MapReduce

Concurrency for data-intensive applications

MapReduce

Jeff Dean

Sanjay Ghemawat

MapReduce: Simplified Data Processing on Large Clusters

by Jeffrey Dean and Sanjay Ghemawat

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large datasets that is amenable to a broad variety of real-world tasks. Users specify the computation in terms of a *map* and a *reduce* function, and the underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disks. Programmers find the system easy to use: more than ten thousand distinct MapReduce programs have been implemented internally at Google over the past four years, and an average of one hundred thousand MapReduce jobs are executed on Google's clusters every day, processing a total of more than twenty petabytes of data per day.

1 Introduction

Prior to our development of MapReduce, the authors and many others at Google implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, Web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of Web documents, summaries of the number of pages crawled per host, and the set of most frequent queries in a given day. 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 these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a map operation to each logical record' in our input in order to compute a set of intermediate key/value pairs, and then applying a reduce operation to all the values that shared the same key in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use reexecution as the primary mechanism for fault tolerance.

Biographies

Jeff Dean (jeff@google.com) is a Google Fellow and is currently working on a large variety of large-scale distributed systems at Google's Mountain View, CA, facility.

Sanjay Ghemawat (sanjay@google.com) is a Google Fellow and works on the distributed computing infrastructure used by most the company's products. He is based at Google's Mountain View, CA, facility.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs. The programming model can also be used to parallelize computations across multiple cores of the same machine.

Section 2 describes the basic programming model and gives several examples. In Section 3, we describe an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. In Section 6, we explore the use of MapReduce within Google including our experiences in using it as the basis for a rewrite of our production indexing system. Section 7 discusses related and future work.

2 Programming Model

The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs. The user of the MapReduce library expresses the computation as two functions: map and reduce.

Map, written by the user, takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key *I* and passes them to the reduce function.

The reduce function, also written by the user, accepts an intermediate key *I* and a set of values for that key. It merges these values together to form a possibly smaller set of values. Typically just zero or one output value is produced per reduce invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

2.1 Example

Consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code similar to the following pseudocode.

COMMUNICATIONS OF THE ACM January 2008/Vol. 51, No. 1 107

Motivation

- Application characteristics
  - Large/massive amounts of data
  - Simple application processing requirements
  - Desired portability across variety of execution platforms

- Execution platforms

	Cluster	CMP/SMP	GPGPU
Architecture	SPMD	MIMD	SIMD
Granularity	Process	Thread x 10	Thread x 100
Partition	File	Buffer	Sub-array
Bandwidth	Scare	GB/sec	GB/sec x 10
Failures	Common	Uncommon	Uncommon

Motivation

- Programming model
  - Purpose
    - Focus developer time/effort on salient (unique, distinguished) application requirements
    - Allow common but complex application requirements (e.g., distribution, load balancing, scheduling, failures) to be met by support environment
    - Enhance portability via specialized run-time support for different architectures
  - Pragmatics
    - Model correlated with characteristics of application domain
    - Allows simpler model semantics and more efficient support environment
    - May not express well applications in other domains

MapReduce model

- Basic operations
  - Map: produce a list of (key, value) pairs from the input structured as a (key value) pair of a different type

    `(k1,v1) → list (k2, v2)`

  - Reduce: produce a list of values from an input that consists of a key and a list of values associated with that key

    `(k2, list(v2)) → list(v2)`

Note: inspired by map/reduce functions in Lisp and other functional programming languages.

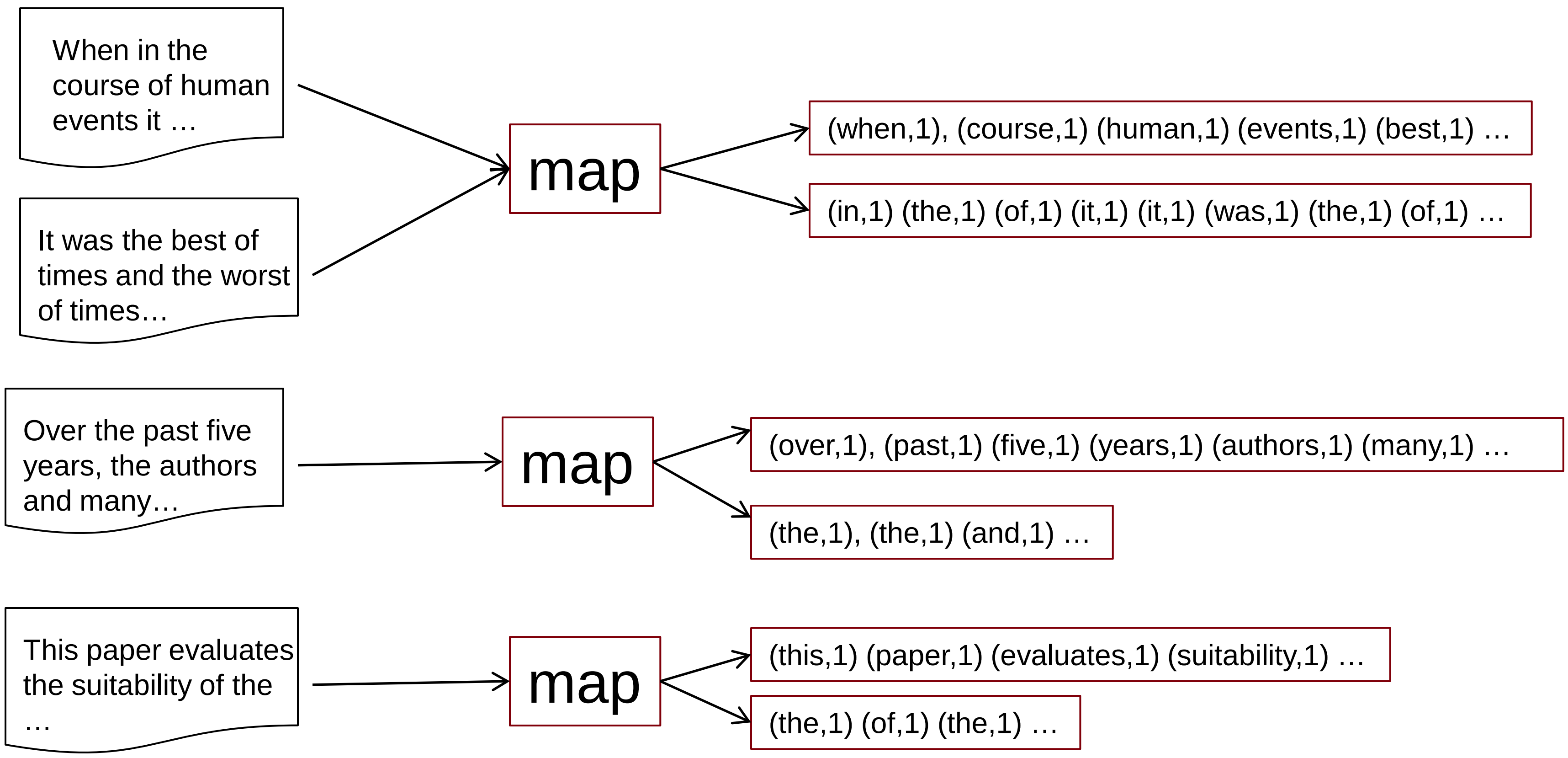
Example

```
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));
```

Example: map phase

inputs **tasks (M=3)** **partitions (intermediate files) (R=2)**



Note: partition function places small words in one partition and large words in another.

Example: reduce phase

partition (intermediate files) (R=2)

(in,1) (the,1) (of,1) (it,1) (it,1) (was,1) (the,1) (of,1) …

(the,1), (the,1) (and,1) …

(the,1) (of,1) (the,1) …

reduce task

sort — run-time function

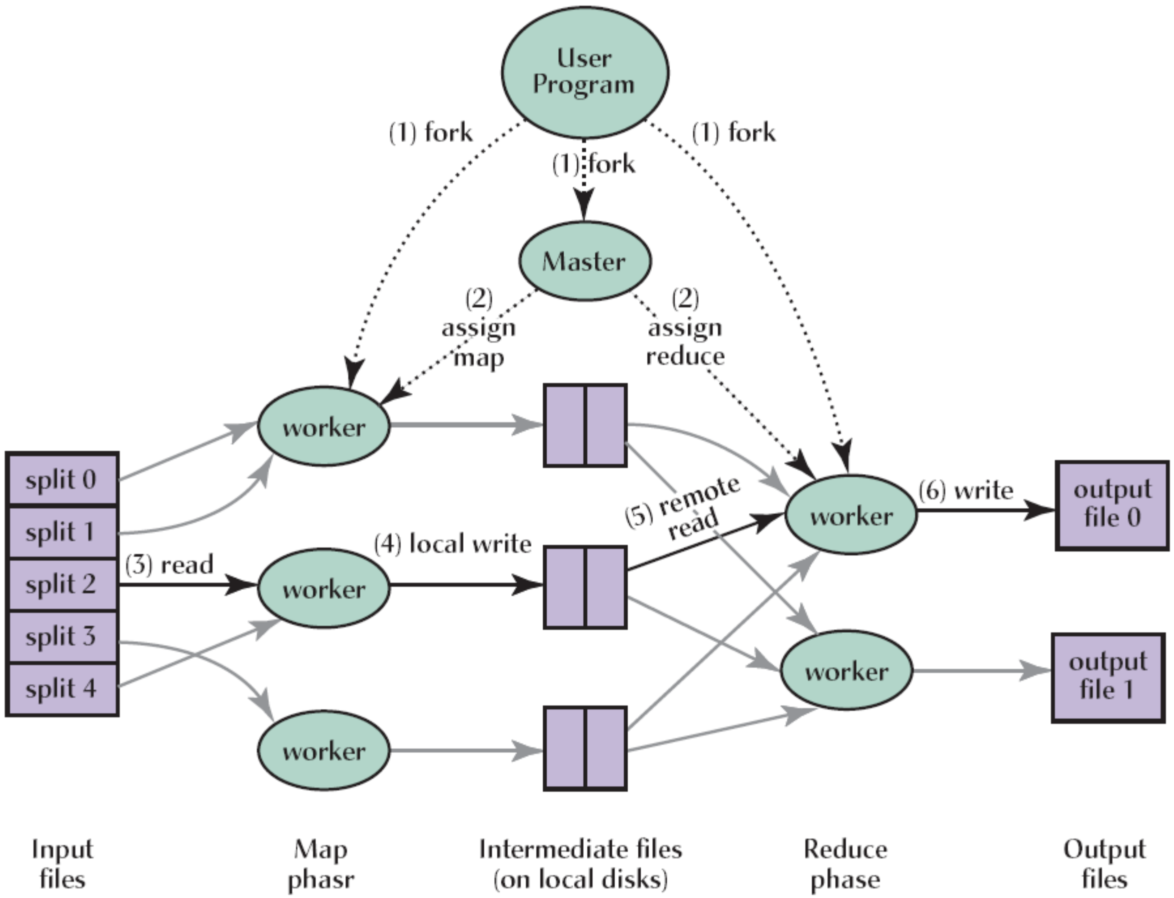
(and, (1)) (in,(1)) (it, (1,1)) (the, (1,1,1,1,1,1))
(of, (1,1,1)) (was,(1))

reduce — user's function

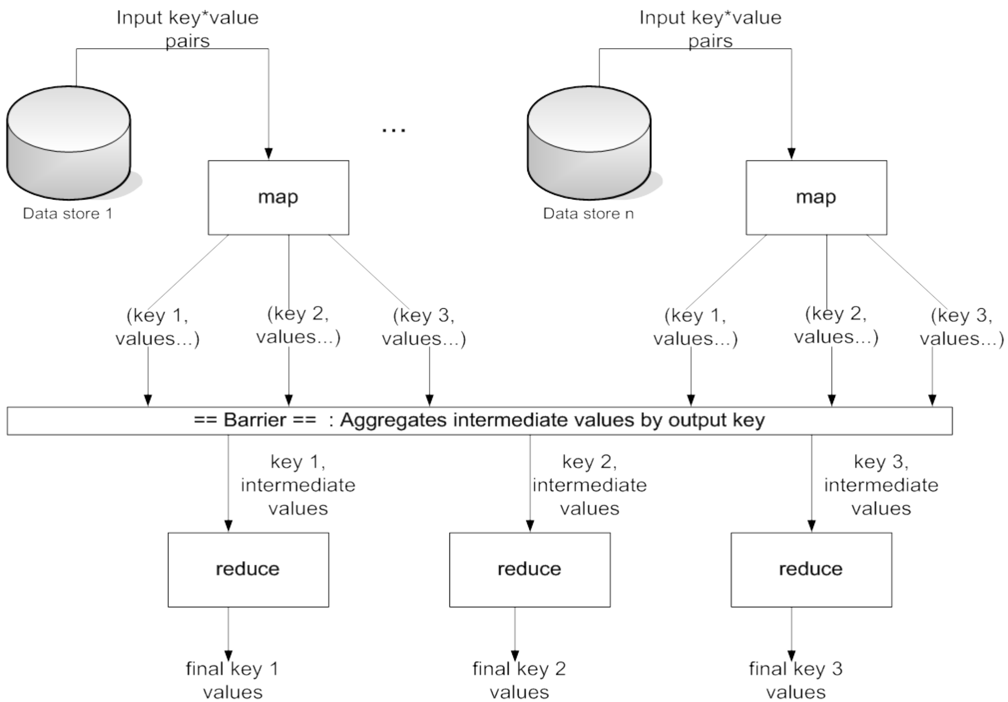
(and,1) (in,1) (it, 2) (of, 3) (the,6) (was,1)

Note: only one of the two reduce tasks shown

Execution Environment



Execution Environment



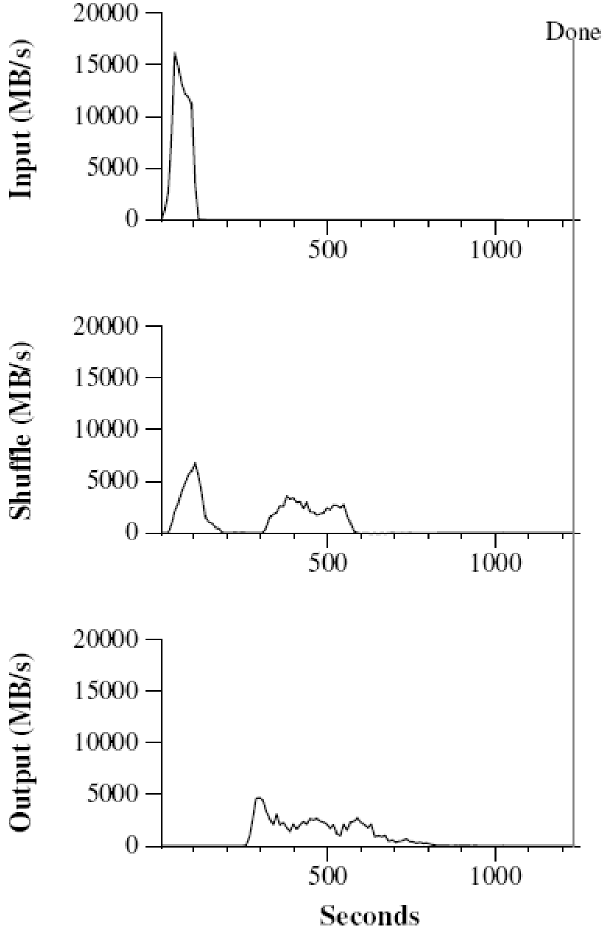
- No *reduce* can begin until *map* is complete
- Tasks scheduled based on location of data
- If *map* worker fails any time before *reduce* finishes, task must be completely rerun
- Master must communicate locations of intermediate files

Note: figure and text from presentation by Jeff Dean.

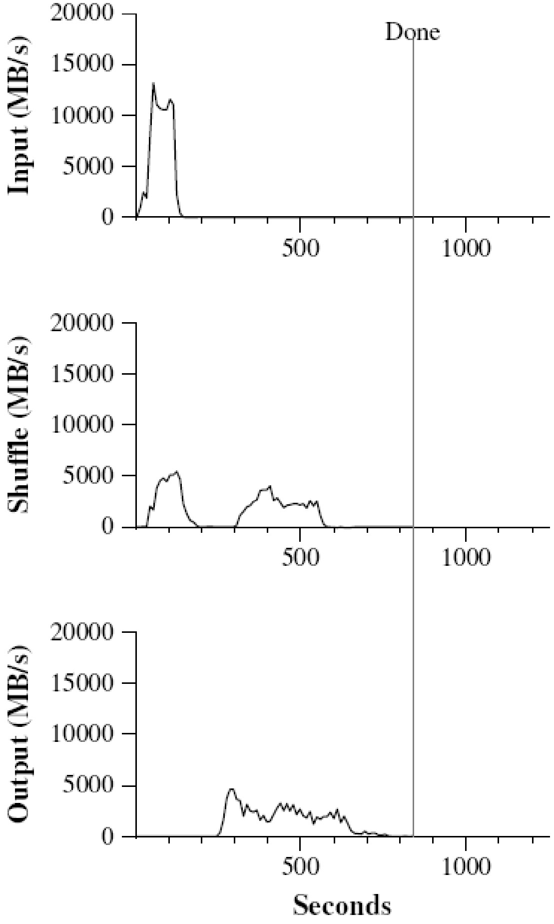
Backup Tasks

- A slow running task (straggler) prolong overall execution
- Stragglers often caused by circumstances local to the worker on which the straggler task is running
  - Overload on worker machined due to scheduler
  - Frequent recoverable disk errors
- Solution
  - Abort stragglers when map/reduce computation is near end (progress monitored by Master)
  - For each aborted straggler, schedule backup (replacement) task on another worker
- Can significantly improve overall completion time

Backup Tasks

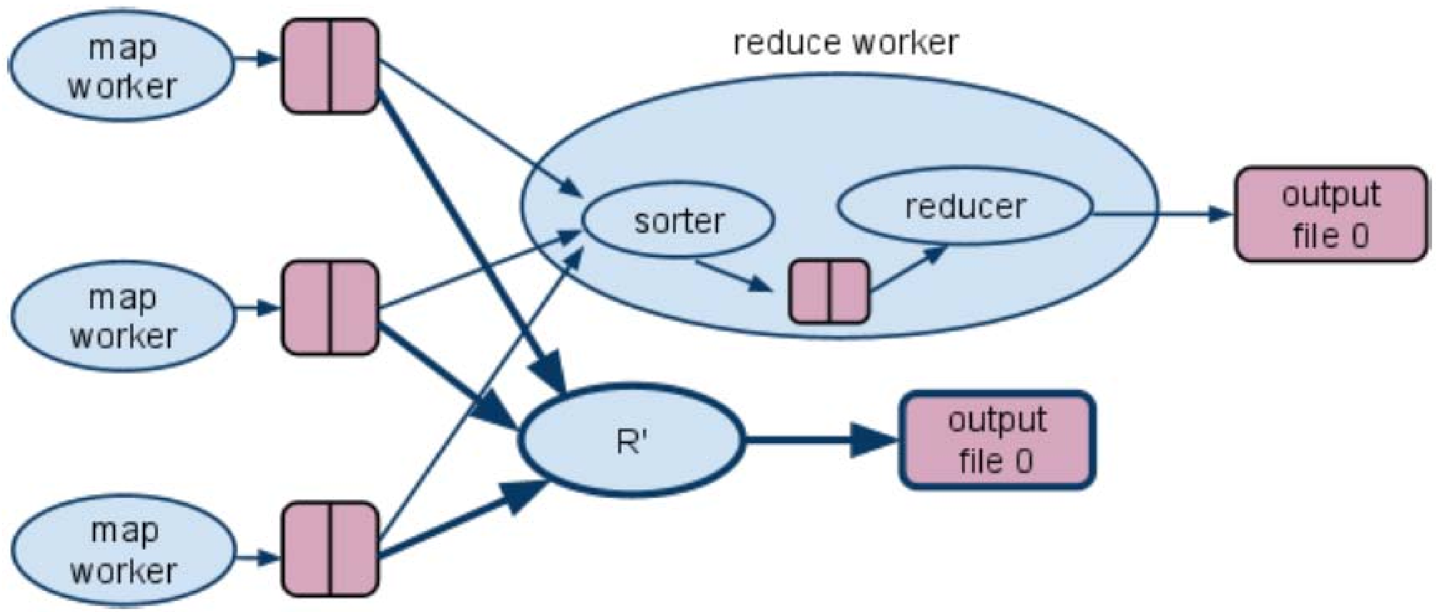


(1) without backup tasks



(2) with backup tasks (normal)

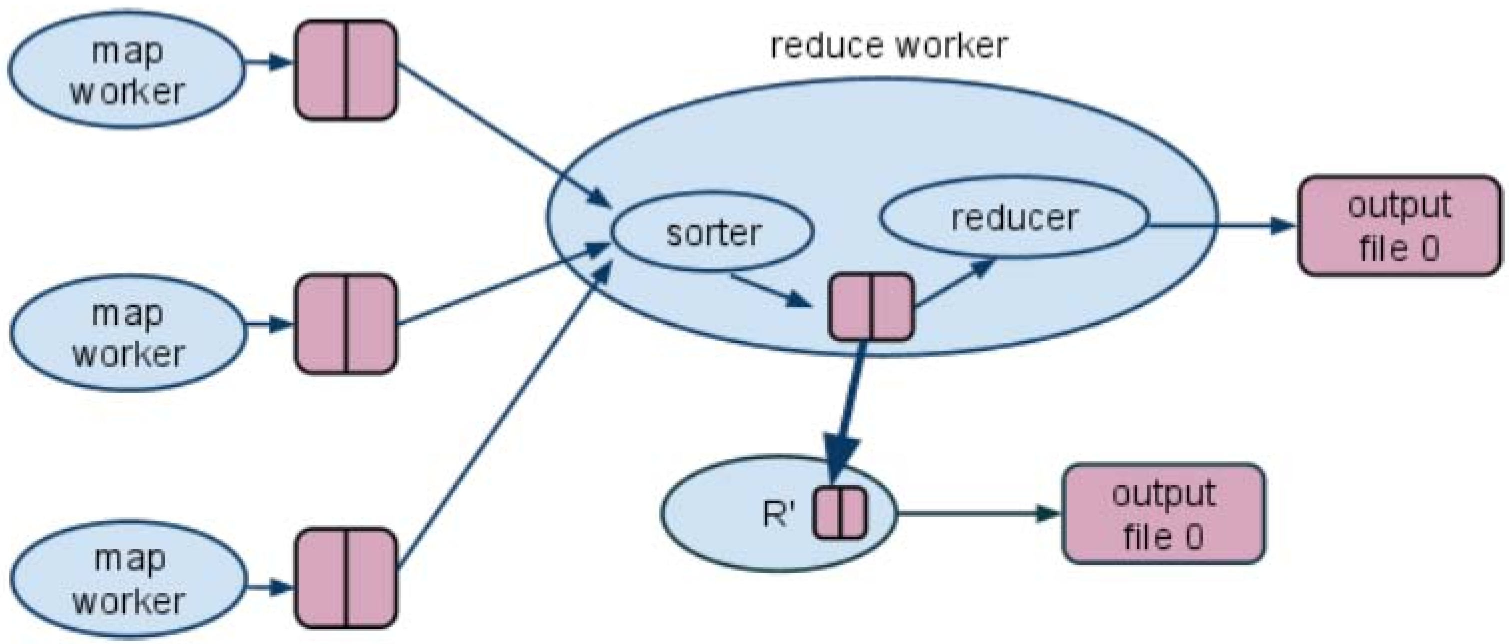
Strategies for Backup Tasks



(1) Create replica of backup task when necessary

Note: figure from presentation by Jerry Zhao and Jelena Pjesivac-Grbovic

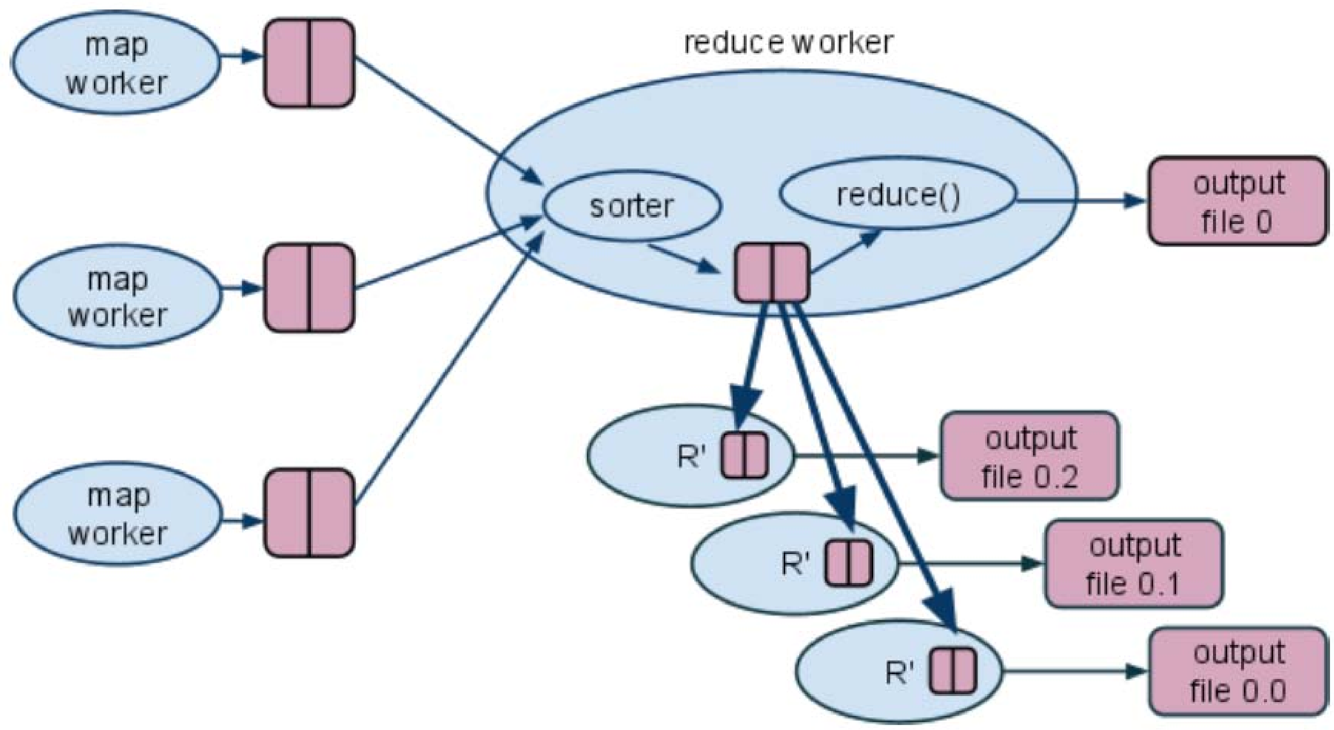
Strategies for Backup Tasks



(2) Leverage work completed by straggler - avoid resorting

Note: figure from presentation by Jerry Zhao and Jelena Pjesivac-Grbovic

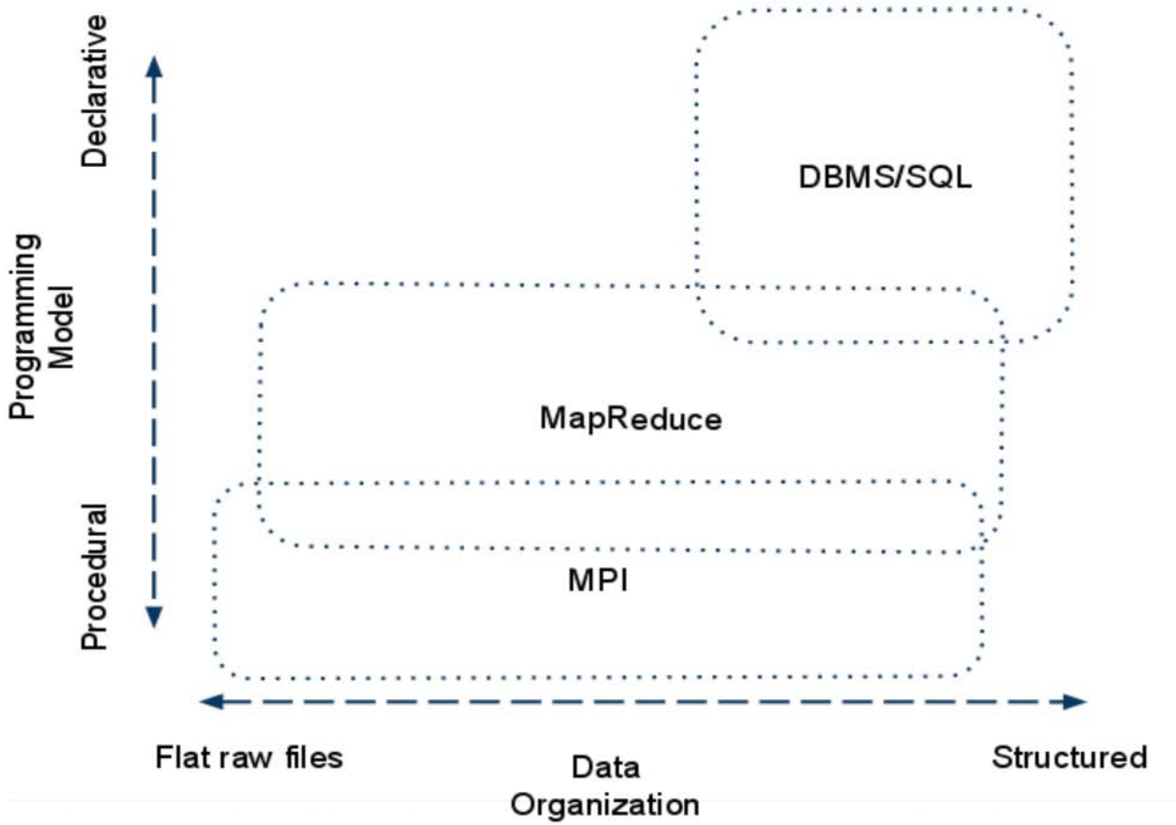
Strategies for Backup Tasks



(3) Increase degree of parallelism – subdivide partitions

Note: figure from presentation by Jerry Zhao and Jelena Pjesivac-Grbovic

Positioning MapReduce



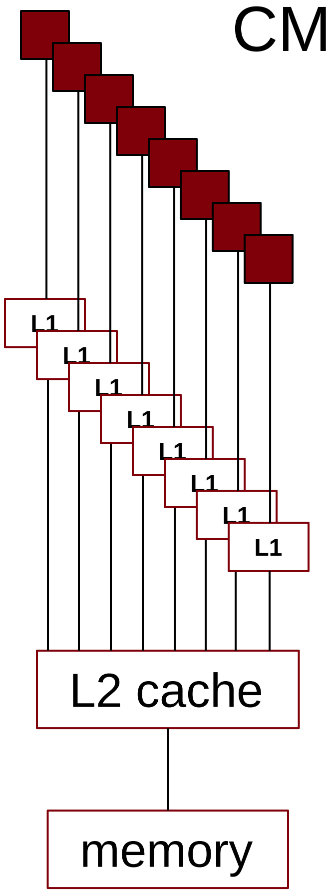
Note: figure from presentation by Jerry Zhao and Jelena Pjesivac-Grbovic

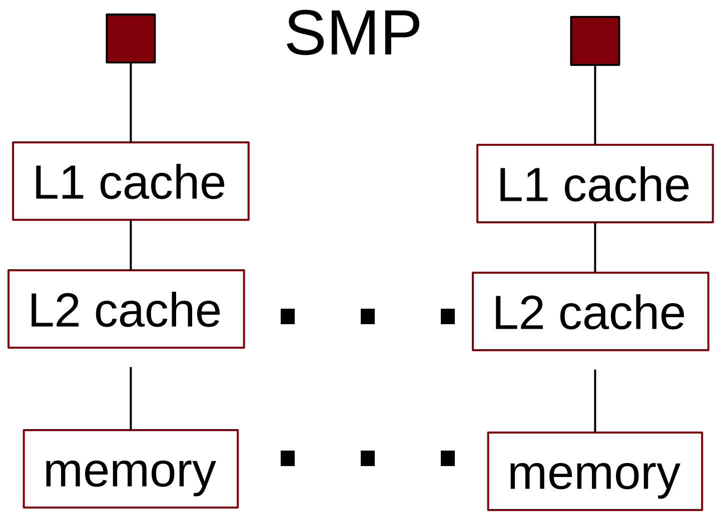
Positioning MapReduce

	MPI	MapReduce	DBMS/SQL
What they are	A general parrellel programming paradigm	A programming paradigm and its associated execution system	A system to store, manipulate and serve data.
Programming Model	Messages passing between nodes	Restricted to Map/Reduce operations	Declarative on data query/retrieving; Stored procedures
Data organization	No assumption	"files" can be sharded	Organized datastructures
Data to be manipulated	Any	k,v pairs: string/protomsg	Tables with rich types
Execution model	Nodes are independent	Map/Shuffle/Reduce Checkpointing/Backup Physical data locality	Transaction Query/operation optimization Materialized view
Usability	Steep learning curve*; difficult to debug	Simple concept Could be hard to optimize	Declarative interface; Could be hard to debug in runtime
Key selling point	Flexible to accommodate various applications	Plow through large amount of data with commodity hardware	Interactive querying the data; Maintain a consistent view across clients

Note: figure from presentation by Jerry Zhao and Jelena Pjesivac-Grbovic

MapReduce on SMP/CMP





	CMP	SMP
Model	Sun Fire T1200	Sun Ultra-Enterprise 6000
CPU Type	UltraSparc T1 single-issue in-order	UltraSparc II 4-way issue in-order
CPU Count	8	24
Threads/CPU	4	1
L1 Cache	8KB 4-way SA	16KB DM
L2 Size	3MB 12-way SA shared	512KB per CPU (off chip)
Clock Freq.	1.2 GHz	250 MHz

Phoenix runtime structure

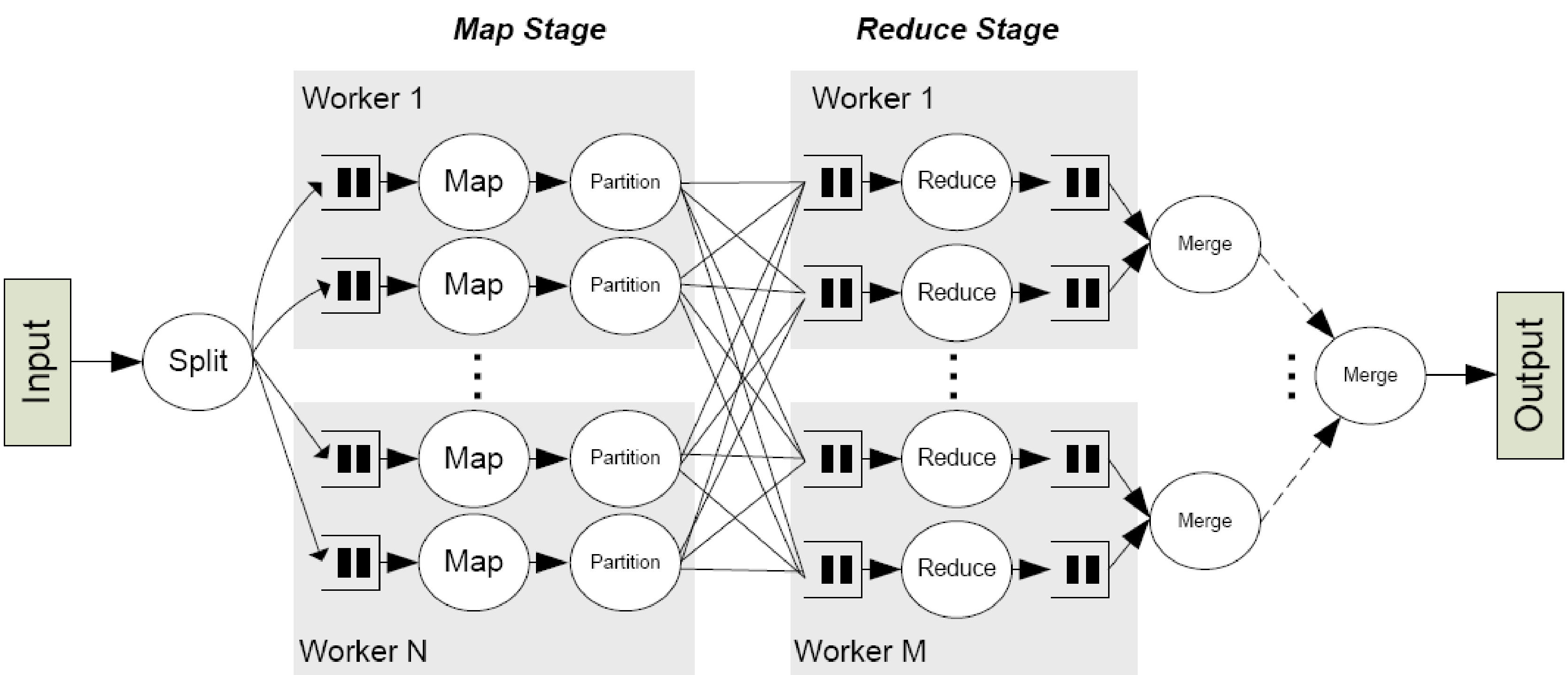


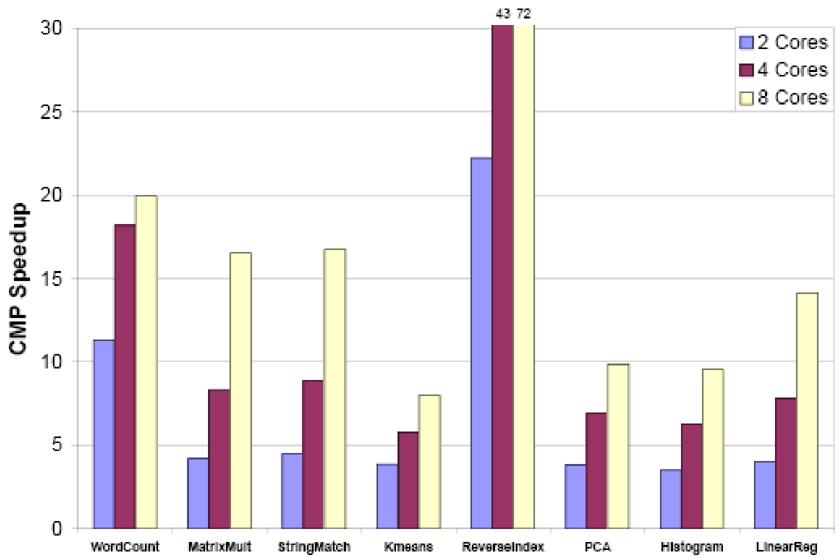
Figure 1. The basic data flow for the Phoenix runtime.

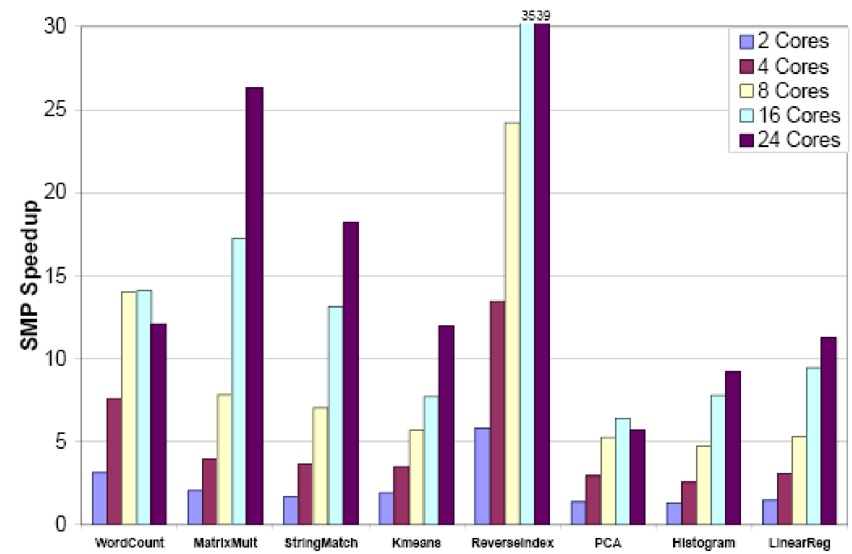
Code size

	Description	Data Sets	Code Size Ratio Pthreads	Code Size Ratio Phoenix
Word Count	Determine frequency of words in a file	S:10MB, M:50MB, L:100MB	1.8	0.9
Matrix Multiply	Dense integer matrix multiplication	S:100x100, M:500x500, L:1000x1000	1.8	2.2
Reverse Index	Build reverse index for links in HTML files	S:100MB, M:500MB, L:1GB	1.5	0.9
Kmeans	Iterative clustering algorithm to classify 3D data points into groups	S:10K, M:50K, L:100K points	1.2	1.7
String Match	Search file with keys for an encrypted word	S:50MB, M:100MB, L:500MB	1.8	1.5
PCA	Principal components analysis on a matrix	S:500x500, M:1000x1000, L:1500x1500	1.7	2.5
Histogram	Determine frequency of each RGB component in a set of images	S:100MB, M:400MB, L:1.4GB	2.4	2.2
Linear Regression	Compute the best fit line for a set of points	S:50M, M:100M, L:500M	1.7	1.6

- Comparison with respect to sequential code size
- Observations
  - Concurrency add significantly to code size (~ 40%)
  - MapReduce is code efficient in compatible applications
  - Overall, little difference in code size of MR vs Pthreads
  - Pthreads version lacks fault tolerance, load balancing, etc.
  - Development time and correctness not known

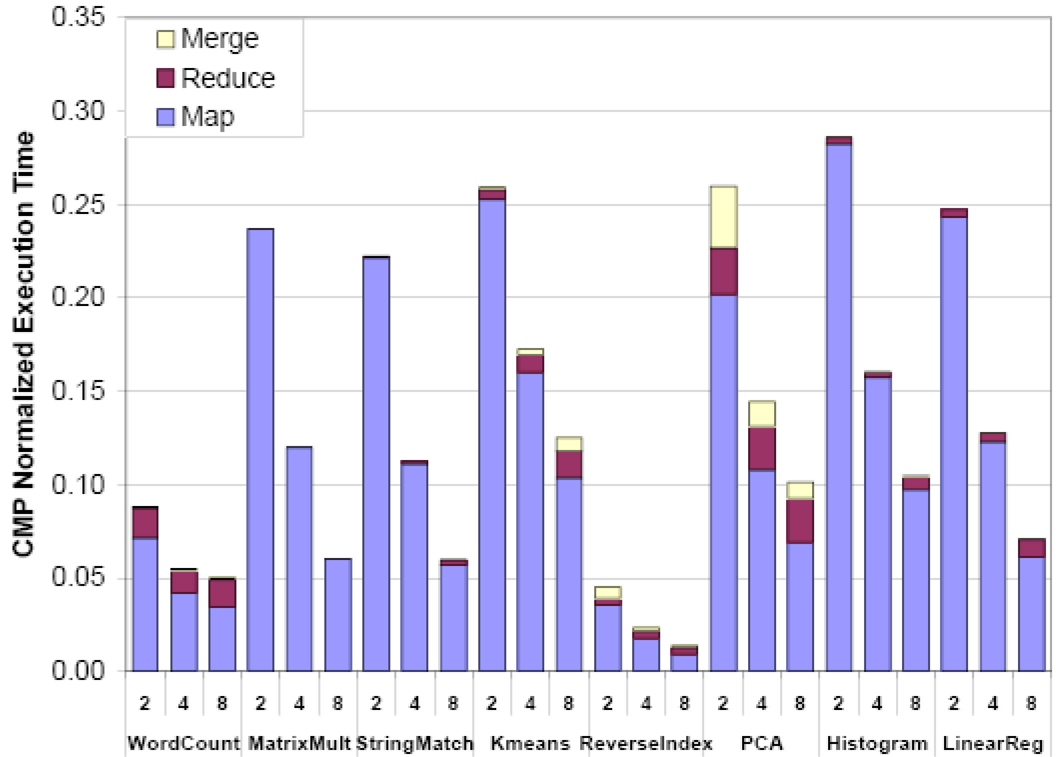
Speedup measures





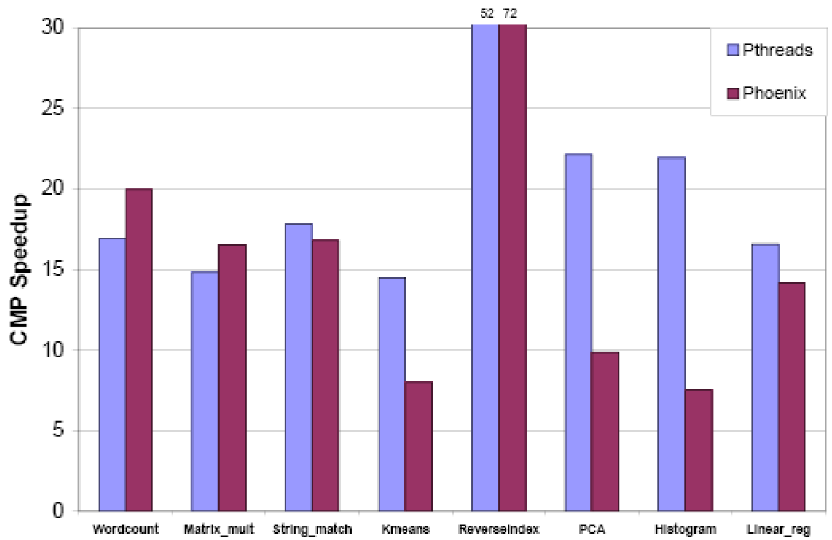
- Significant speedup is possible on either architecture
- Clear differences based on application characteristics
- Effects of application characteristics more pronounced than architectural differences
- Superlinear speedup due to
  - Increased cache capacity with more cores
  - Distribution of heaps lowers heap operation costs
  - More core and cache capacity for final merge/sort step

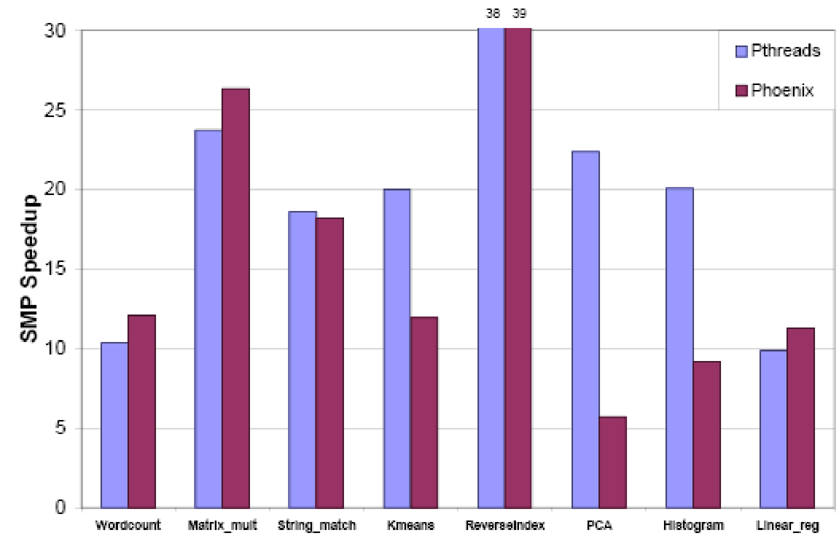
Execution time distribution



- Execution time dominated by Map task

MapReduce vs Pthreads



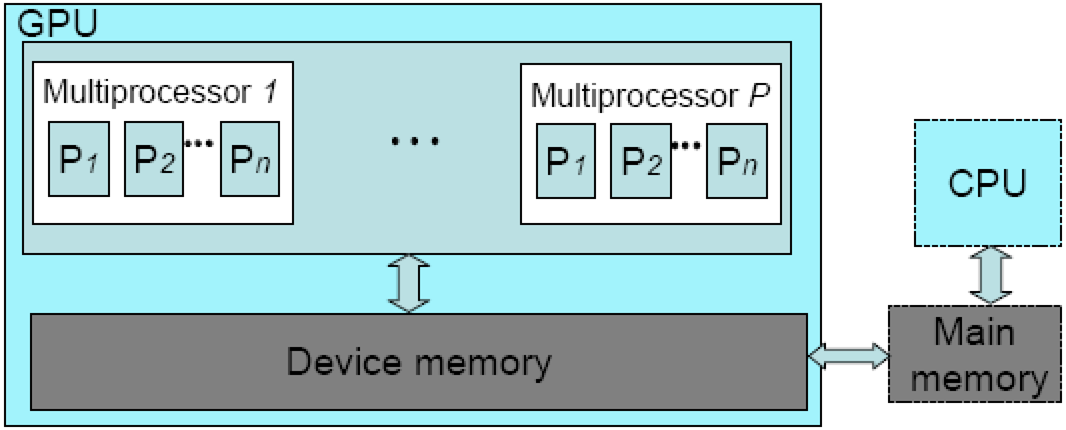


- MapReduce compares favorably with Pthreads on applications where the MapReduce programming model is appropriate
- MapReduce is not a general-purpose programming model

MapReduce on GPGPU

- General Purpose Graphics Processing Unit (GPGPU)
  - Available as commodity hardware
  - GPU vs. CPU
    - 10x more processors in GPU
    - GPU processors have lower clock speed
    - Smaller caches on GPU
  - Used previously for non-graphics computation in various application domains
  - Architectural details are vendor-specific
  - Programming interfaces emerging
- Question
  - Can MapReduce be implemented efficiently on a GPGPU?

GPGPU Architecture



- Many Single-instruction, Multiple-data (SIMD) multiprocessors
- High bandwidth to device memory
- GPU threads: fast context switch, low creation time
- Scheduling
  - Threads on each multiprocessor organized into thread groups
  - Thread groups are dynamically scheduled on the multiprocessors
- GPU cannot perform I/O; requires support from CPU
- Application: kernel code (GPU) and host code (CPU)

System Issues

- Challenges
  - Requires low synchronization overhead
  - Fine-grain load balancing
  - Core tasks of MapReduce are unconventional to GPGPU and must be implemented efficiently
  - Memory management
    - No dynamic memory allocation
    - Write conflicts occur when two threads write to the same shared region

System Issues

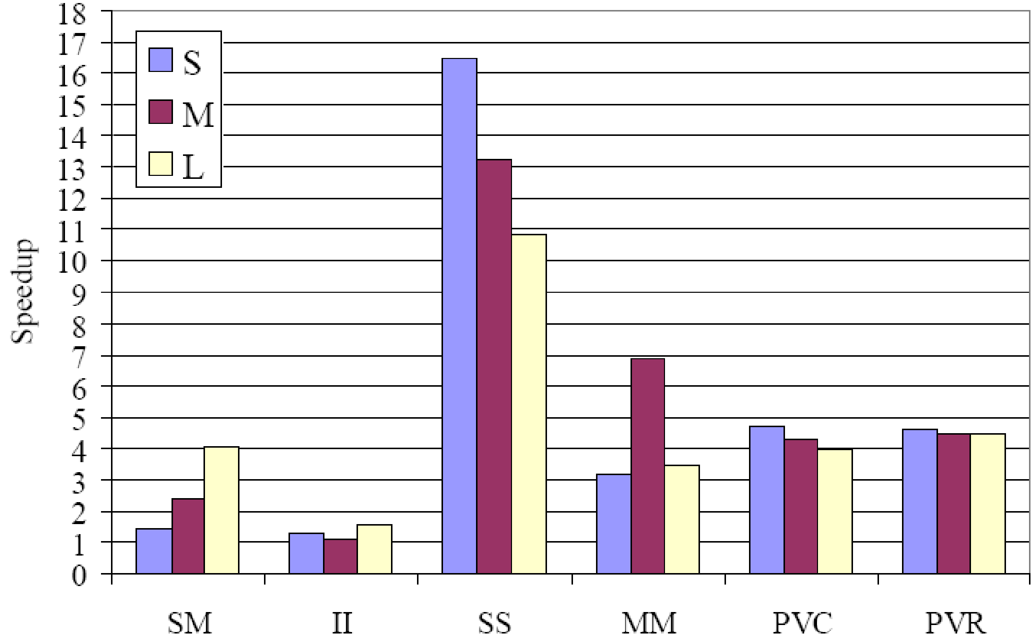
- Optimizations
  - Two-step memory access scheme to deal with memory management issue
    - Steps
      - Determine size of output for each thread
      - Compute prefix sum of output sizes
    - Results in fixed size allocation of correct size and allows each thread to write to pre-determined location without conflict

System Issues

- Optimizations (continued)
  - Hashing (of keys)
    - Minimizes more costly comparison of full key value
  - Coalesced accesses
    - Access by different threads to consecutive memory address are combined into one operation
    - Keys/values for threads are arranged in adjacent memory locations to exploit coalescing
  - Built in vector types
    - Data may consist of multiple items of same type
    - For certain types (`char4`, `int4`) entire vector can be read as a single operations

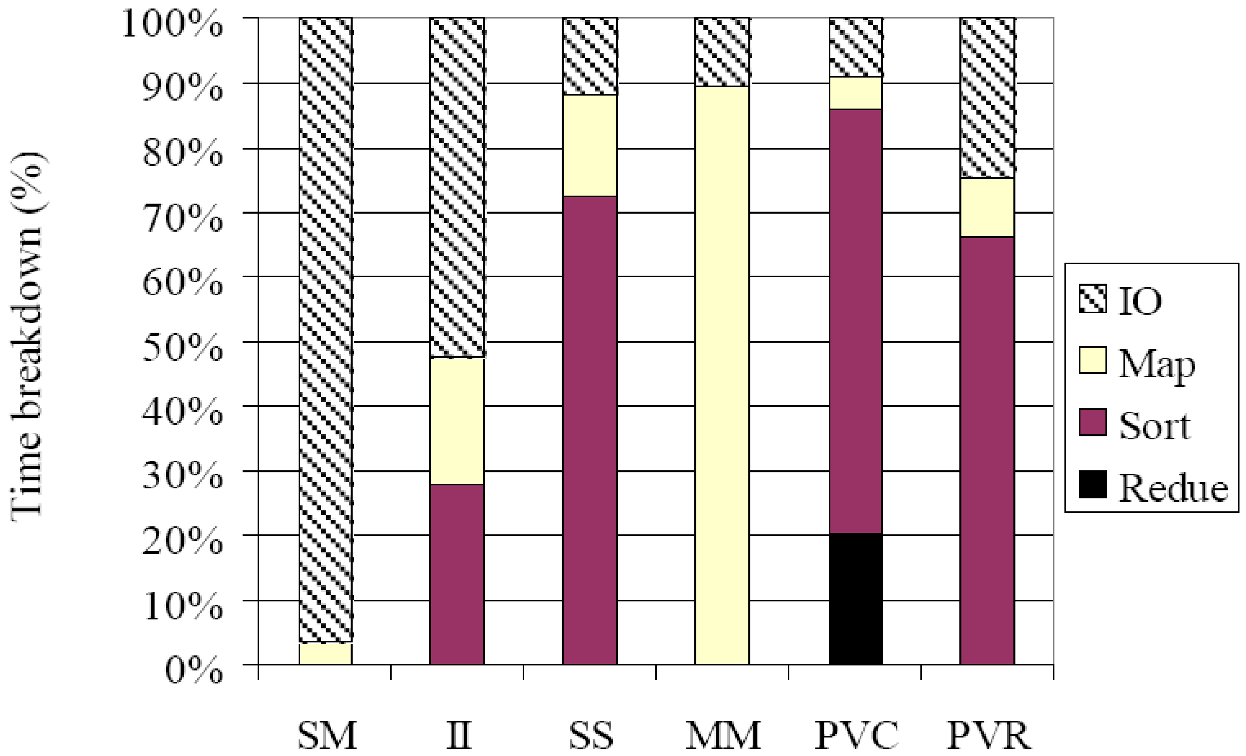
Mars Speedup

- Compared to Phoenix



- Optimizations
  - Hashing (1.4-4.1X)
  - Coalesced accesses (1.2-2.1X)
  - Built-in vector types (1.1-2.1X)

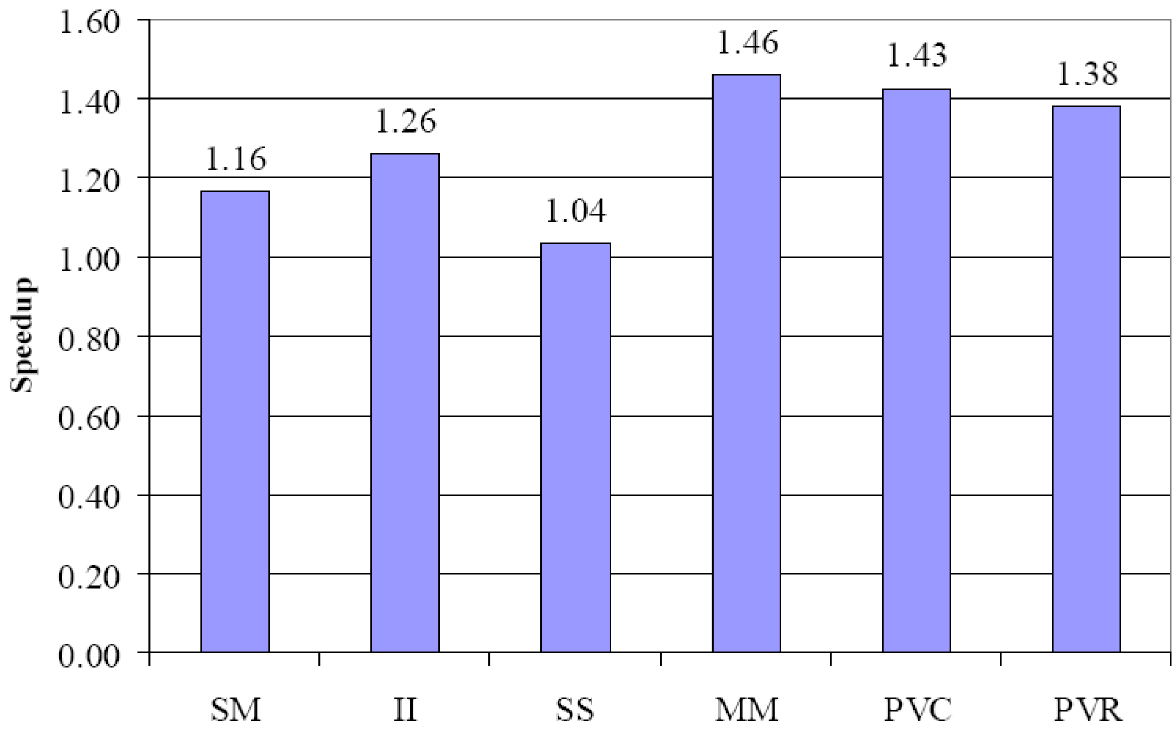
Execution time distribution



- Significant execution time in infrastructure operations
  - IO
  - Sort

Co-processing

- Co-processing (speed-up vs. GPU only)
  - CPU – Phoenix
  - GPU - Mars



Overall Conclusion

- MapReduce is an effective programming model for a class of data-intensive applications
- MapReduce is not appropriate for some applications
- MapReduce can be effectively implemented on a variety of platforms
  - Cluster
  - CMP/SMP
  - GPGPU