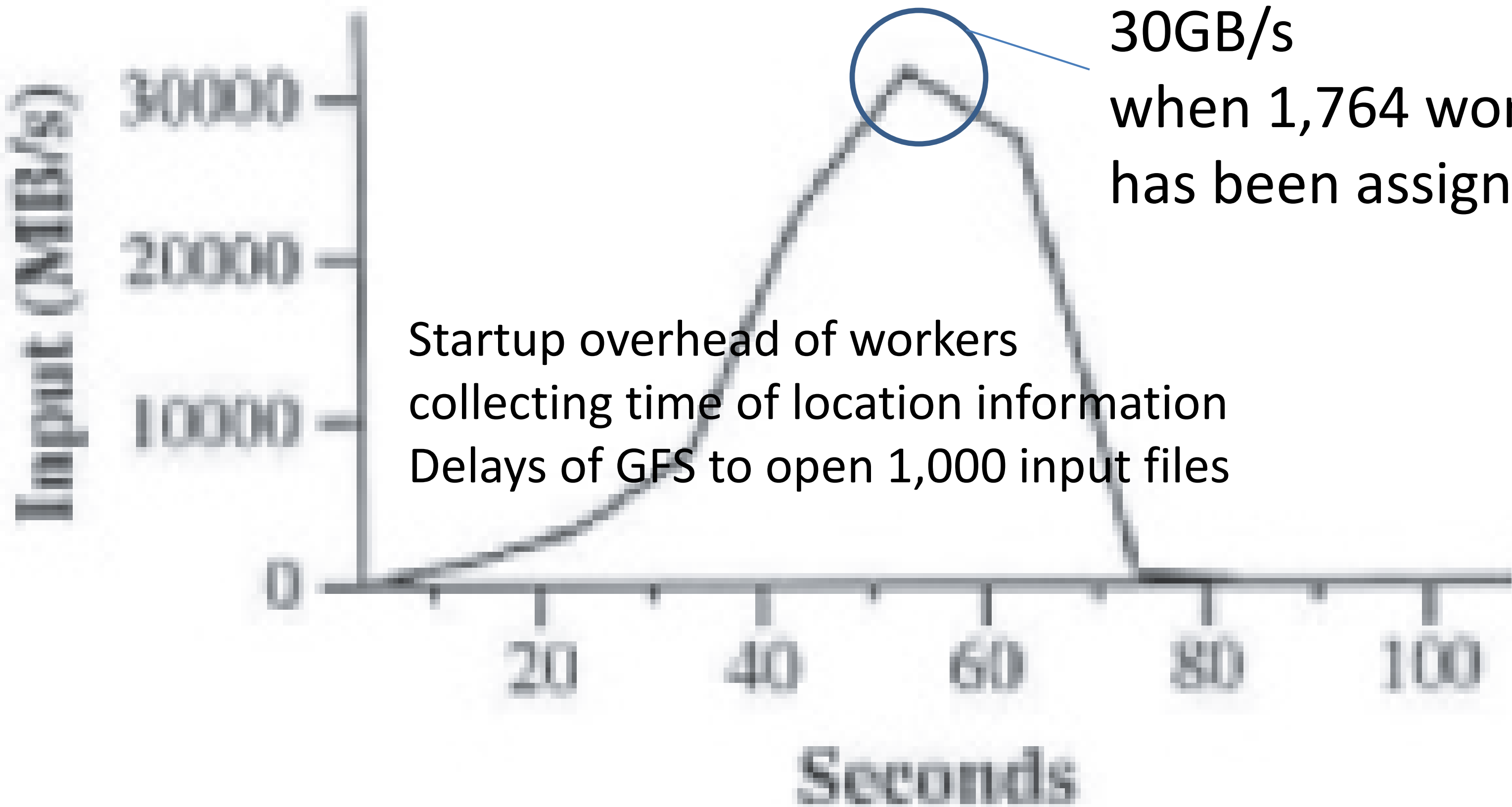
- 1,800 nodes of cluster
  - Two 2GHz Xeon with Hyper-Threading enabled
  - 4GB of memory
  - Two 160GB IDE disks
  - Gigabit Ethernet
- Network configuration
  - Two-level tree-shaped switched network
  - 100-200Gbps of aggregate bandwidth available at the root
- In the same hosting facility, RTT is less than a millisecond

Grep

- 10^{10} 100-byte records (～1TB)
- Searching for three-character pattern
  - The pattern occurs in 92,337 records
- M = 15,000 (input data is split into 64MB pieces), R = 1

Data transfer rate over time

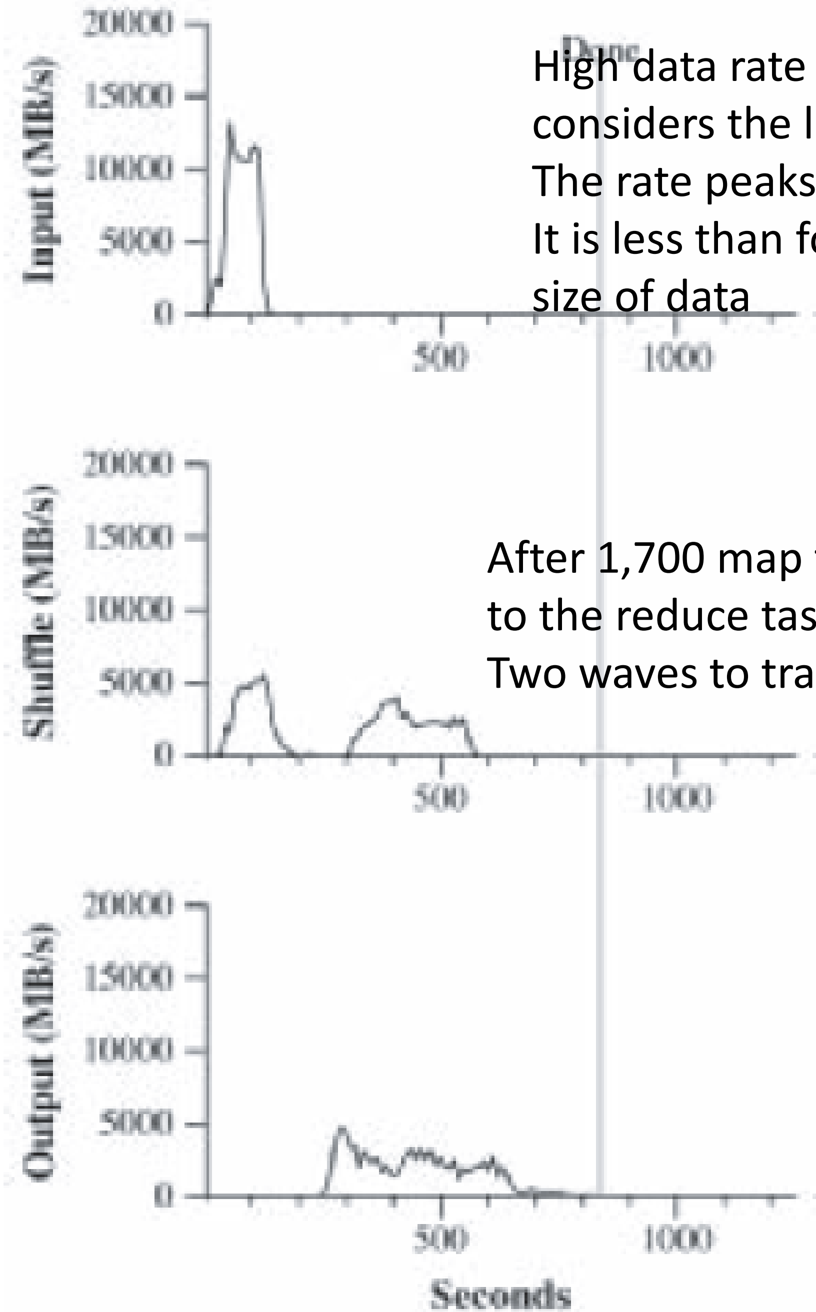
The rate peaks at over 30GB/s
when 1,764 workers has been assigned



Startup overhead of workers
collecting time of location information
Delays of GFS to open 1,000 input files

Sort

- Sorts 10^{10} 100-byte records (～1TB)
  - Cf. TeraSort benchmark http://sortbenchmark.org/
- Less than 50 lines of user code
- The final output is written to a set of 2-way replicated GFS files
- M = 15,000, R = 4,000
- Partitioning function uses the initial bytes of the key (12bit?)
  - In general, knowledge of the distribution of keys is required
  - Which can be obtained by prepassing MapReduce operation to obtain a sample of the keys



High data rate by scheduling of map tasks that considers the locality
The rate peaks at about 13GB/s
It is less than for grep since the sort needs to output the same size of data

After 1,700 map tasks complete, the intermediate data is transferred to the reduce tasks
Two waves to transfer data

Due to storing two copies, the rate is less than for shuffle

Example of larges-scale indexing

- All indexing processes are written in MapReduce in Google
  - The indexing code is simpler and smaller. 3,800 lines in C++ to 700 lines
  - Easy to change the indexing process
  - The operator intervention is not needed by fault tolerance of MapReduce
  - Easy to improve the performance by adding new machines to the cluster

Summary of MapReduce

- MapReduce programming model has been successfully used at Google for many different purposes
  - Easy to use
  - It hides details of parallelization, fault tolerance, locality optimization and load balancing
  - A large variety of problems are easily expressible
  - Scales to large clusters of machines comprising thousands of machines
- It can be obtained by restricting the programing model