MapReduce: Simplified Data Processing on Large Clusters

Nikhil Panpalia

Outline

- What is MapReduce?
  - What are Map and Reduce?
- Scalability
- Implementing MapReduce
  - opportunities for parallelism
  - input, output, execution
  - optimizations and extensions
- Fault Tolerance
- Performance
- MapReduce on multicore platforms
- MapReduce on mobile platforms
- Does it work for any computation?

What is MapReduce?

- A framework for processing large-scale data sets using a cluster of machines.

- Who should use MapReduce?

  A programmer with:
  - Lots of data to store and analyze
  - Lots of machines available for processing the data
  - Doesn’t have the time to become a distributed systems expert who can build an infrastructure to handle this task

What is MapReduce?

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 values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown

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 computation with large amounts of complex code to deal with

A simple problem

- Search for a pattern “cs395t” in a collection of files
- You would typically run a command like this:
  grep -r “cs395t” <directory>
- Now, suppose you have to do this search over terabytes of data and you have a cluster of machines at your disposal.
  How can you make this grep faster?
  Build a distributed grep!

Do we really need a distributed solution?

- Why can't I just use my desktop to do the processing?
  - How long does it take to read 1 TB of data?

    Considering an average read speed of 90MB/s[1]: ~3.23 hours

    If you use an SSD with read speed of 350MB/s[2]: ~50 minutes

- How much time it will take for searching through a terabyte of data? Or maybe sorting it?

- MapReduce can sort 1000 TB of data in 33 minutes![3]

[1] Numbers are for Western Digital 1TB SATA/300 drive.

[2] Numbers are for Crucial 128 GB m4 2.5-Inch Solid State Drive SATA 6Gb/s

[3] using 8000 machines - http://googleresearch.blogspot.com/2011/09/sorting-petabytes-with-mapreduce-next.html

Should I build my own distributed system/framework?

- It's hard!
  - Machine and network management
  - Task management
  - Fault tolerance
  - Availability despite failures
  - Scalability

Understanding Map and Reduce

```
var a = [1,2,3];

for (i=0; i<a.length; i++)
      a[i] = a[i] * 2;

for (i=0; i<a.length; i++)
      a[i] = a[i] + 2;
```

Understanding Map and Reduce

I can change it to:

```
function map(fn, a) {
    for (i = 0; i < a.length; i++)
        a[i] = fn(a[i]);
}

map(function(x){return x*2;}, a);
map(function(x){return x+2;}, a);
```

Understanding Map and Reduce

```
function sum(a) {
        var s = 0;
        for (i = 0; i < a.length; i++)
            s += a[i];
        return s;
}

function join(a) {
        var s = "";
        for (i = 0; i < a.length; i++)
            s += a[i];
        return s;
}

alert(sum([1,2,3]));
alert(join(["a","b","c"]));
```

Understanding Map and Reduce

```
function reduce(fn, a, init) {
        var s = init;
        for (i = 0; i < a.length; i++)
            s = fn( s, a[i] );
        return s;
}

function sum(a) {
       return reduce(function(a, b){return a+b;}, a, 0);
}

function join(a) {
       return reduce(function(a, b){return a+b;}, a, “” );
}

alert(sum([1,2,3]));
alert(join(["a","b","c"]));
```

Understanding Map and Reduce

- Passing functions as arguments – functional programming
- map – does something to every element in an array – can be done in any order!
  - amenable to parallelization
- So, if you have 2 CPUs, map will run twice as fast
- map is an example of embarrassingly parallel computation

Understanding Map and Reduce

- Suppose you have a huge array with elements which are all the webpages from the Internet

- To search the entire Internet:
  - you just need to pass a string_searcher function to map
  - reduce will be an identity function
  - run a MapReduce job on a cluster
  - …that’s it! you are searching the Internet by writing just a few lines of code!

Map and Reduce

- map – function that takes key/value pairs as input and generates an intermediate set of key/value pairs

- reduce – function that merges all the intermediate values associated with the same intermediate key

Map and Reduce

- User needs to define these 2 functions
- Inspired by functional primitives in Lisp
- Functional model – data is immutable, functions don’t have side-effects
  - Allows automatic parallelization and distribution of large-scale computations easily

MapReduce

map: `(k1, v1)` → `list(k2, v2)`

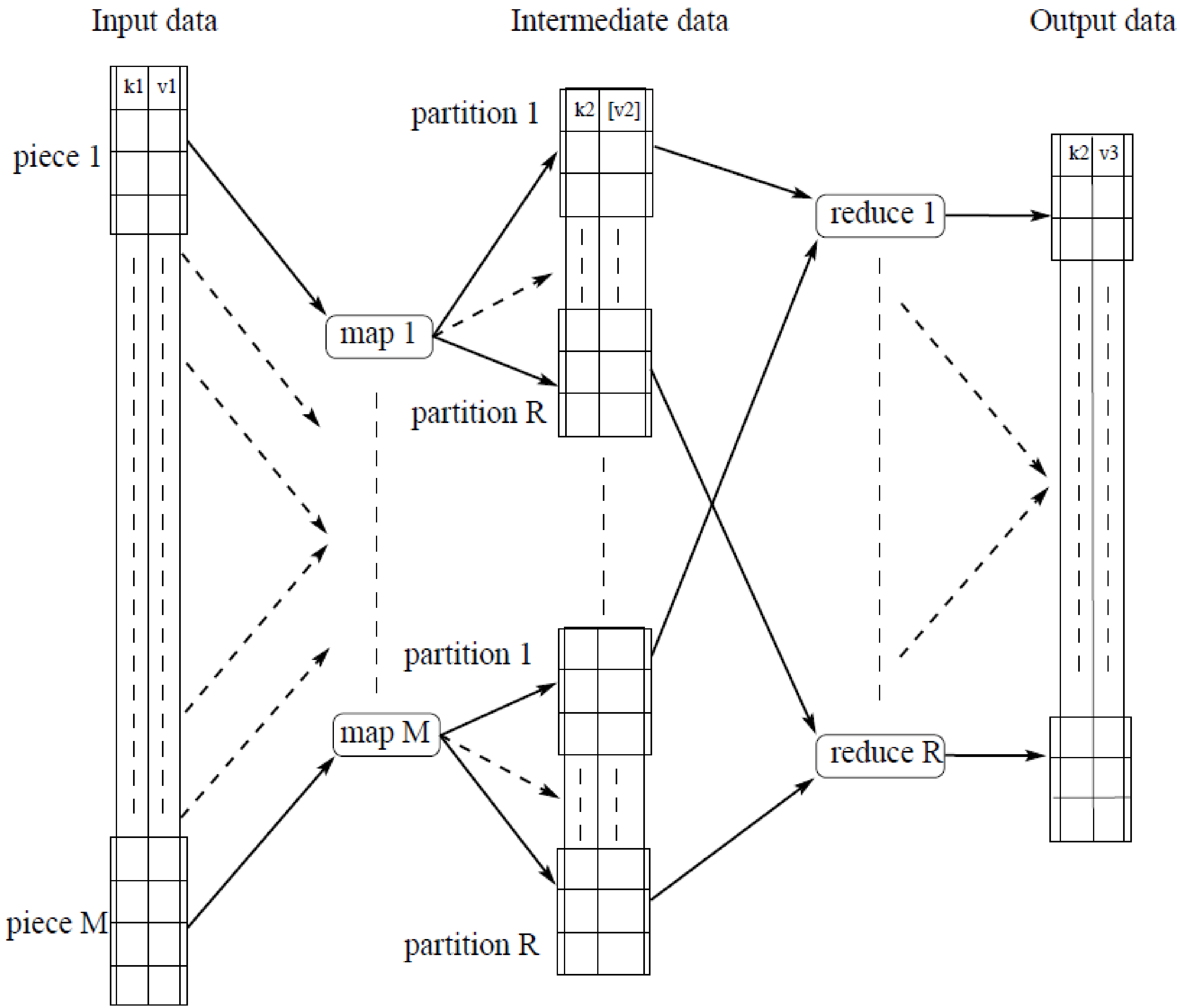
reduce: `(k2, list(k2, v2))` → `list(v2)`

map → shuffle → reduce

map
(input key/value pair
and produces intermediate
key/value pairs)

shuffle
(groups all values associated
with the same intermediate key)

reduce
(takes an intermediate key
and associated intermediate
values and merges them to
form a possibly smaller set
of values)

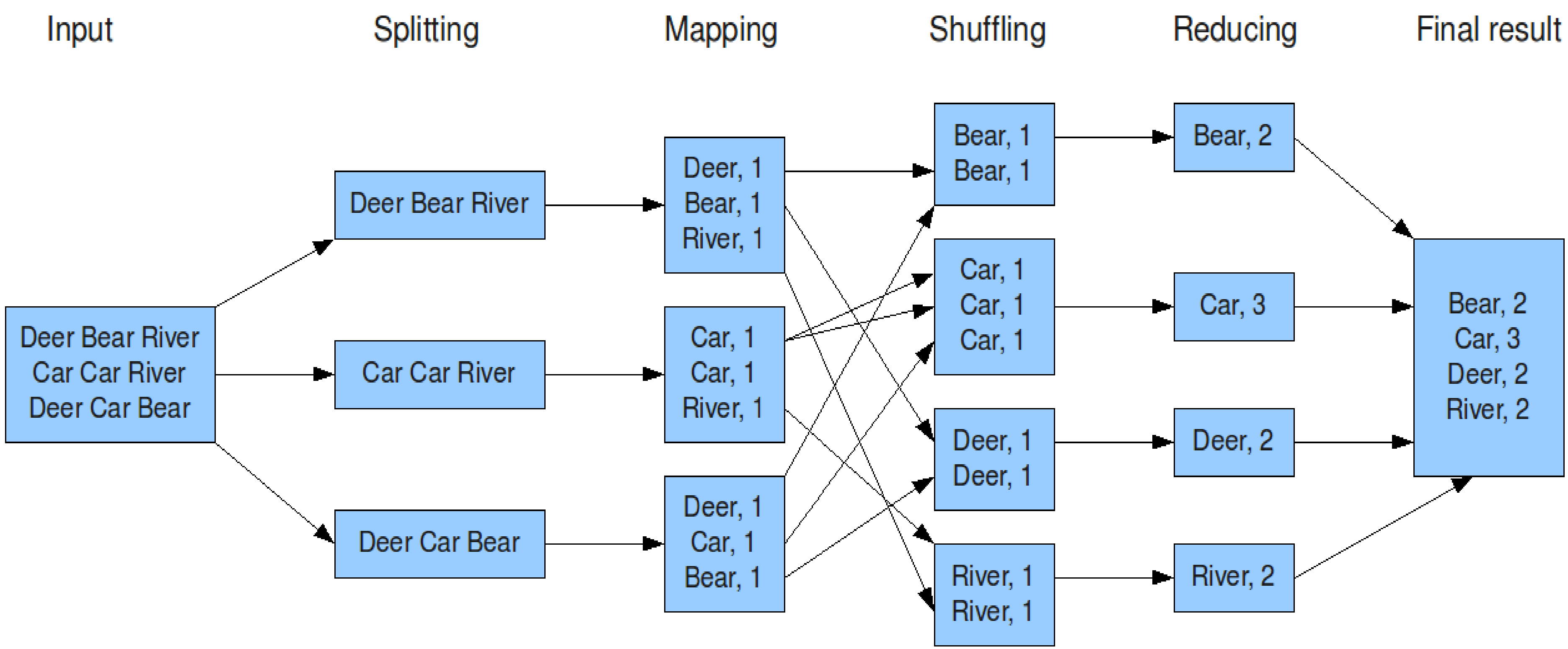

Input data
Intermediate data
Output data
k1
v1
piece 1
piece M
map 1
map M
partition 1
k2
[v2]
partition R
partition 1
partition R
reduce 1
reduce R
k2
v3

Example – Word Count

- Problem: counting occurrences of words in a large collection of documents

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

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



Word Counting using MapReduce

Example – Word Count

- Other than map and reduce, user needs to provide:
  - names of input and output files
  - optional tuning parameters (size of split, M, R, etc.)
- User's code is linked with MapReduce library and the binary is submitted to a task runner

Other Examples

- Distributed grep
  - map emits a line if it matches the given pattern
  - reduce just copies input to output

- Counting URL access frequency
  - map processes web server logs and outputs <URL, 1>
  - reduce sums all numbers for a single URL

Other Examples

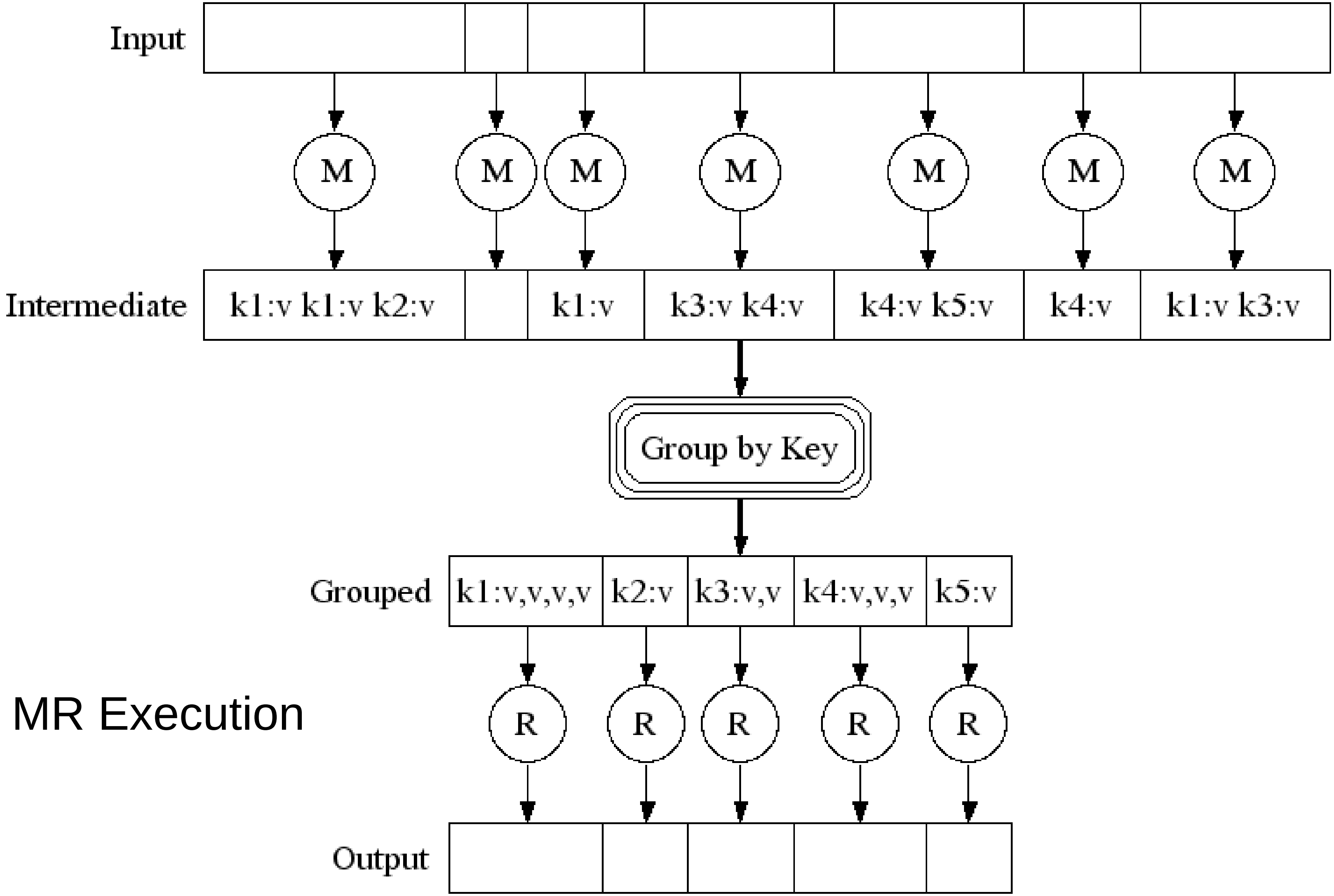
- Inverted index
  - map function parses document and emits <word, docID>
  - reduce gets all pairs for a given word and emits <word, list(docID)>
- Distributed sort
  - map extracts key for a record and emits <key, record>
  - reduce emits all pairs unchanged

Implementing Map and Reduce

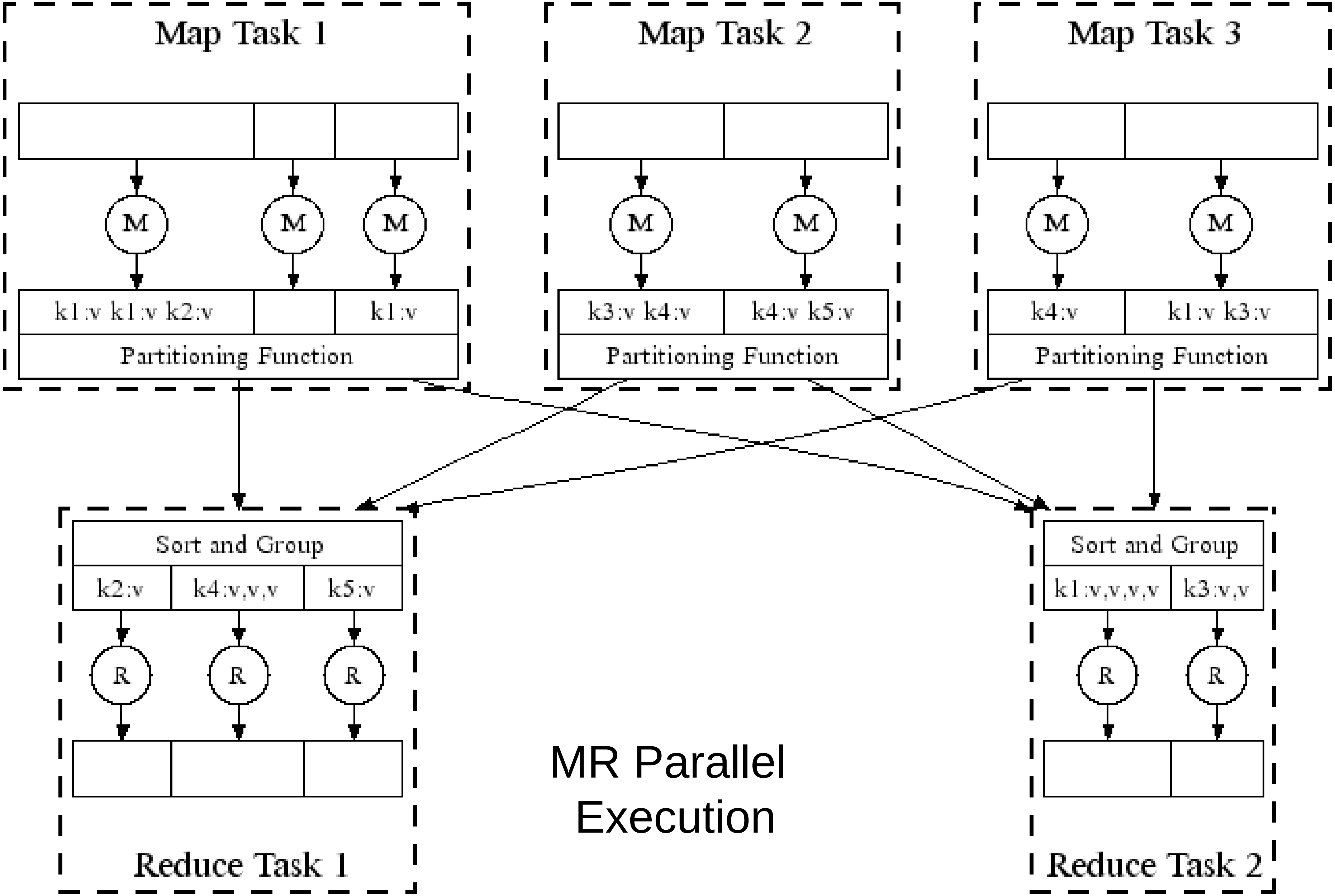
- Now, all we need is some “genius” to implement these 2 abstractions – map & reduce
  - Exploit parallelism in the computation
  - Massively scalable – can run on hundreds or thousands of machines
  - Hide the details of cluster management tasks like scheduling of tasks, partitioning of data, network communication from the user
  - Fault tolerant (in large clusters failures are a norm rather than being an exception)

Implementing MR: Opportunities for Parallelism

- Input – all key/value pairs can be read and processed in parallel by map
- Intermediate grouping of data – essentially a sorting problem; can be done in parallel and results can be merged
- Output – All reducers can work in parallel
  - each individual reduction can be parallelized



MR Execution



MR Parallel Execution

Implementing MR: Exploit parallelism using a cluster

- Characteristics of the cluster:
  - Lot of commodity PCs connected together
  - Network is a scarce resource
  - Failures are very common
  - Storage is provided by a distributed file system using inexpensive disks
  - File system replication is used to provide reliability and availability
  - A scheduling system decides which jobs will run on which machines

Distributed File System

- Allows access to files from multiple hosts over the network
- Support concurrency (multiple clients reading/writing the same file)
- Support for replication

- GFS: distributed file system used in Google's MapReduce is important for achieving good performance (high availability and durability via replication)

Google File System (GFS)

- Motivation: redundant storage of massive amounts of data on cheap unreliable machines

- Assumptions:
  - modest number of very large files
  - files are write-once, never modified, mostly appended
  - fast streaming reads – high throughput desired
  - large number of component failures

Google File System (GFS) - Design

- Files stored as chunks (typically of 64MB)
  - helps in load balancing and better distribution of data across machines
  - can support files which cannot fit on 1 disk
- Each chunk is replicated multiple times (typically 3)
  - provides reliability and higher throughput for reads
- Single master (maintains all metadata) and multiple chunkservers (store actual data chunks)
- No caching of data (little benefit since data sets are large)
- Can (theoretically) scale to any number of chunkservers
- Writes at arbitrary positions in files supported but are not efficient (mostly append operations on files)

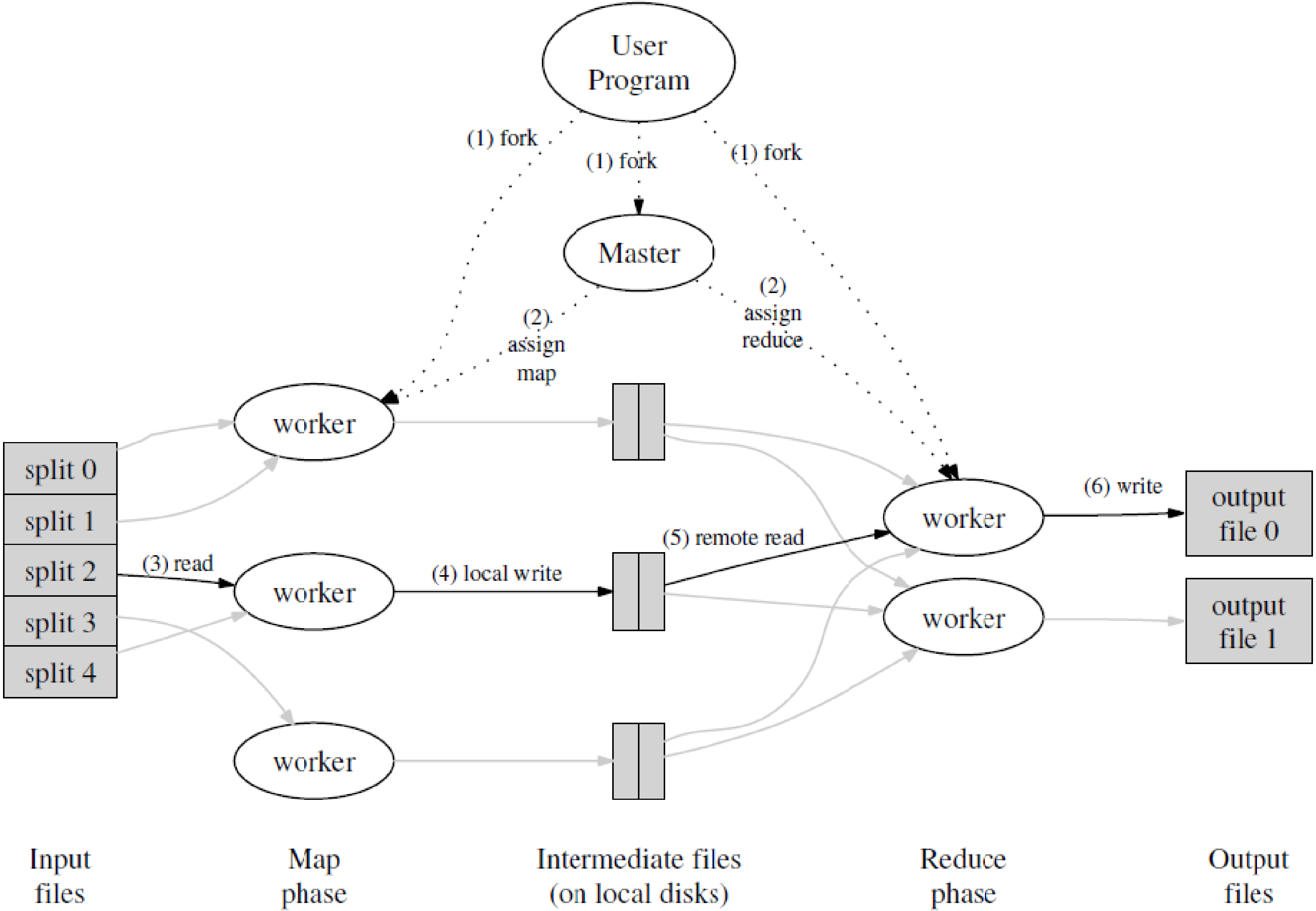
Implementing MR: Distributing the input

- Input data is partitioned into splits of size S and is processed by M mappers
  - splitting the data helps exploit the data parallelism in the input
  - number of map tasks is usually more than the number of available worker machines (better dynamic load balancing)
  - splits are of smaller size – typically the size of a filesystem block
    - better load balancing for storage
    - faster recovery:
      - less repetition of work in case of failures
      - repeated work can also be parallelized
  - M and S can be configured by the user

(Note: this step is optional if the files blocks are already distributed across machines by GFS.)

Implementing MR: Master

- Only 1 Master per MR computation
- Master:
  - assigns map and reduce tasks to the idle workers
  - informs the location of input data to mappers
  - stores the state (idle, in-progress, completed) and identity of each worker machine
  - for each completed map task, master stores the location and sizes of intermediate files produced by the mapper; this information is pushed to workers which have in-progress reduce tasks


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
worker
worker
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

MR: Step-by-Step Execution

- Split the input into M pieces and start copies of program on different machines
- One invocation acts as the master which assigns work to idle machines
- Map task:
  - read the input and parse the key/value pairs
  - pass each pair to user-defined Map function
  - write intermediate key-value pairs to disk in R files partitioned by the partitioning function
  - pass location of intermediate files back to master

MR: Step-by-Step Execution

- Master notifies the reduce worker
- Reduction is distributed over R tasks which cover different parts of the intermediate key's domain
- Reduce task:
  - read the intermediate key/value pairs
  - sort the data by intermediate key (external sort can be used)

    (note: many different keys can map to the same reduce task)
  - iterate over sorted data and for each unique key, pass the key and set of values to user-defined Reduce function
  - output of Reduce is appended to final output for the reduce partition
- MR completes when all map and reduce tasks have finished

MR: Output

- The output of MR is R output files (one per reduce task)
- The partitioning function for intermediate keys can be defined by the user
  - by default, it is "*hash(key) mod R" to generate well-balanced partitions*
- Result files can be combined or fed to another MR job

MR: Handling Faults

- With thousands of machines all made of cheap hardware, faults are very common

- MR library must tolerate any faults in the machines of the network gracefully *without significantly impacting the speed of the computation*

Fault Tolerance: Scenarios

- worker failure
- master failure
- network failure
- file system or disk failure – data corruption
- malformed records in input
- bugs in user code

Fault Tolerance: Worker Failures

- Master pings every worker periodically (alternatively, the worker can send a heartbeat message periodically)
- If worker does not respond, master marks it as failed
- Map worker:
  - any *completed* or *in-progress* tasks are reset to *idle* state
  - *completed* tasks need to be re-run since output is stored on a local file system
  - all reduce workers notified of this failure (to prevent duplication of data)
- Reduce worker:
  - any *in-progress* tasks are reset to *idle* state
  - no need to re-run *completed* tasks since output stored in global file system

Fault Tolerance: Master Failure

- Master periodically checkpoints its data structures
- On failure, new master can be elected using some leader election algorithm
- Theoretically, the new master can start off from this checkpoint
- Implementation: MR job is aborted if the master fails

Fault Tolerance: Network Failure

- Smart replication of input data by underlying file-system
- Workers unreachable due to network failures are marked as failed since its hard to distinguish this case from worker failure
- Network partitions can slow down the entire computation and may need a lot of work to be re-done

Fault Tolerance: Filesystem/Disk failure

- Depend on the filesystem replication for reliability

- Each data block is replicated f number of times (default: 3)
  - replication across machines on the same rack (machine failure)
  - replication across machines on different racks (rack failure)
  - replication across data-center (data-center failure)

Fault Tolerance: Malformed input

- Malformed input records could cause the map task to crash
- Usual course of action: fix the input
- But what if this happens at the end of a long-running computation?
- Acceptable to skip some records (sometimes)
  - word count over very large dataset
- MR library detects bad records which cause crashes deterministically
  - Signal handler catches error and communicates to the master
  - If more than 1 failure seen for the same record, master instructs the mapper to skip that record

Fault Tolerance: Bugs in user code

- Bugs in user provided Map and Reduce functions could cause crashes on particular records
- This case similar to the failure due to malformed input

Fault Tolerance: Semantics

- Map and Reduce must be deterministic functions of their input values
  - output produced by the distributed execution is same as the one produced by non-faulting sequential execution

- Atomic commit of output
  - on completion, map task sends names of R intermediate files to master (master ignores this if the map task was already completed elsewhere)
  - on completion, reduce task *atomically* renames its temporary file to final output file (on a global file system)

Locality Optimization

- Effective utilization of network
- Move computation near the input data
- Input data (managed by GFS) stored on local disks
  - several copies of each block
- Master considers this block location information when scheduling map task on a machine
- Most input data is read locally and consumes zero network bandwidth

Task Granularity

- M map tasks and R reduce tasks
- M and R much larger than the number of machines
  - Improves *dynamic* load balancing (add/remove machines)
  - Speeds up recovery
    - less work needs to be redone
    - work already completed by a failed task can be distributed across multiple idle workers
  - Bounds:
    - Master makes O(M+R) scheduling decisions
    - Master maintains O(M*R) state in memory
- M is chosen such that each task works on one block of data (maximize locality)
- R is usually constrained by users to reduce the number of output files

Stragglers and Backup tasks

- Straggler: machine that takes unusually long to complete one of the last few map/reduce tasks
  - reasons: bad disk, incorrect configuration, heavy load
  - significantly lengthens the total time of execution
- Solution: master schedules backup tasks for all *in-progress* tasks when MR is near completion
  - task marked complete when either primary or backup task finishes
  - tuned such that it does not increase the overall resource consumption by more than a few percent

Refinements/Extensions

- Partitioning function for intermediate keys
  - default: “hash(key) mod R”
  - user can provide custom function
    - eg: keys are URLs and we want all entries for a host in a single output file – “hash(Hostname(urlkey)) mod R”
- Ordering guarantees
  - within a partition, all intermediate key/values pairs are processed in increasing key order
  - generates a sorted output file per partition

Refinements/Extensions

- Combiner
  - same map task produces a lot of values for a single intermediate key
  - if Reduce is commutative and associative:
    - user can specify an optional combiner function
    - combiner runs on the same machine as the map task
    - combiner does partial reduction of the output of map before the data is send to the reducer
    - preserves network bandwidth and speeds up overall computation
  - Example – word count
    - every map task will produce hundreds of pairs of the form <"the", 1> which will be sent over the network
      - combiner can do partial reduction
      - only 1 pair is sent to the reducer from every map with key "the"

Refinements/Extensions

- Local Execution
  - all map/reduce tasks can be executed locally
  - helps with testing/debugging/profiling

- Counters
  - count occurrences of various events

```
Counter* uppercase;
uppercase = GetCounter("uppercase");
map(String name, String contents):
  for each word w in contents:
        if (IsCapitalized(w)):
              uppercase->Increment();
        EmitIntermediate(w, "1");
```

  - updated counters propagated to master periodically

Refinements/Extensions

- Support for arbitrary input types and sources
  - user needs to implement a reader interface

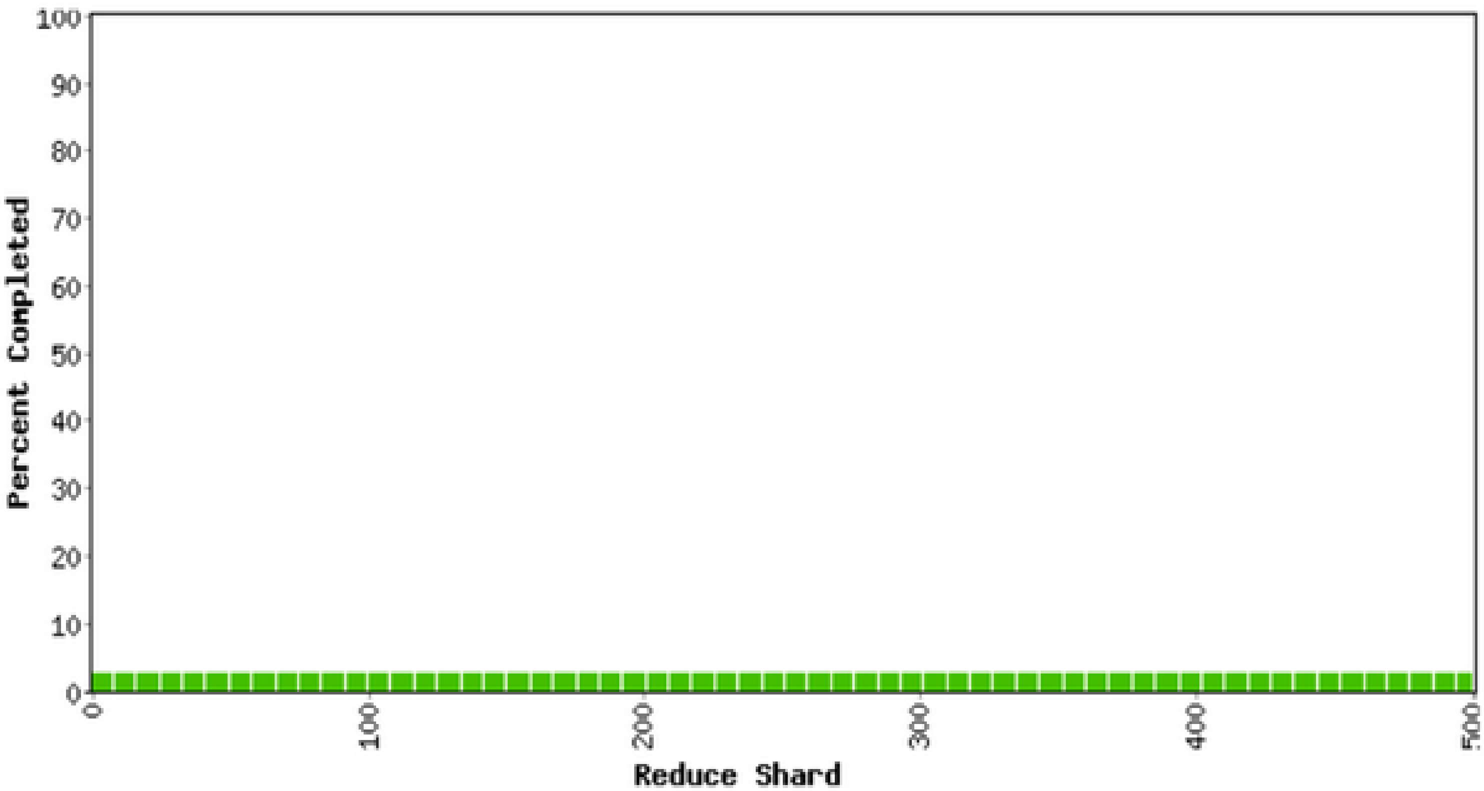
- Status Information
  - master runs an HTTP server and exports status pages
    - progress of computation
    - processing rate for input data
    - status of map/reduce tasks
    - failed workers
    - various counters – number of input key/value pairs, number of output records, etc.

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 00 min 18 sec

323 workers; 0 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	0	323	878934.6	1314.4	717.0
Shuffle	500	0	323	717.0	0.0	0.0
Reduce	500	0	0	0.0	0.0	0.0



Counters

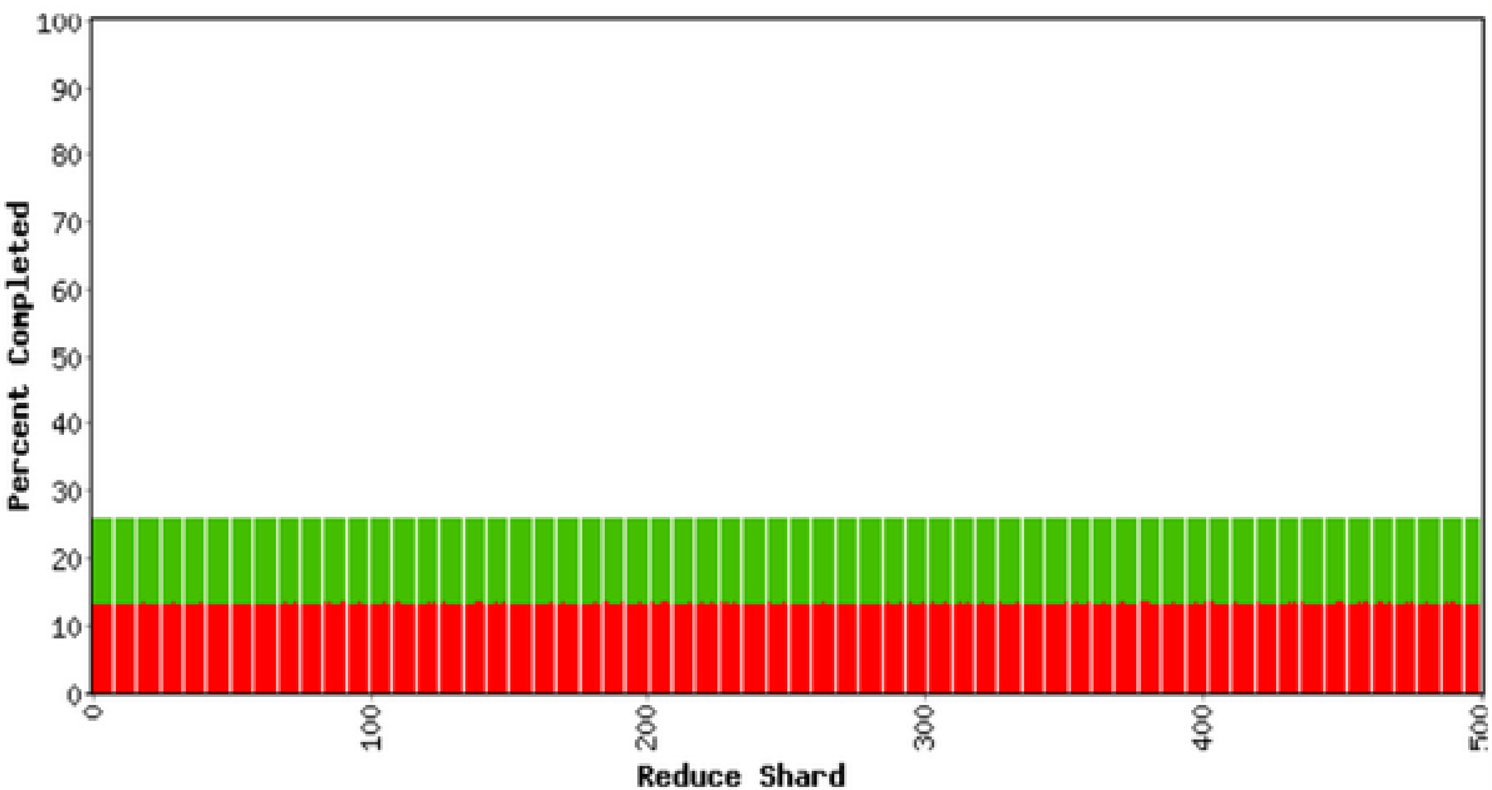
Variable	Minute
Mapped (MB/s)	72.5
Shuffle (MB/s)	0.0
Output (MB/s)	0.0
doc-index-hits	145825686
docs-indexed	506631
dups-in-index-merge	0
mr-operator-calls	508192
mr-operator-outputs	506631

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 05 min 07 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	1857	1707	878934.6	191995.8	113936.6
Shuffle	500	0	500	113936.6	57113.7	57113.7
Reduce	500	0	0	57113.7	0.0	0.0



Counters

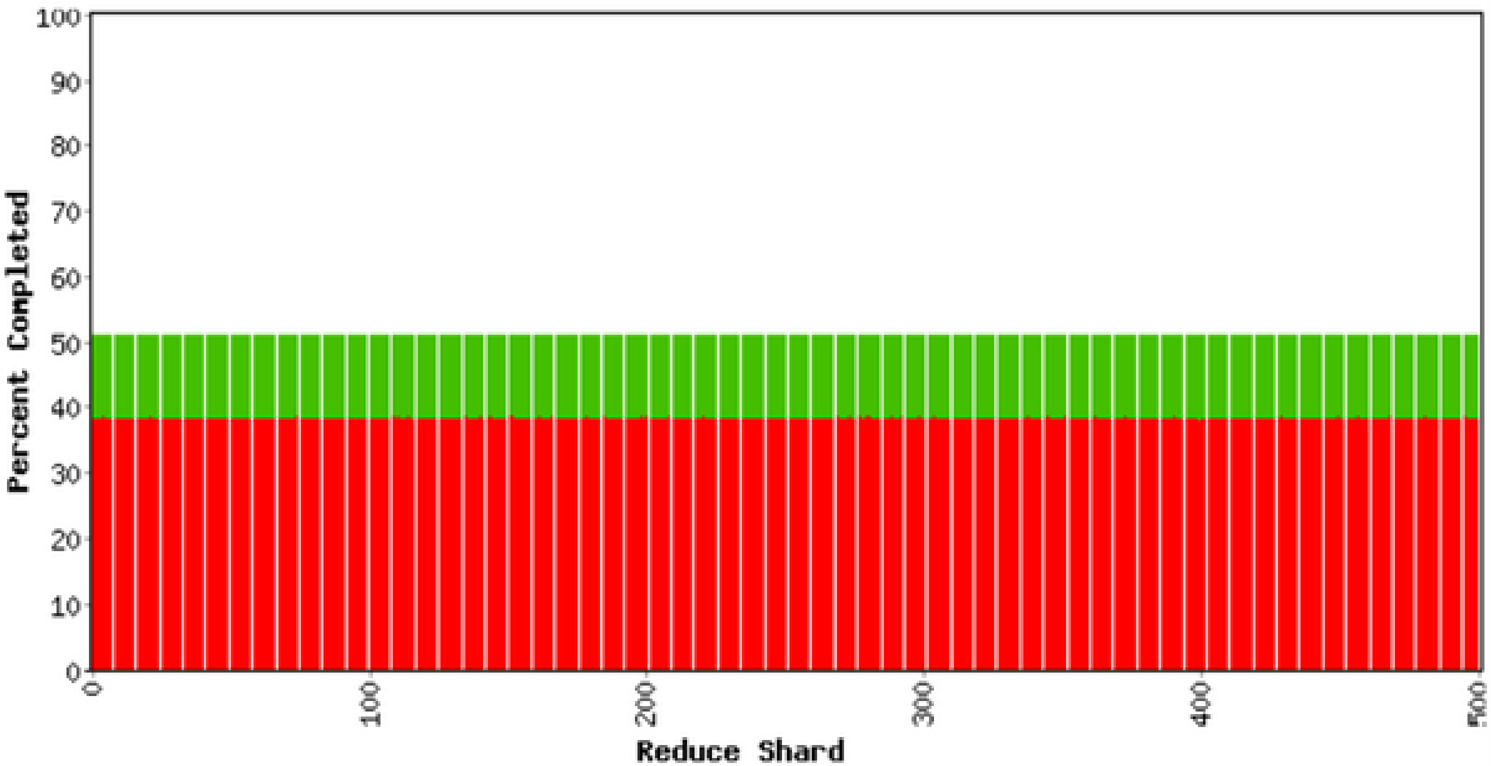
Variable	Minute
Mapped (MB/s)	699.1
Shuffle (MB/s)	349.5
Output (MB/s)	0.0
doc-index-hits	5004411944
docs-indexed	17290135
dups-in-index-merge	0
mr-operator-calls	17331371
mr-operator-outputs	17290135

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 10 min 18 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	5354	1707	878934.6	406020.1	241058.2
Shuffle	500	0	500	241058.2	196362.5	196362.5
Reduce	500	0	0	196362.5	0.0	0.0



Counters

Variable	Minute
Mapped (MB/s)	704.4
Shuffle (MB/s)	371.9
Output (MB/s)	0.0
doc-index-hits	5000364228
docs-indexed	17300709
dups-in-index-merge	0
mr-operator-calls	17342493
mr-operator-outputs	17300709

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 15 min 31 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	8841	1707	878934.6	621608.5	369459.8
Shuffle	500	0	500	369459.8	326986.8	326986.8
Reduce	500	0	0	326986.8	0.0	0.0

Counters

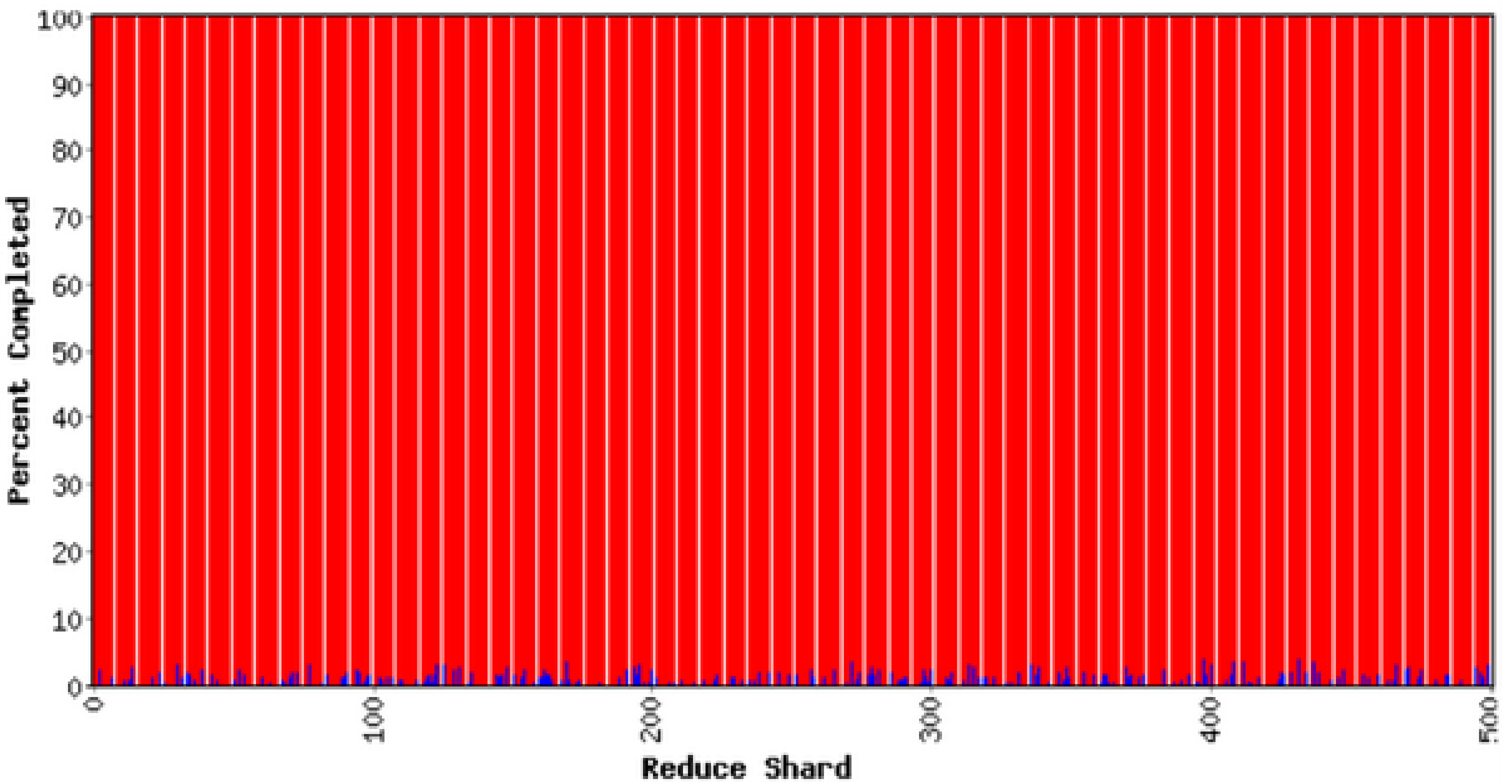
Variable	Minute
Mapped (MB/s)	706.5
Shuffle (MB/s)	419.2
Output (MB/s)	0.0
doc-index-hits	4982870667
docs-indexed	17229926
dups-in-index-merge	0
mr-operator-calls	17272056
mr-operator-outputs	17229926

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 29 min 45 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	195	305	523499.2	523389.6	523389.6
Reduce	500	0	195	523389.6	2685.2	2742.6



Counters

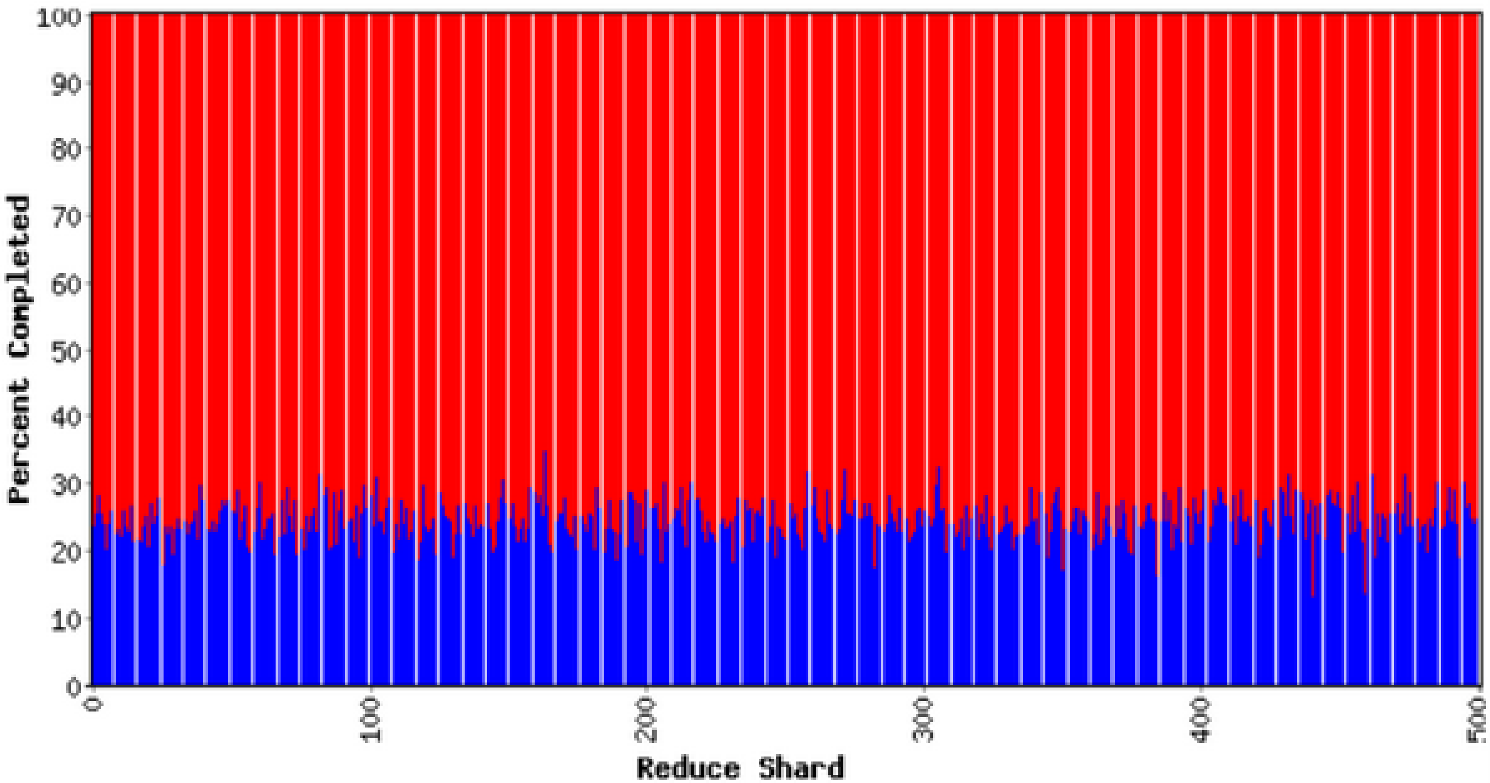
Variable	Minute	
Mapped (MB/s)	0.3	
Shuffle (MB/s)	0.5	
Output (MB/s)	45.7	
doc-index-hits	2313178	105
docs-indexed	7936	
dups-in-index-merge	0	
mr-merge-calls	1954105	
mr-merge-outputs	1954105	

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 31 min 34 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	500	0	523499.2	523499.5	523499.5
Reduce	500	0	500	523499.5	133837.8	136929.6



Counters

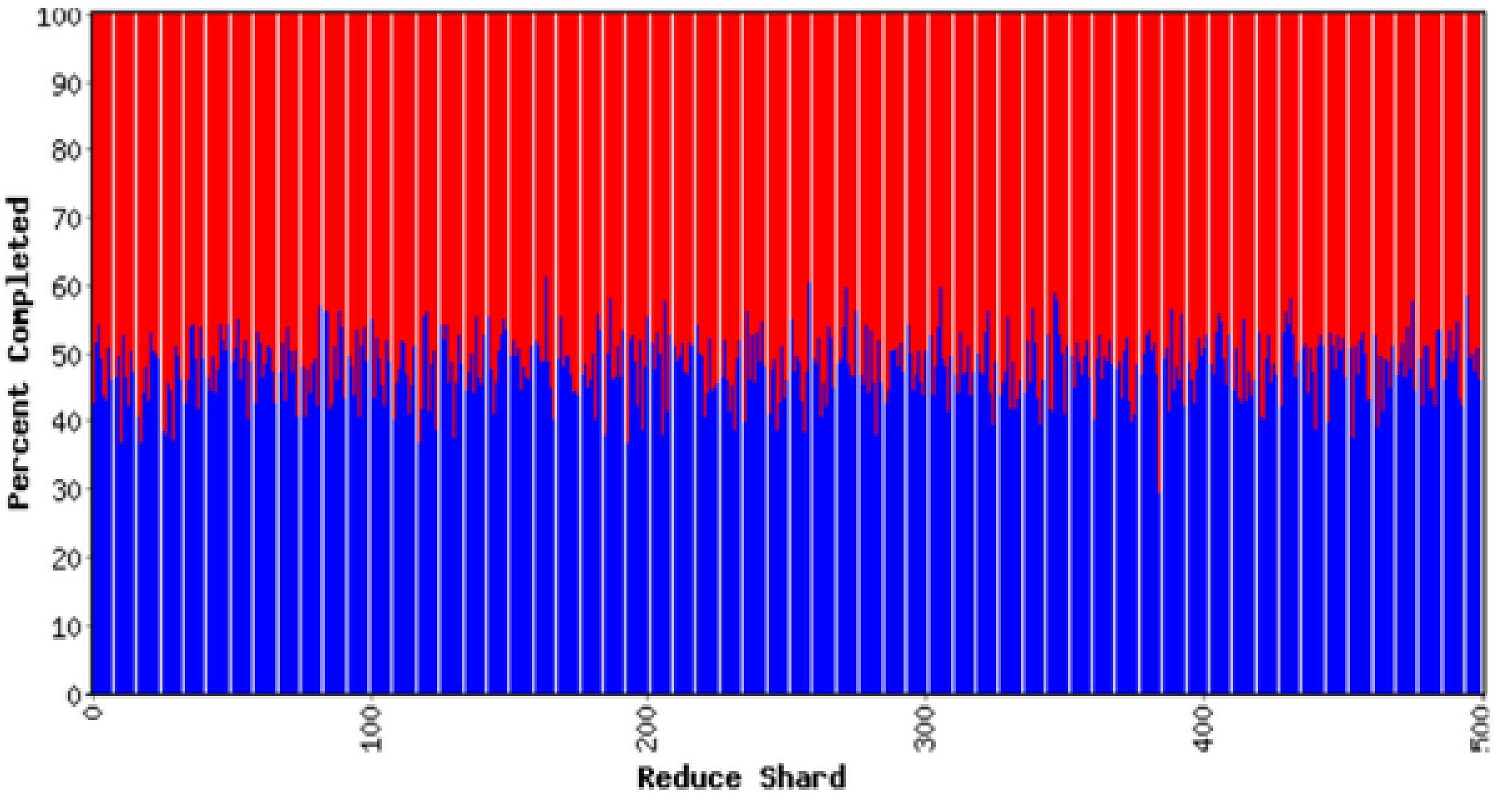
Variable	Minute	
Mapped (MB/s)	0.0	
Shuffle (MB/s)	0.1	
Output (MB/s)	1238.8	
doc-index-hits	0	1(
docs-indexed	0	
dups-in-index-merge	0	
mr-merge-calls	51738599	
mr-merge-outputs	51738599	

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 33 min 22 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	500	0	523499.2	523499.5	523499.5
Reduce	500	0	500	523499.5	263283.3	269351.2



Counters

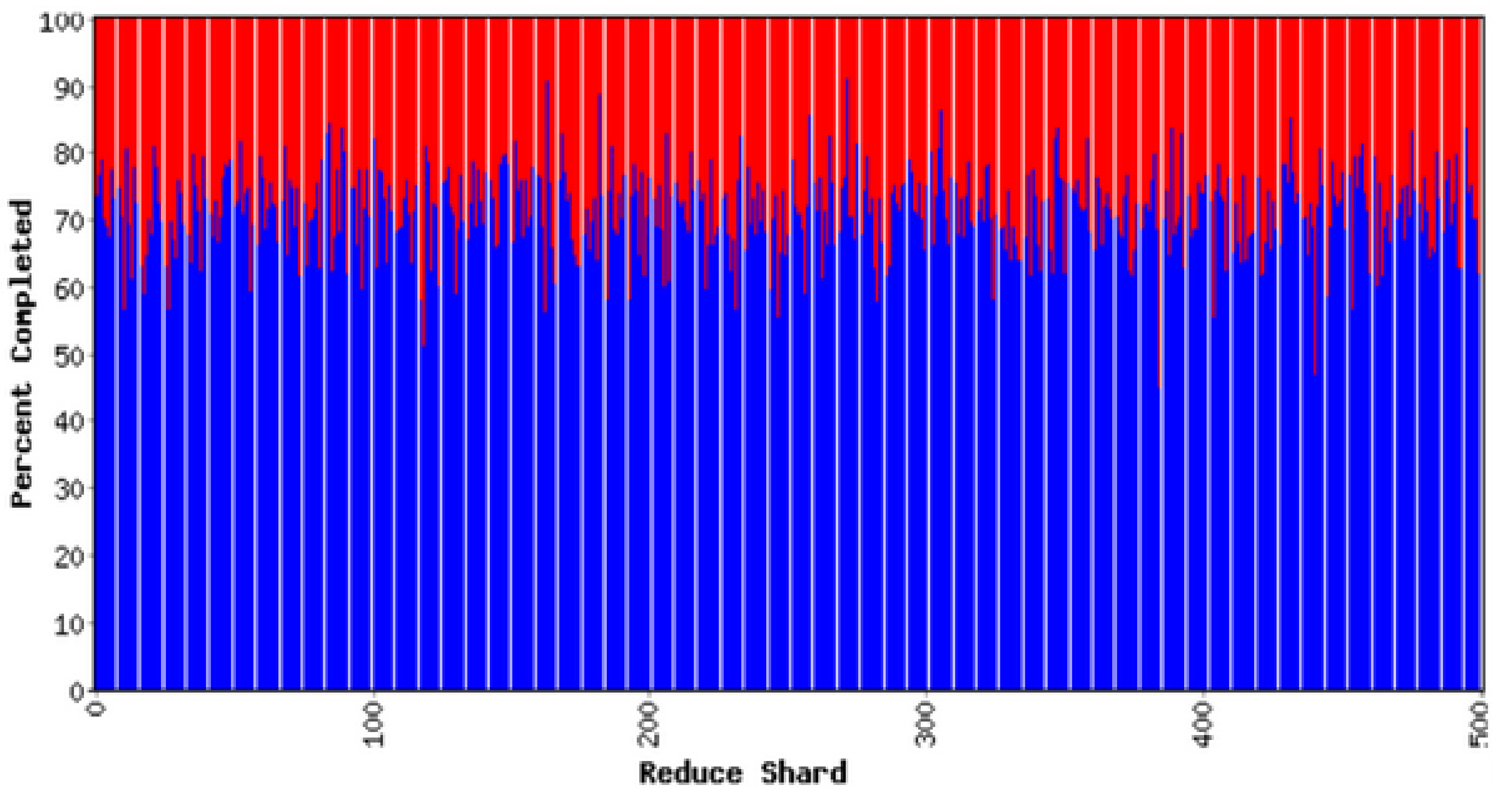
Variable	Minute
Mapped (MB/s)	0.0
Shuffle (MB/s)	0.0
Output (MB/s)	1225.1
doc-index-hits	0
docs-indexed	0
dups-in-index-merge	0
mr-merge-calls	51842100
mr-merge-outputs	51842100

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 35 min 08 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	500	0	523499.2	523499.5	523499.5
Reduce	500	0	500	523499.5	390447.6	399457.2



Counters

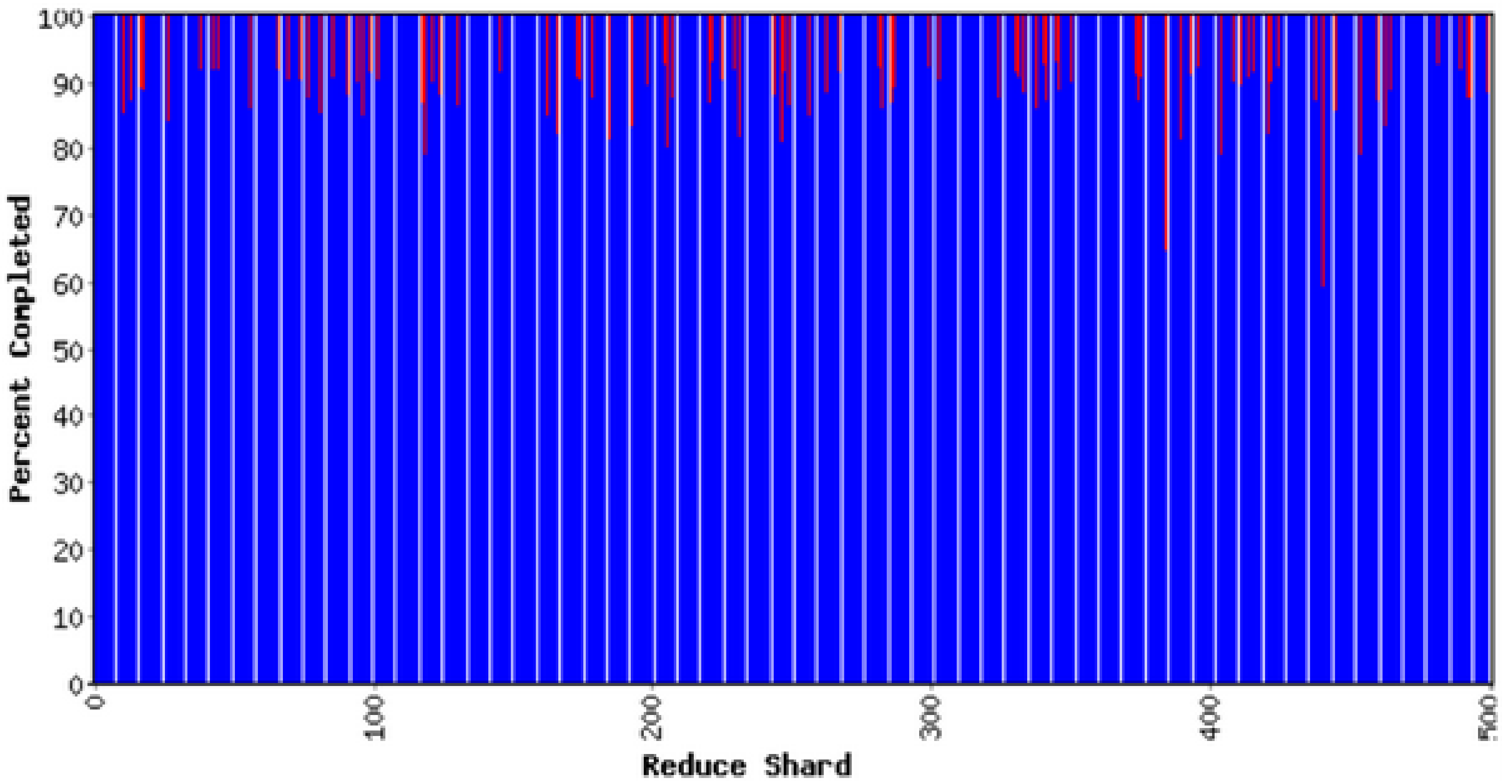
Variable	Minute	
Mapped (MB/s)	0.0	
Shuffle (MB/s)	0.0	
Output (MB/s)	1222.0	
doc-index-hits	0	1[illegible]
docs-indexed	0	
dups-in-index-merge	0	
mr-merge-calls	51640600	
mr-merge-outputs	51640600	

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 37 min 01 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
[Map]()	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	500	0	523499.2	520468.6	520468.6
[Reduce]()	500	406	94	520468.6	512265.2	514373.3



Counters

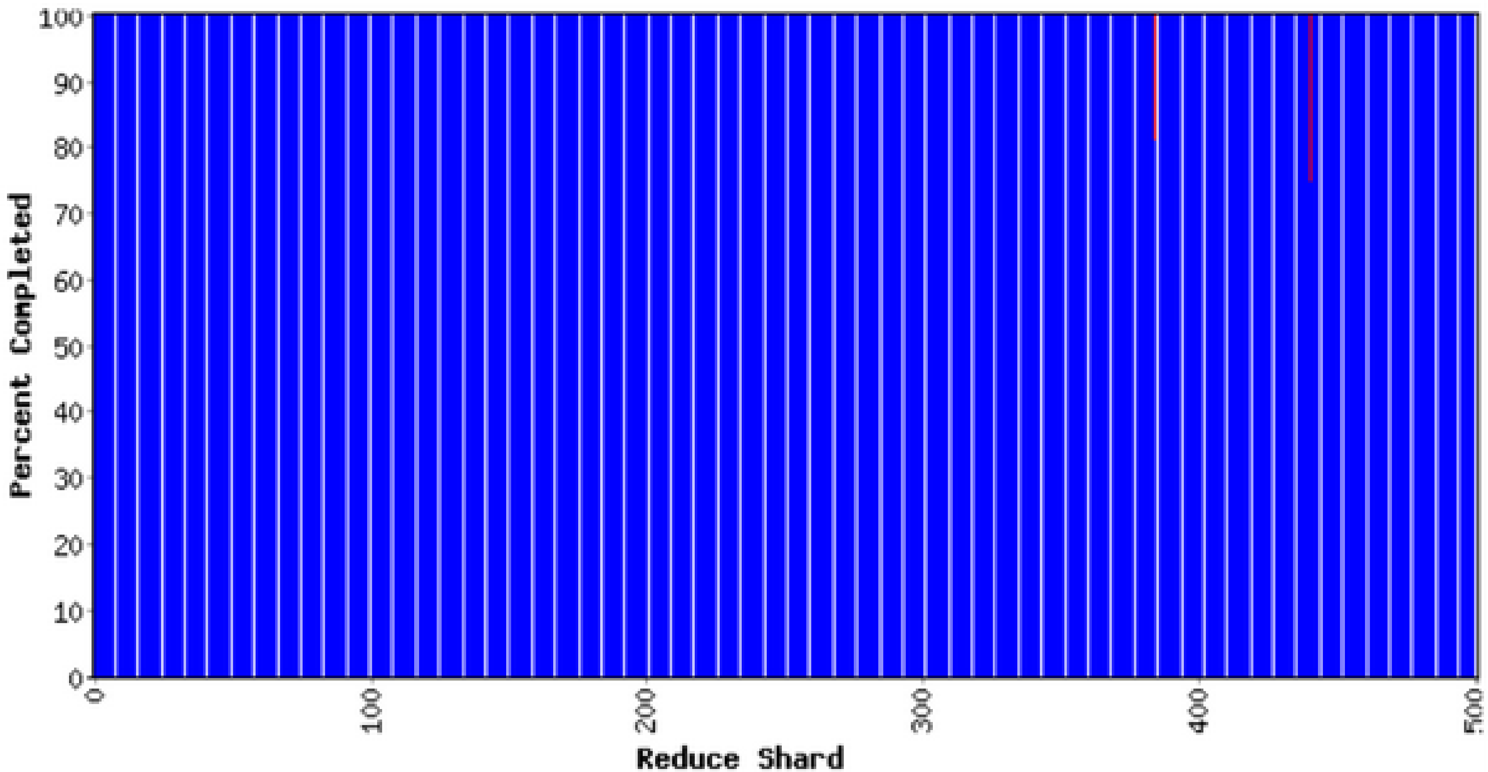
Variable	Minute	
Mapped (MB/s)	0.0	
Shuffle (MB/s)	0.0	
Output (MB/s)	849.5	
doc-index-hits	0	10
docs-indexed	0	
dups-in-index-merge	0	
mr-merge-calls	35083350	
mr-merge-outputs	35083350	

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 38 min 56 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	500	0	523499.2	519781.8	519781.8
Reduce	500	498	2	519781.8	519394.7	519440.7



Counters

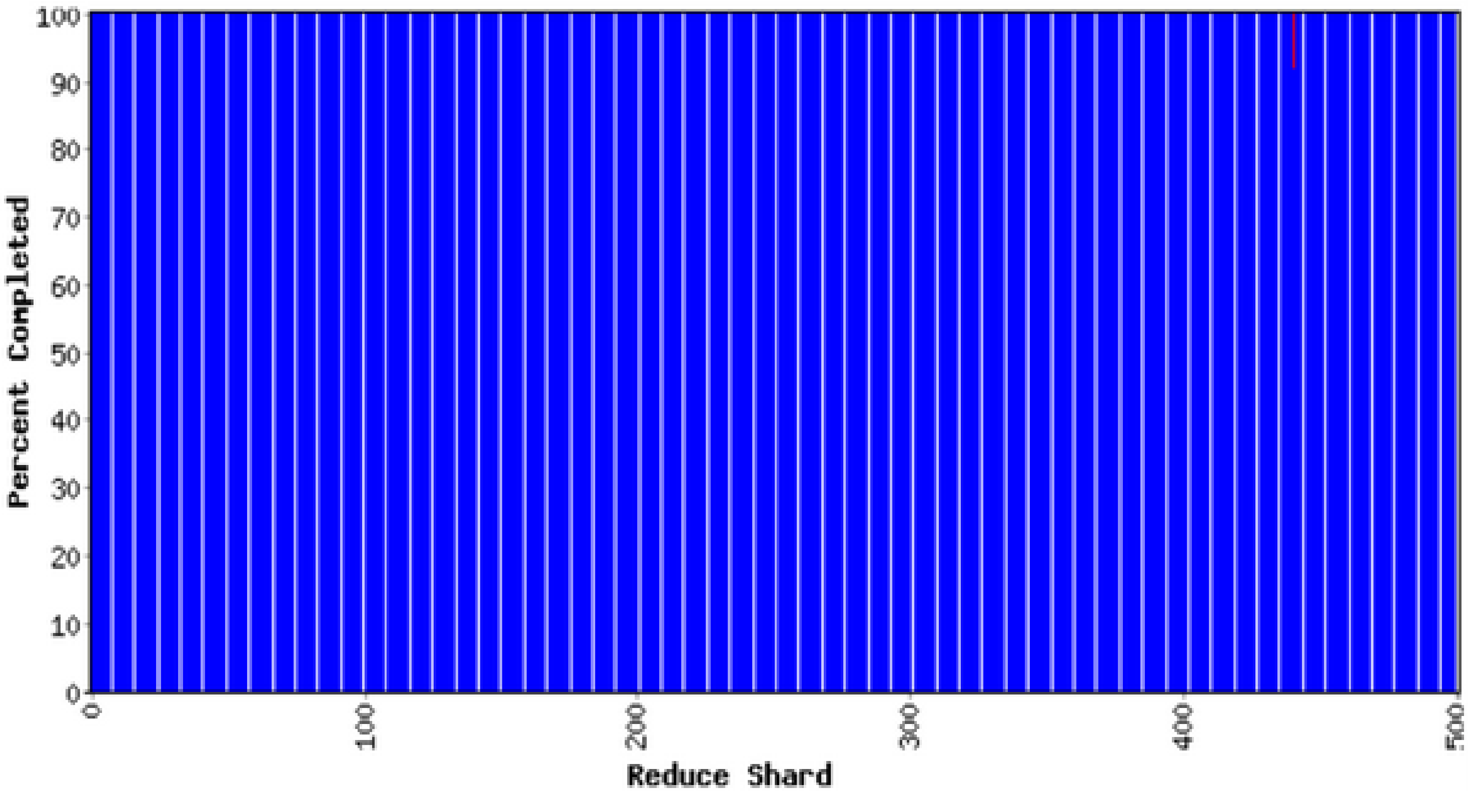
Variable	Minute	
Mapped (MB/s)	0.0	
Shuffle (MB/s)	0.0	
Output (MB/s)	9.4	
doc-index-hits	0	1056
docs-indexed	0	
dups-in-index-merge	0	
mr-merge-calls	394792	
mr-merge-outputs	394792	

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 40 min 43 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	500	0	523499.2	519774.3	519774.3
Reduce	500	499	1	519774.3	519735.2	519764.0



Counters

Variable	Minute	
Mapped (MB/s)	0.0	
Shuffle (MB/s)	0.0	
Output (MB/s)	1.9	
doc-index-hits	0	105
docs-indexed	0	
dups-in-index-merge	0	
mr-merge-calls	73442	
mr-merge-outputs	73442	

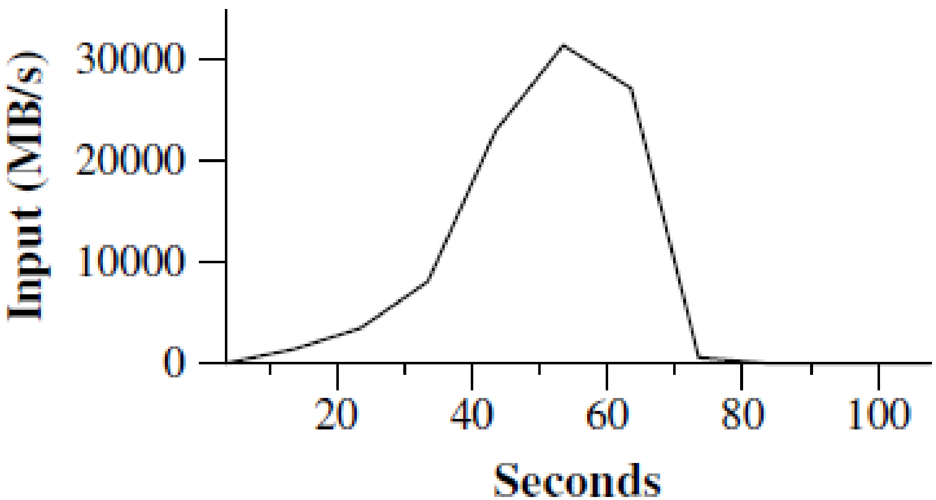
Performance

- Benchmarks:
  - MR_Grep - Scan 10^{10} 100-byte records to extract records matching a pattern (92K matching records)
  - MR_Sort - Sort 10^{10} 100-byte records (similar to TeraSort benchmark)

- Testbed:
  - Cluster of 1800 machines
  - Each machine has:
    - 4 GB of memory
    - Dual-processor 2 GHz Xeons with HT
    - Dual 160 GB IDE disks
    - Gigabit Ethernet

Performance

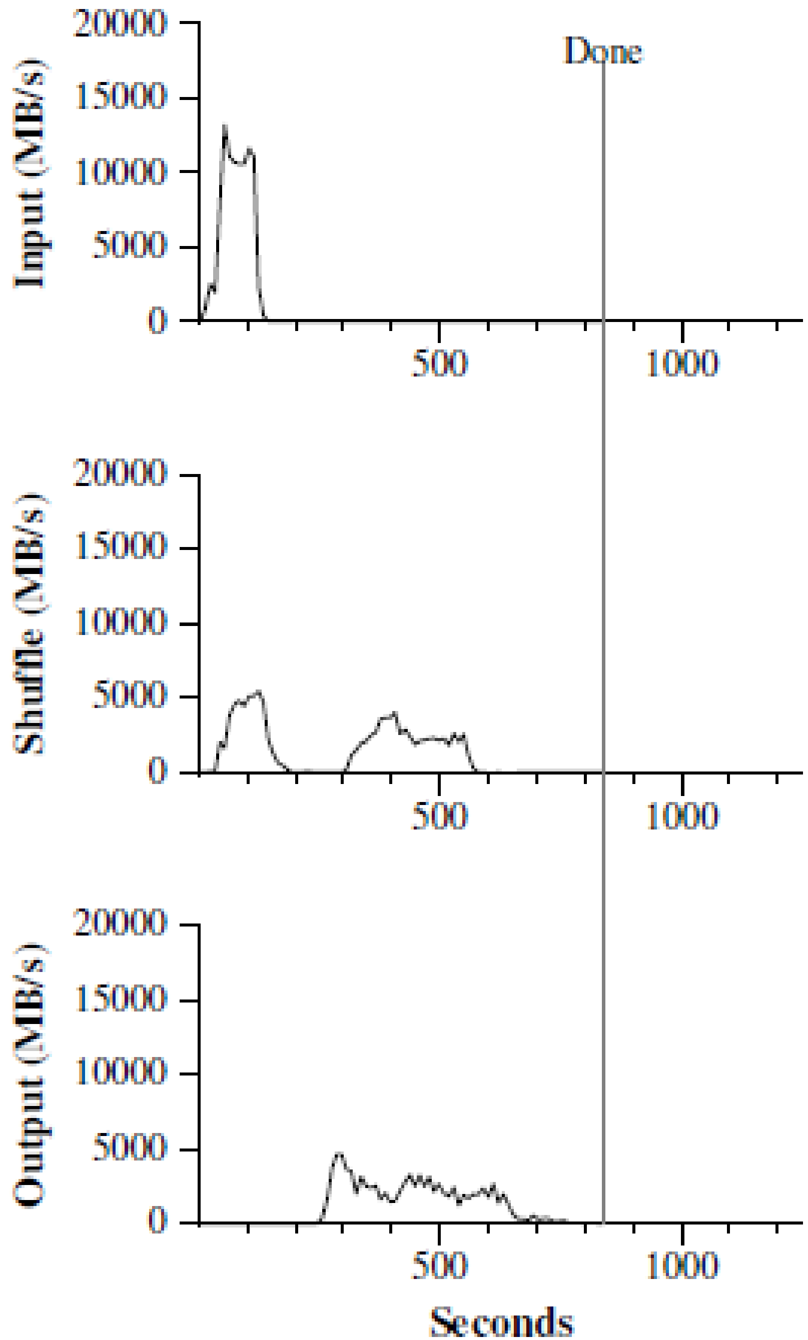
- MR_Grep
  - M=15000, R=1 (64 MB input splits)
  - total time – 150 secs



  - peak rate ~ 31GB/s
  - w/o locality optimization, peak rate < 10GB/s

Performance

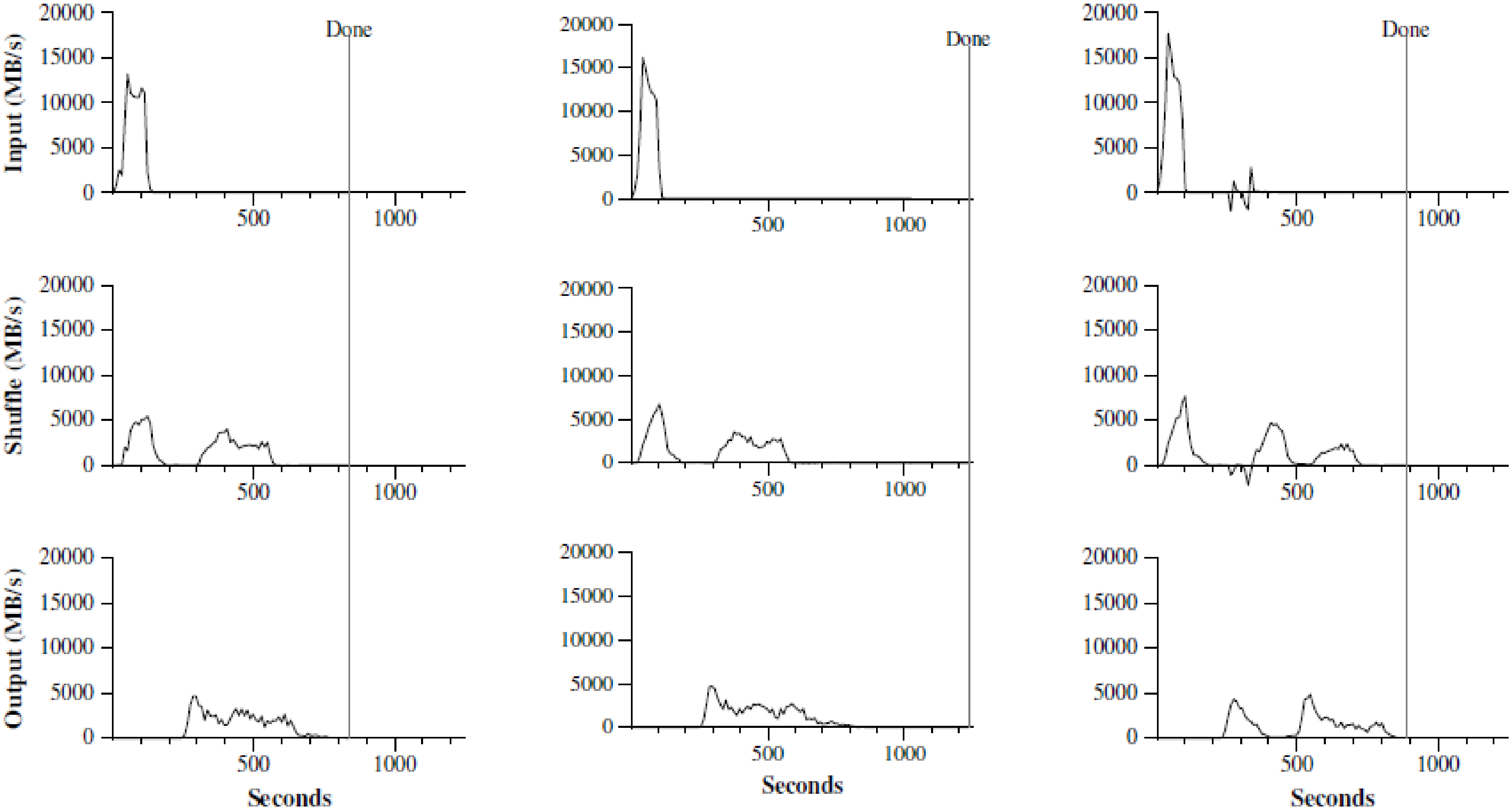
- MR_Sort
  - M=15000, R=4000
    (64 MB input splits)
  - 1 TB input
  - 2 TB output
    (2-way replication)
  - total time – 891 seconds



Performance

- Impact of Backup Tasks – MR_Sort
  - After 960 seconds, all except 5 reduce tasks are completed – take 300 additional seconds to finish
  - MR_sort takes 44% more time overall if backup tasks are disabled

- Impact of Machine Failures – MR_Sort
  - intentionally killed 200 workers some time after the computation started
  - overall time – 933 seconds (+5%)



(a) Normal execution (b) No backup tasks (c) 200 tasks killed

Chaining MR jobs

- Many problems which cannot be expressed easily with a single MR job
  - use a chain of MR jobs!

Map1 → Reduce1 → Map2 → Reduce2 → Map3 → Reduce3 → …

Example: Count the average number of characters in a line with has a particular pattern

Distributed grep → Average calculator

MR on multicore systems

- MPI and shared-memory threads implementations are too complex and error-prone
- Needs to be tuned for efficiency on different platforms by the programmer
- Can we develop a simple interface like MR on multicore platforms?

MR on multicore systems

- To simplify parallel programming we need 2 components:
  - practical programming model - allows to specify concurrency and locality at a high level
  - efficient runtime system – handles low-level mapping, resource management and fault tolerance

MR on multicore systems

- *Phoenix*: implementation of MR on shared-memory symmetric multiprocessor systems

Evaluating MapReduce for Multi-core and Multiprocessor Systems

Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, Christos Kozyrakis*
Computer Systems Laboratory
Stanford University

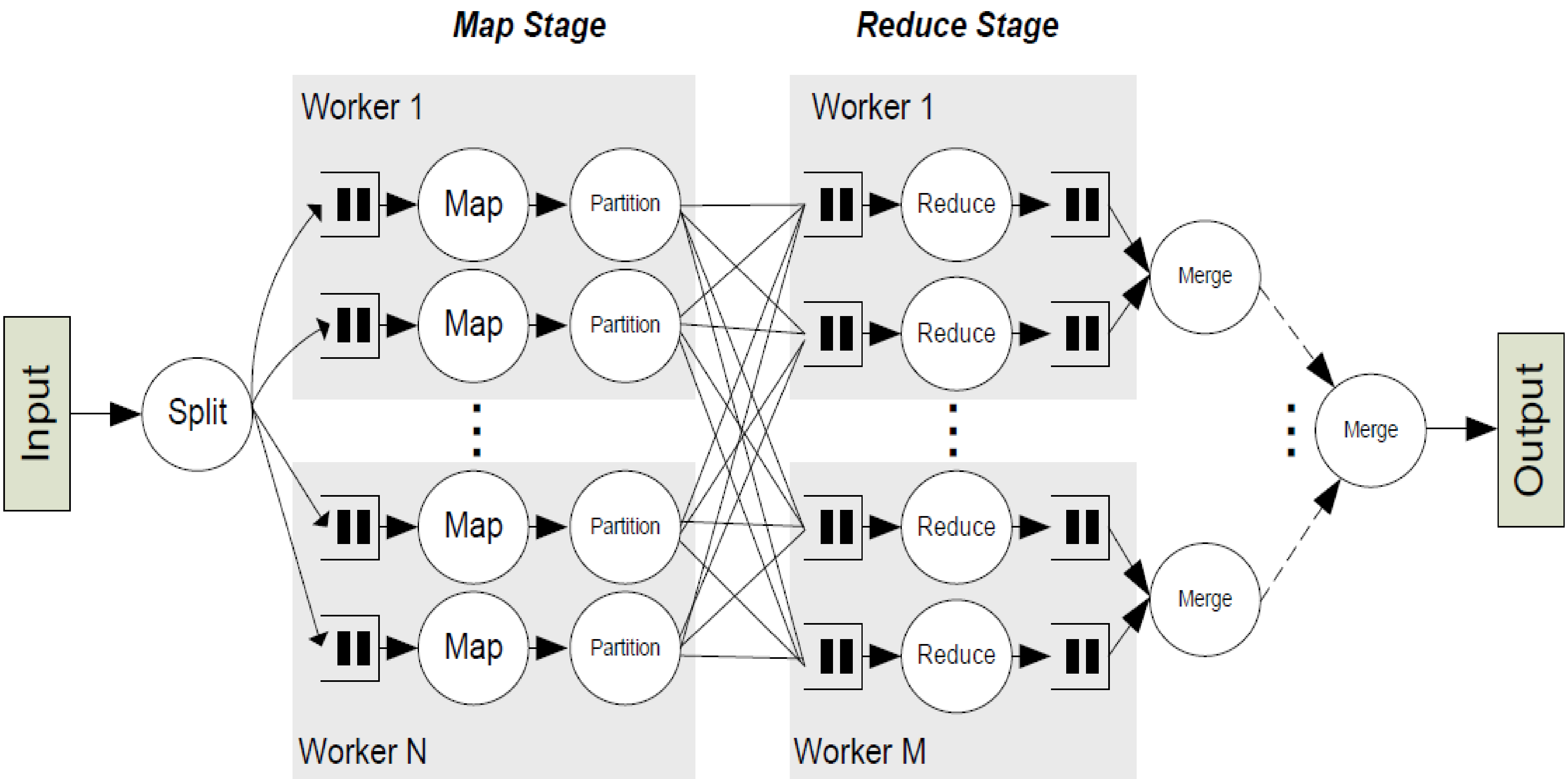
Abstract

This paper evaluates the suitability of the MapReduce model for multi-core and multi-processor systems. MapReduce was created by Google for application development on data-centers with thousands of servers. It allows programmers to write functional-style code that is automati-

efficient runtime system that handles low-level mapping, resource management, and fault tolerance issues automatically regardless of the system characteristics or scale. Naturally, the two components are closely linked. Recently, there has been a significant body of research towards these goals using approaches such as streaming [13, 15], memory transactions [14, 5], data-flow based schemes [2], asyn-

Phoenix

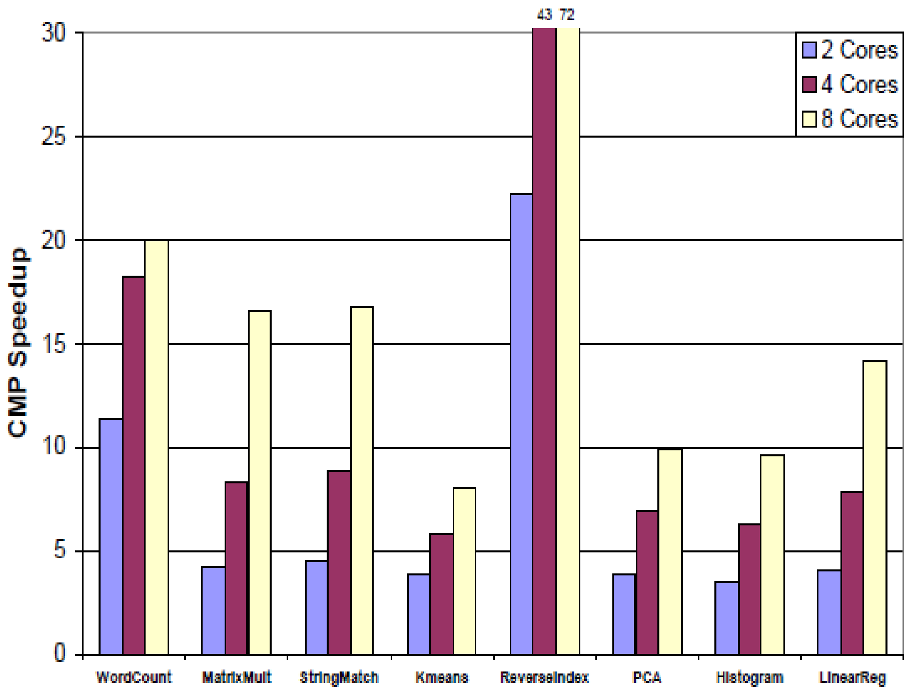
- uses threads instead of machines in a cluster for parallelism
- communication done via shared-memory instead of the network
- Phoenix Runtime:
  - assigns map and reduce to threads; handles buffer allocation and communication
  - dynamic scheduling for load balancing
  - locality optimization via granularity adjustment (input/output for map should fit in L1 cache)
  - detects and recovers from faults
  - mainly, hides a lot of low-level details from the programmer

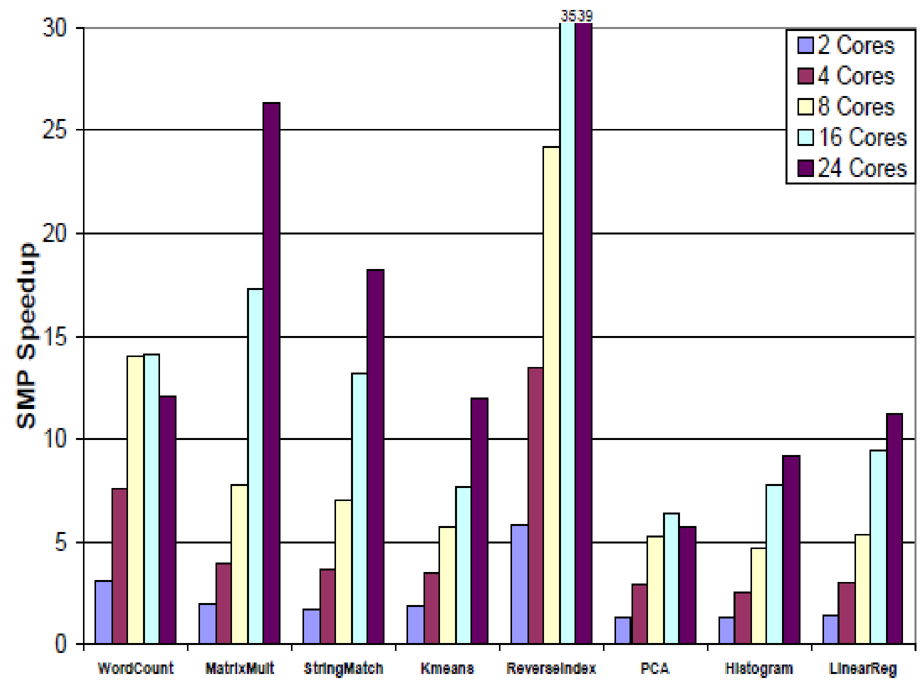


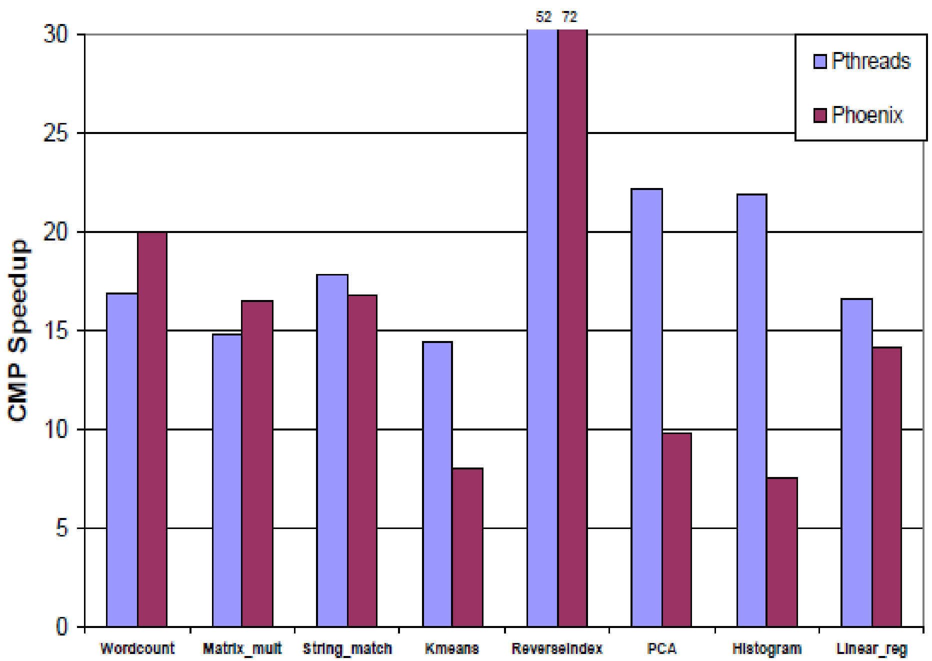
Map Stage
Reduce Stage
Input
Split
Worker 1
Map
Map
Partition
Partition
Map
Map
Partition
Partition
Worker N
Worker 1
Reduce
Reduce
Reduce
Reduce
Worker M
Merge
Merge
Merge
Output

Phoenix - Performance

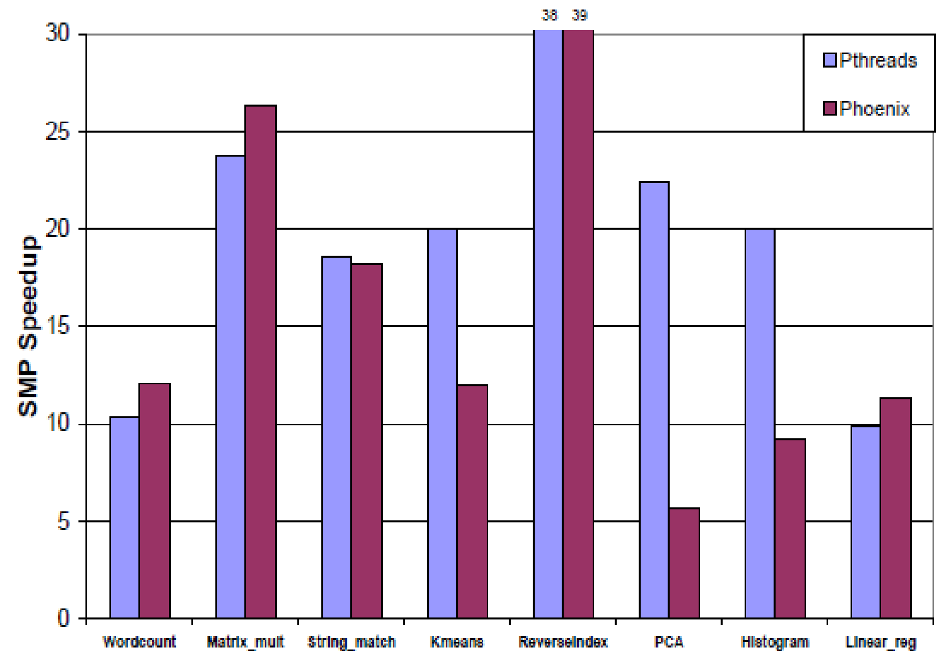
- Performance evaluated on 2 systems:
  - CMP: 1.2GHz Sun Fire T1200 (8 CPUs, 4 threads/CPU)
  - SMP: 250MHz Sun Ultra-Enterprise 6000 (24 CPUs, 1 thread/CPU)
- Computations:
  - word count, string match, reverse index, linear regression, matrix multiply, Kmeans, PCA, histogram of RGB components in an image
  - datasets of different sizes are used for different computations


CMP Speedup
43
72
2 Cores
4 Cores
8 Cores
WordCount
MatrixMult
StringMatch
Kmeans
ReverseIndex
PCA
Histogram
LinearReg


SMP Speedup
3539
2 Cores
4 Cores
8 Cores
16 Cores
24 Cores
WordCount
MatrixMult
StringMatch
Kmeans
ReverseIndex
PCA
Histogram
LinearReg



CMP Speedup
30
25
20
15
10
5
0
52
72
Pthreads
Phoenix
Wordcount
Matrix_mult
String_match
Kmeans
ReverseIndex
PCA
Histogram
Linear_reg



SMP Speedup
30
25
20
15
10
5
0
38
39
Pthreads
Phoenix
Wordcount
Matrix_mult
String_match
Kmeans
ReverseIndex
PCA
Histogram
Linear_reg

MR on mobile platforms

- Misco: MapReduce framework for mobile systems
  - uses mobile devices as nodes to schedule map and reduce tasks
  - works on any device which supports Python and has network connectivity
  - tested using 10 Nokia N95 phones connected to a Linksys router
  - can be used by applications which require more computing power than locally available
    - eg: processing images/videos

MapReduce – works everywhere?

- Real time computations
  - MR can be used for preprocessing data
- Small datasets
  - too much overhead
- Interactive analysis of data
- Anything which requires a lot of communication between tasks
- Anything where tasks depend on each other
- Stream processing
  - reduce waits for map to finish

Criticism for MapReduce

- nothing new – just a specific implementation of 25-30 year old techniques
  - MR imposes "simplified" data processing with cluster of cheap commodity machines
- not a DBMS
  - MR is a framework for one-off processing of data
- sub-optimal implementation (uses brute force instead of indexing to process data)
  - MR can be used to generate indexes but its not an optimized data storage and retrieval system

Conclusion

- MapReduce programming model has been a huge success
  - easy to use for programmers with no experience in distributed systems
  - hides details of parallelization, load balancing, fault tolerance, task management from the user
  - massively scalable
  - provides status monitoring tools

- Many open source implementations
  - eg: Hadoop

Thank you!

Questions?

Comparison with Parallel DBMS

- Parallel DBMS – similar to MR?
  - Parallelize query operation across multiple machines
- MapReduce:
  - Distributed file system
  - MR scheduler
  - Map, Combine and Reduce operations
- Parallel DBMS
  - Relational tables
  - Data spread over cluster nodes
  - SQL for programming

Comparison with Parallel DBMS

- Indexing
  - MR:
    - No direct support; indexes can be built
    - Customized indexes harder to reuse and share
  - DBMS
    - Use hash or b-tree for indexing
    - Fast access to any data

- Data format
  - MR:
    - No specific format required
  - DBMS:
    - Relational schema required

Comparison with Parallel DBMS

- Fault Tolerance:
  - MR:
    - Intermediate results stored to files
    - Quicker to recover from faults
  - DBMS:
    - No storage of intermediate results (send over network)
    - Lot of rework needed if a node fails

Comparison with Parallel DBMS

- Performance:
  - Cluster configuration:
    - 100 nodes
    - Each 2.4GHz Intel Core 2 Duo, 4GB RAM, 2 256GB SATA HDDs
  - Comparison of:
    - Hadoop
    - DBMS-X (row store)
    - Vertica (column store)

Comparison with Parallel DBMS

- Benchmark – Data Loading
  - Hadoop
    - Copy file in parallel to HDFS
  - DBMS-X
    - SQL load in parallel
    - Distribute records to machines, build index, compress data
  - Vertica
    - Load data in parallel; compress data

Comparison with Parallel DBMS

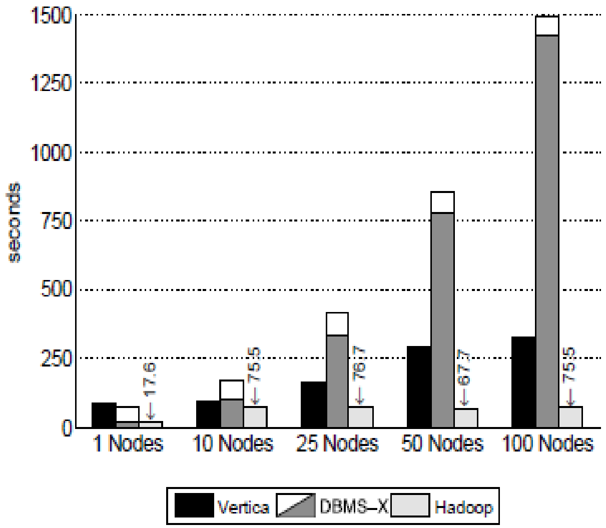


Figure 1: Load Times – Grep Task Data Set (535MB/node)

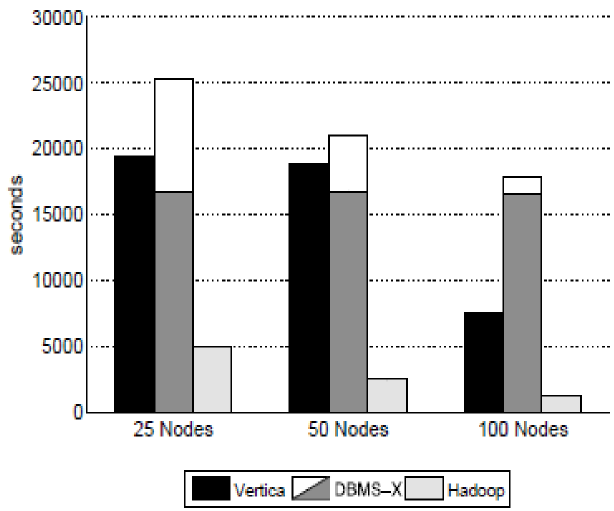


Figure 2: Load Times – Grep Task Data Set (1TB/cluster)

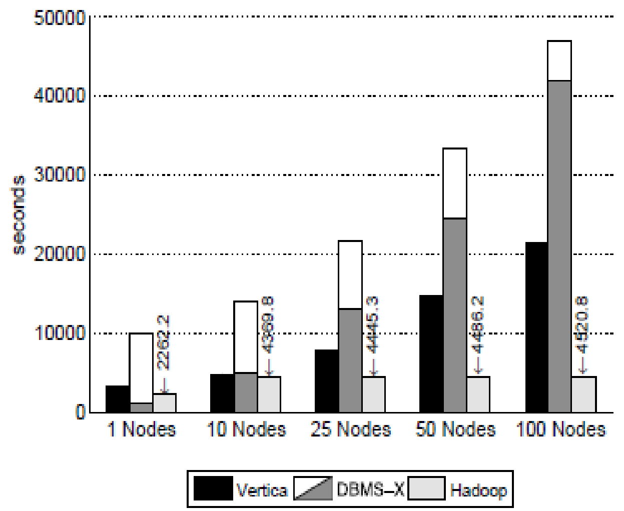


Figure 3: Load Times – UserVisits Data Set (20GB/node)

Comparison with Parallel DBMS

- Benchmark – grep for pattern
  - Hadoop
    - Map outputs line what matches a pattern
    - Identity Reduce
  - DBMS-X
    - SELECT * FROM data WHERE field LIKE "%XYZ%"
  - Vertica
    - SELECT * FROM data WHERE field LIKE "%XYZ%"

Comparison with Parallel DBMS

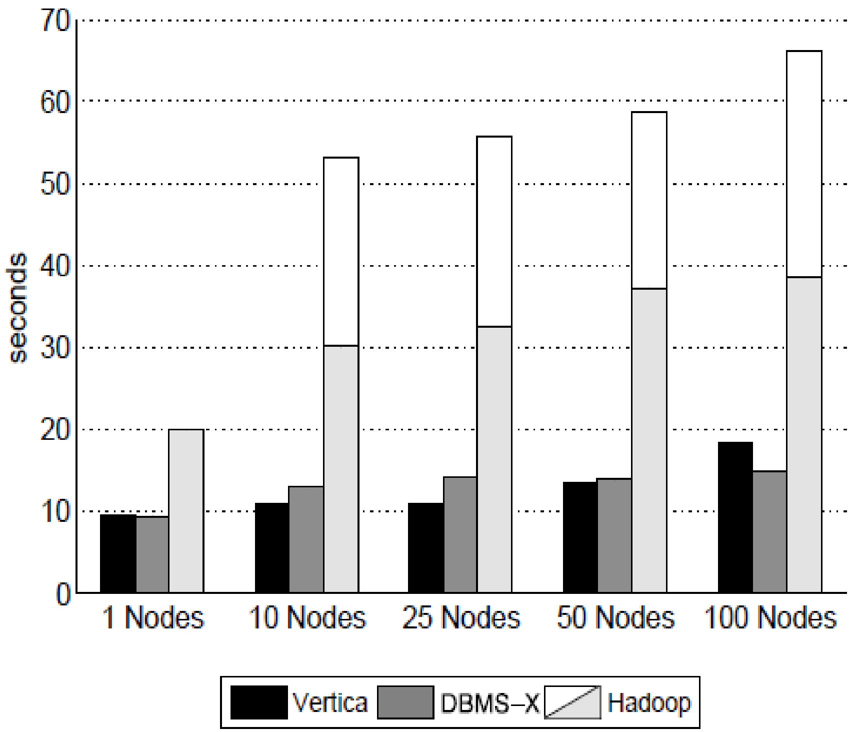


Figure 4: Grep Task Results – 535MB/node Data Set

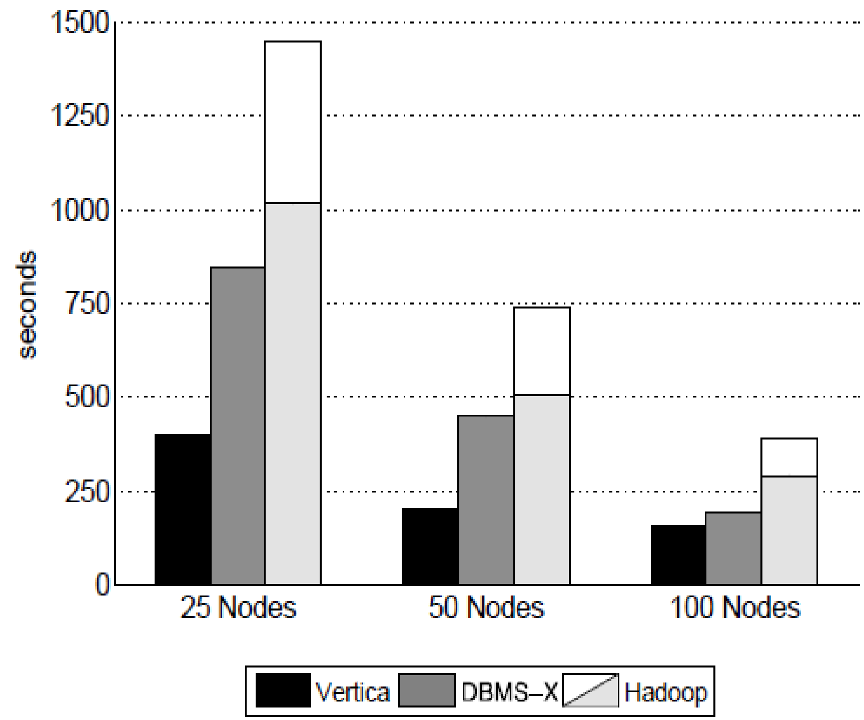


Figure 5: Grep Task Results – 1TB/cluster Data Set

Comparison with Parallel DBMS

- Conclusion
  - Advantages over MR:
    - Provide schema support
    - Indexing for faster access to data
    - Programming model is more expressive and easier
  - Disadvantages over MR:
    - Cant work with any arbitrary data
    - Load times for data are very high
    - MR is better at fault tolerance (less repeated work)