Evaluating MapReduce for Multi-core and Multiprocessor Systems

Colby Ranger, Ramanan Raghuraman, Arun Penmetsa,
Gary Bradski, Christos Kozyrakis

Computer Systems Laboratory
Stanford University

Presentation: Fabian Gremper

Structure

- Introduction
  - MapReduce
- Phoenix
- Evaluation
- Conclusion

Introduction

MapReduce

Introduction
What is MapReduce?

- MapReduce
  - Allows programmers to write functional style code that is automatcally parallelized and scheduled in a distributed system
  - Practical. Avoid having to
    - manage concurrency manually (threads/locks)
    - deal with data locality
  - Portable

Introduction
What does MapReduce do for you?

- MapReduce
  - Specify concurrency and locality at a high level
  - Efficient runtime system handles low-level mapping, resource management and fault management

Introduction
How does MapReduce work?

- Map
  - Input: ???
  - Output: Intermediate <key, value> pairs.
- Reduce
  - Input: Intermediate <key, value> pairs with the same key.
  - Output: Zero or more output pairs, sorted by their key.

Introduction
MapReduce Example

- Example: Word count

```
Map(void *input) {
  for each word w in input
    EmitIntermediate(w, 1);
}

Reduce(String key, Iterator values) {
  int result = 0;
  for each v in values
    result += v;
  Emit(w, result);
}
```

Introduction
Why MapReduce is awesome

- Why?
  - Simplicity
  - Programmer focuses on functionality
  - Model provides enough high-level information for parallelization
  - Pretty widely applicable

Phoenix

(Stanford University)

Phoenix

- Implementation of MapReduce
  - Multi-core and multi-processor systems
  - Shared memory
- Includes a programming API
- Run-time system
  - Thread creation
  - Dynamic task scheduling
  - Fault tolerance across processor nodes

Phoenix

Functions provided by the runtime

```
int phoenix_scheduler(scheduler_args_t *args)
```

- Initializes the runtime system.
- scheduler_args_t provides the needed functions and data pointers.

```
void emit_intermediate(void *key, void *val, int key_size)
```

- Used in Map to emite intermediate output.

```
void emit(void *key, void *val)
```

- Used in Reduce to emit a final output pair.

Phoenix

Functions defined by the user

`int (*splitter_t)(void *, int, map_args_t *)`

- Splits input across Map tasks.
  - Arguments: input data pointer, unit size for each task, input buffer pointer for each Map task.

`void (*map_t)(map_args_t *)`

- The map function. Each Map tasks executes this function on its input.

`int (*partition_t)(int, void *, int)`

- Partitions intermediate pairs for Reduce task.
  - Arguments: number of Reduce tasks, a pointer to the keys, and a size of the key. Default partitioning is based on the key hashing.

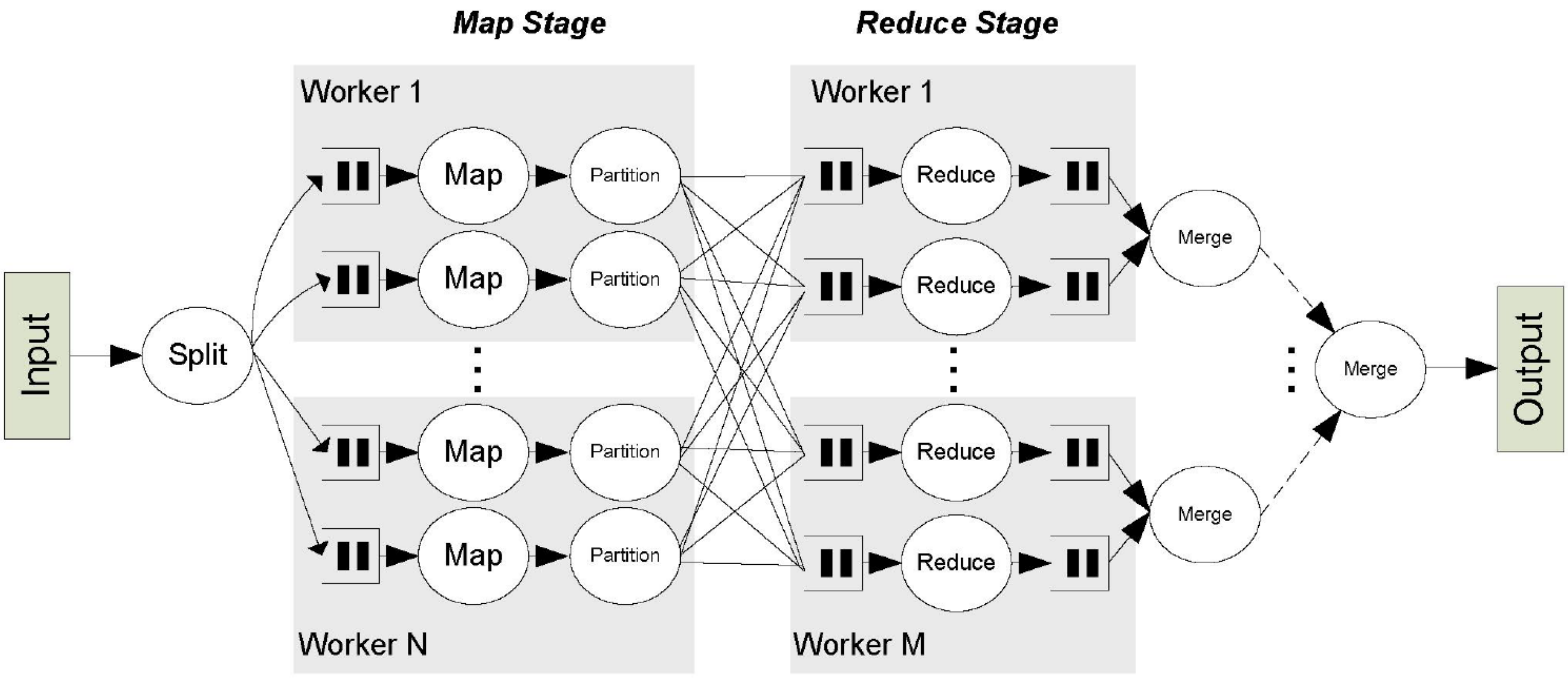
`void (*reduce_t)(void *, void **, int)`

- Reduce function. Each reduce task executes this on its input.
  - Arguments: pointer to a key, a pointer to the associated values, value count. Default is the identity function.

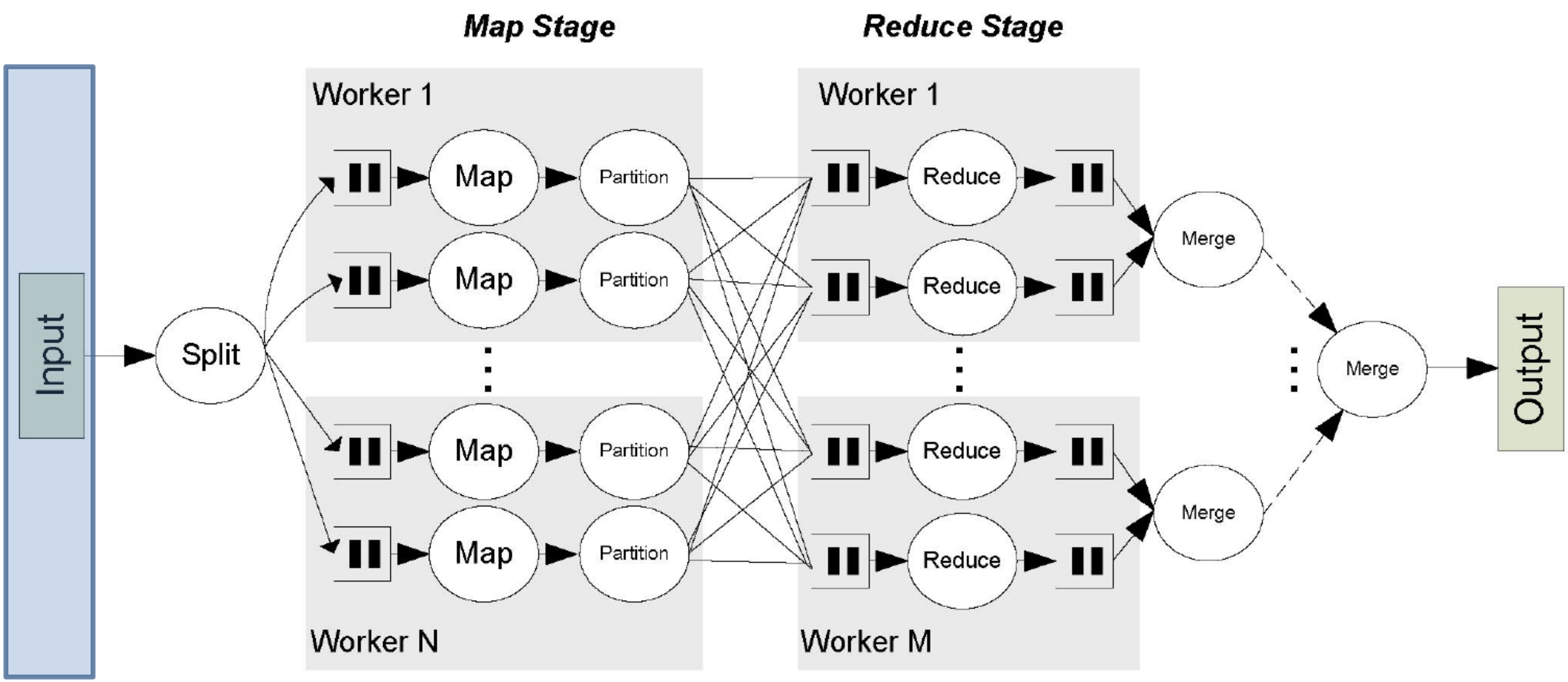
`int (*key_cmp_t)(const void *, const void *)`

- Compare function.

Phoenix
Data flow

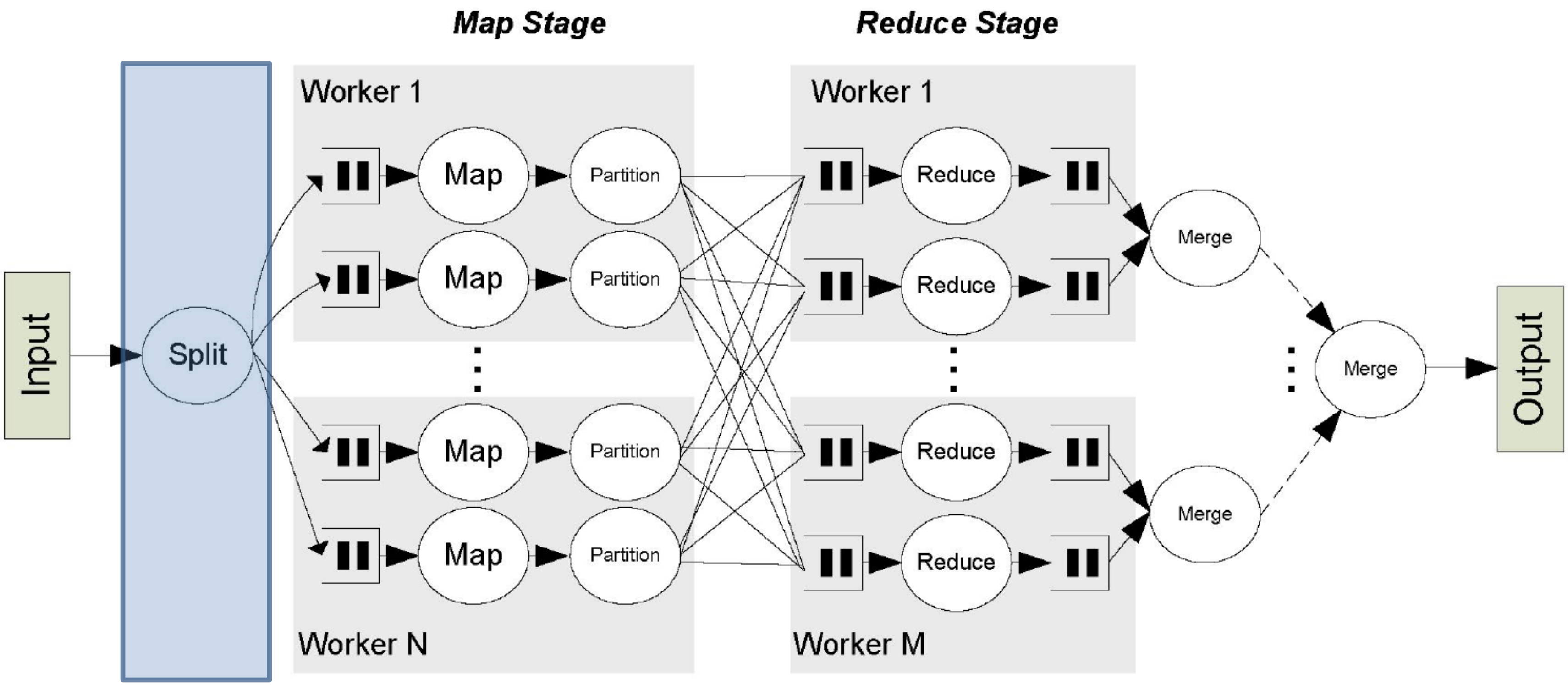


Phoenix
Data flow



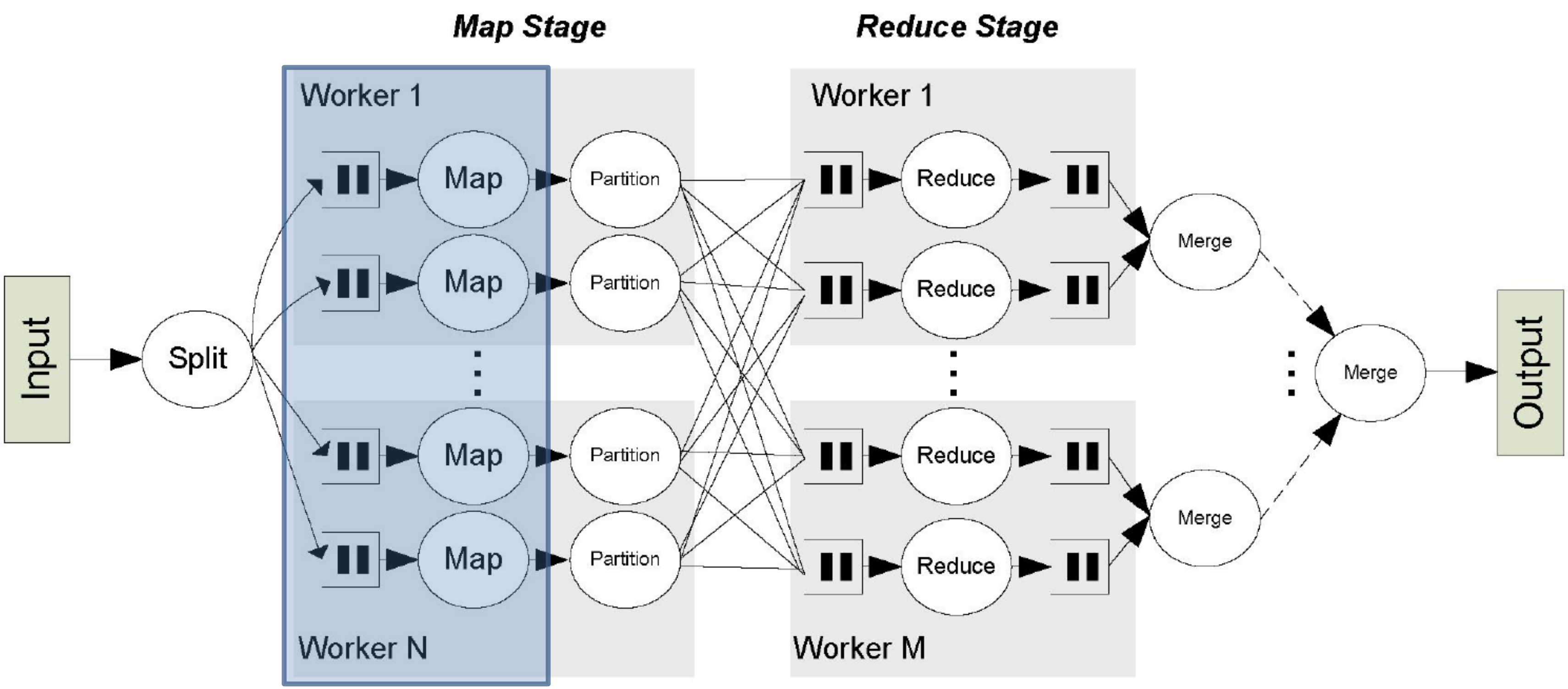
- Scheduler determines the number of cores to use for this computation.
- Spawns a worker thread for each core.
- Map and Reduce tasks are then later dynamically assigned.

Phoenix
Data flow



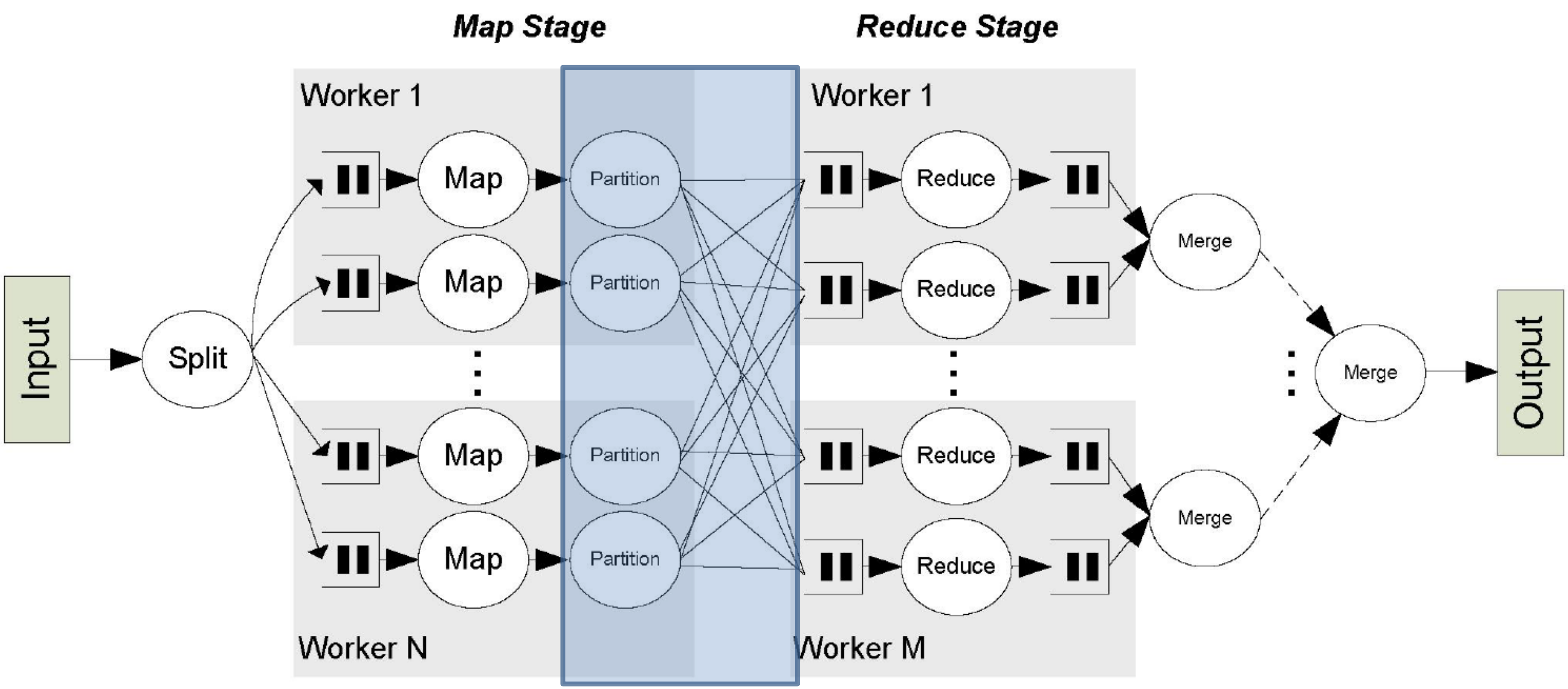
- Scheduler uses the Splitter to divide input pairs into equally sized units.

Phoenix
Data flow



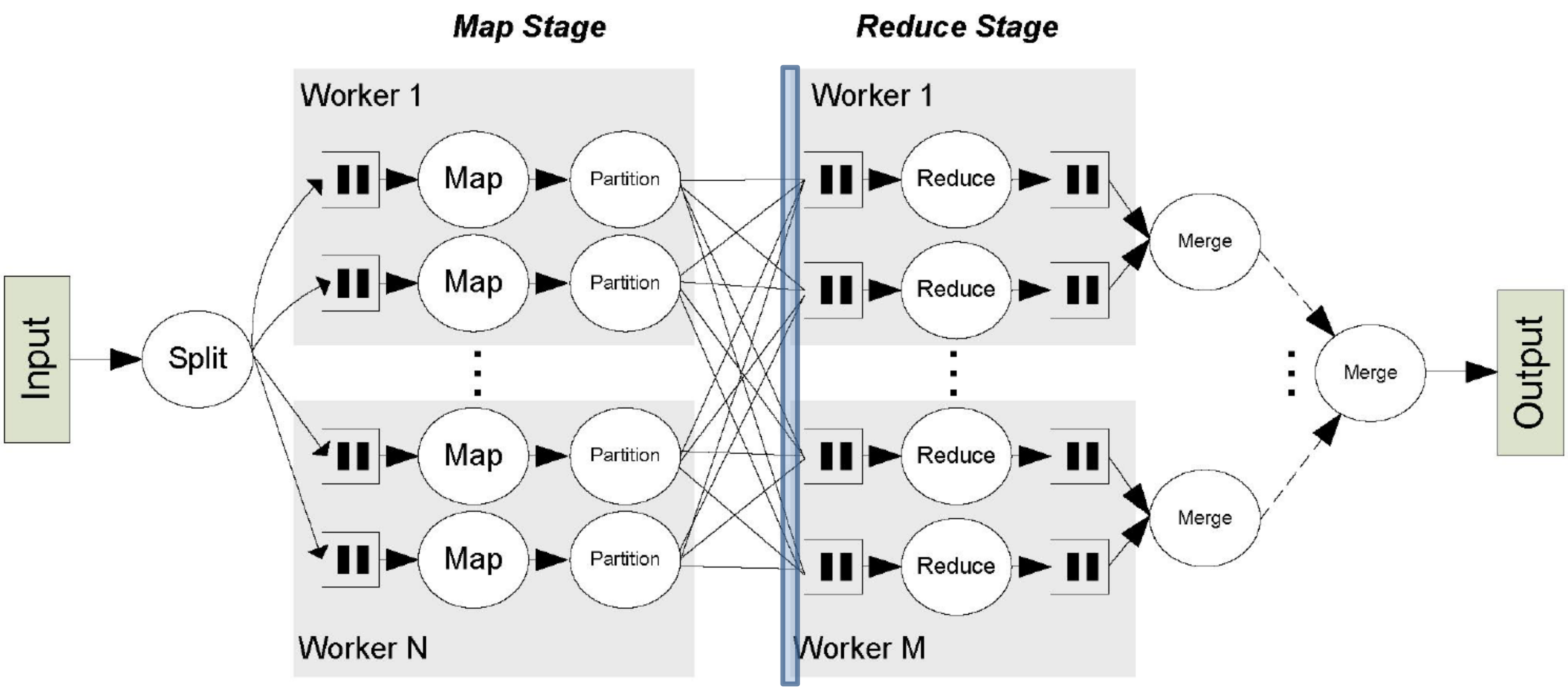
- Map tasks are assigned dynamically to workers.
- Intermediate <key, value> pairs.

Phoenix
Data flow



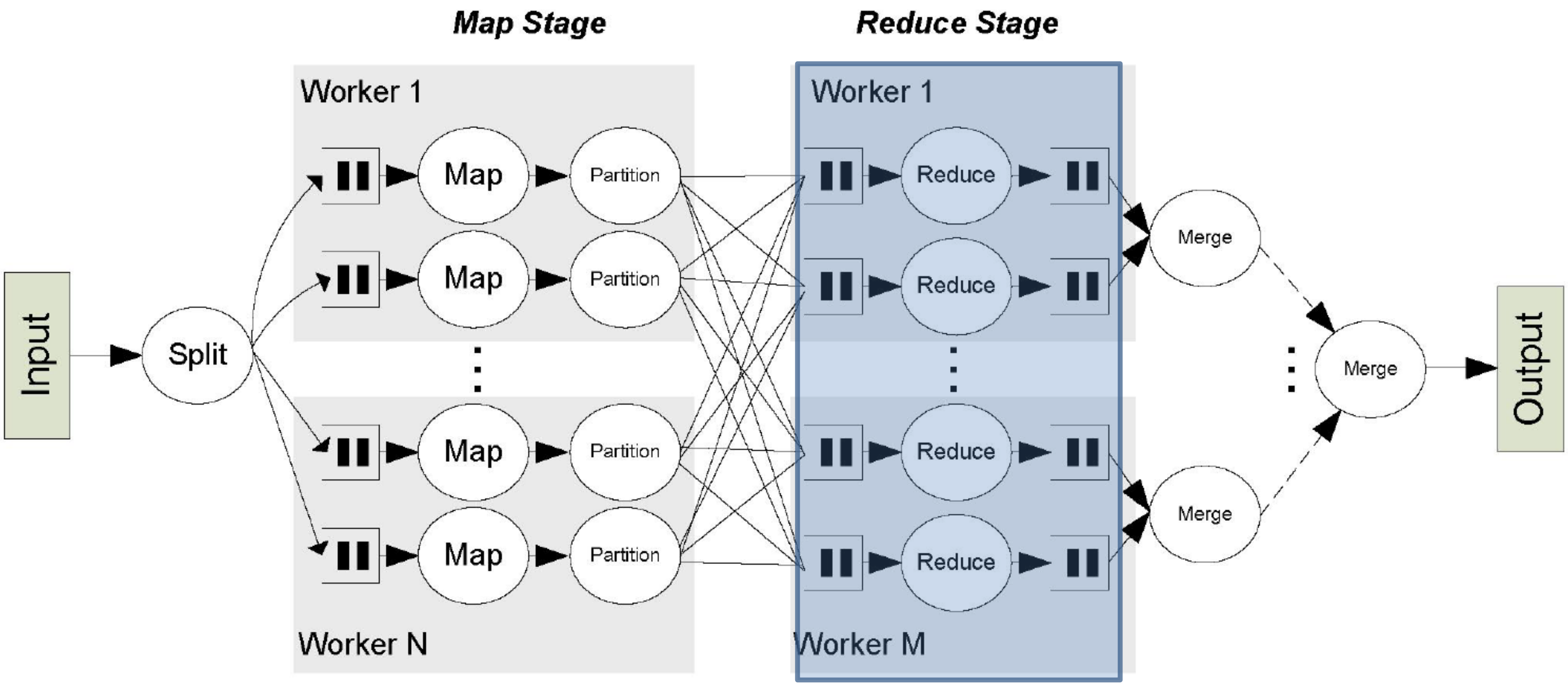
- Splits <key, value> into units for the Reduce tasks.
- Ensures all values of the same key go to the same unit.

Phoenix
Data flow



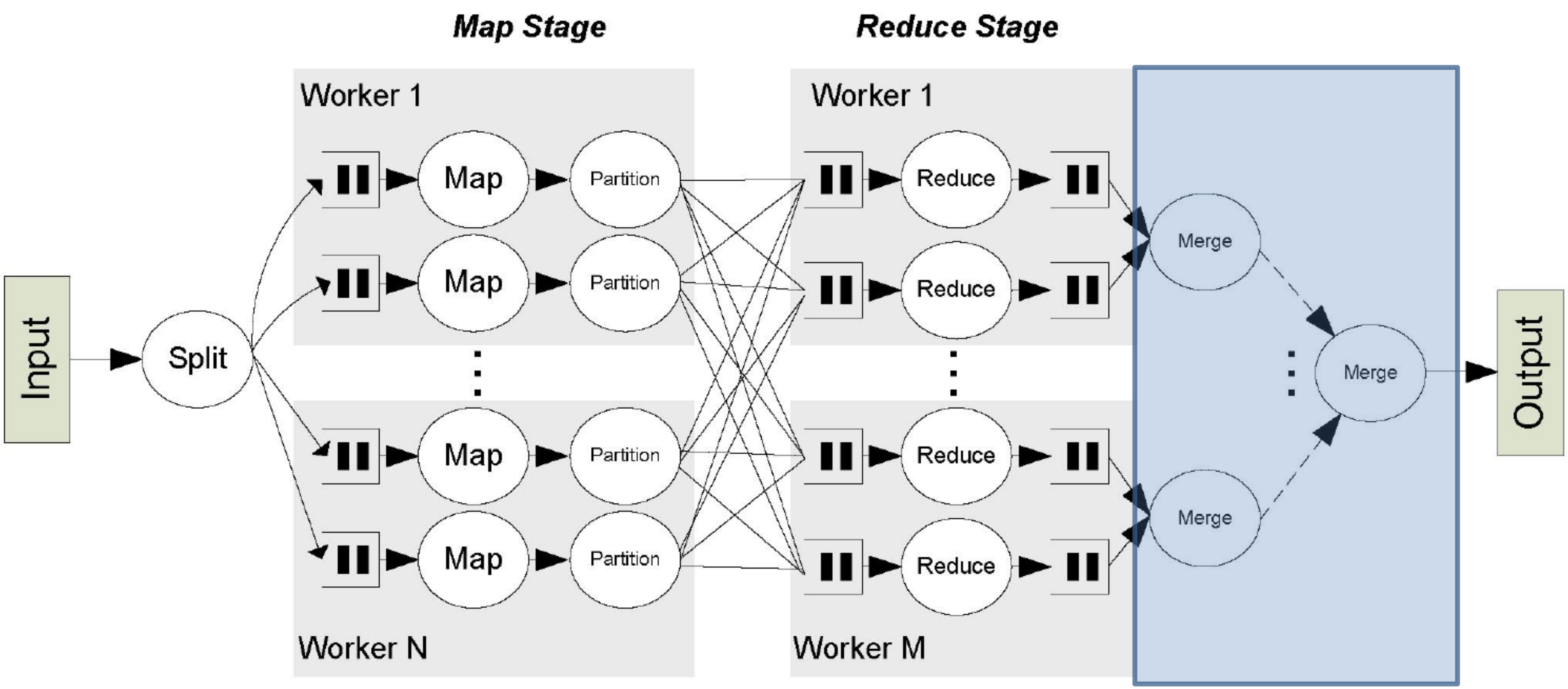
- Wait until Map stage finished completely.

Phoenix
Data flow



- Reduce tasks dynamically assigned to workers.
- Possibly higher imbalance. (Same key → same worker)

Phoenix
Data flow



- Merge into a single buffer.
- Takes log(P/2) steps.
- Ordered.

Phoenix
Buffer management

- Buffers allocated in shared memory.
  - Accessed in a well specified way by a few functions.
  - Intermediate buffers not visible to user code.
- Intermediate buffers are used to store intemediate output pairs.
  - Each worker has its own set of buffers.
  - Dynamically resized.

Phoenix
Fault recovery

- Limited fault detection, focuses on recovery
- Detect faults through timeouts
  - Re-execute the failed task
  - Assume transient error
- Repeated errors → assume permanent error.
  - Do not use this worker anymore.
- Corruption of the shared memory?
- No fault recovery for the scheduler itself.

Evaluation

of Phoenix

Evaluation

- Shared memory systems

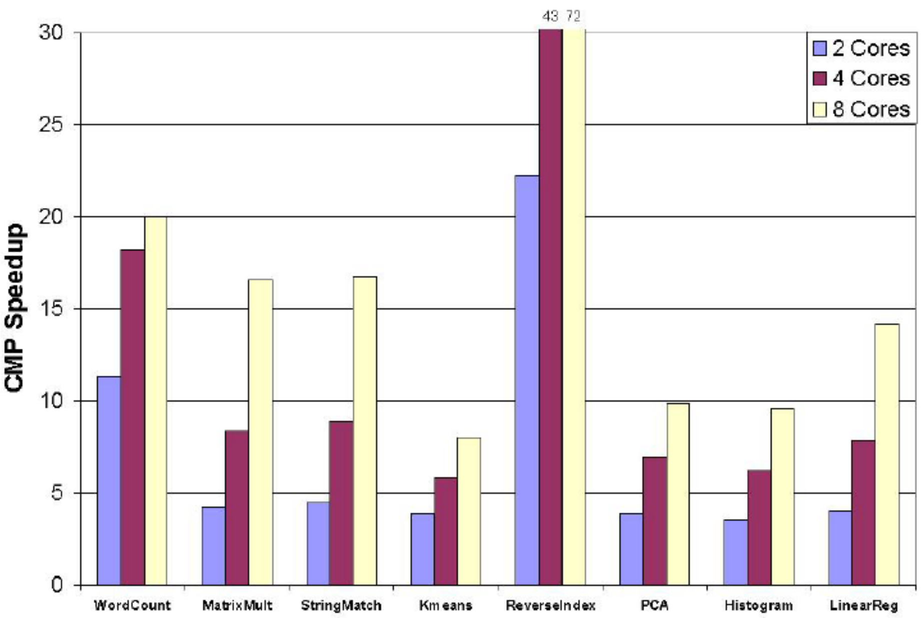
	CMP	SMP
Model	Sun Fire T1200	Sun Ultra-Enterprise 6000
CPU Type	UltraSparc T1 single-issue in-order	UltraSparc II 4-way issue in-order
CPU Count	8	24
Threads/CPU	4	1
L1 Cache	8KB 4-way SA	16KB DM
L2 Size	3MB 12-way SA shared	512KB per CPU (off chip)
Clock Freq.	1.2 GHz	250 MHz

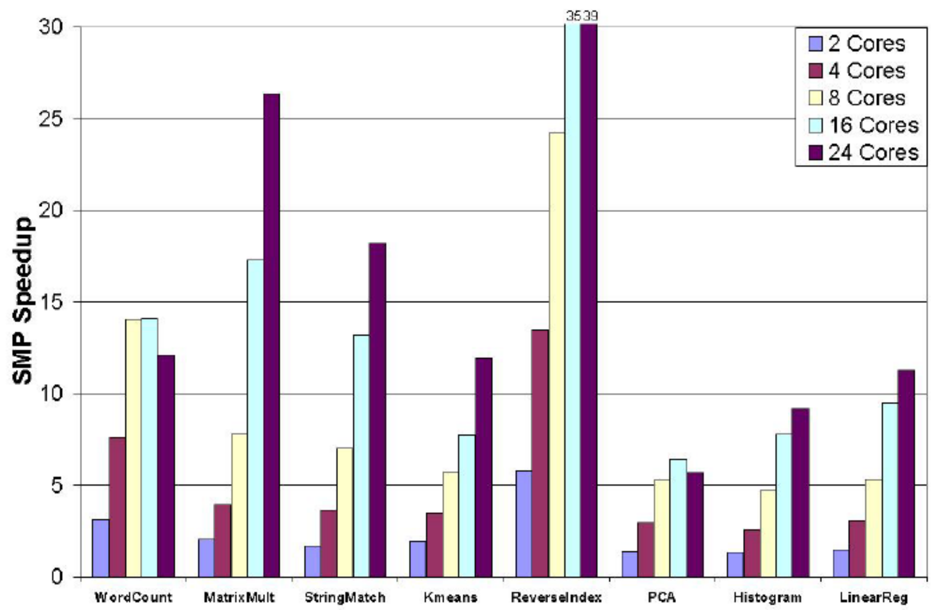
  - The same program should run as efficiently as possible on any type of shared-memory system.
  - Without any involvement of the user.

Evaluation
Applications

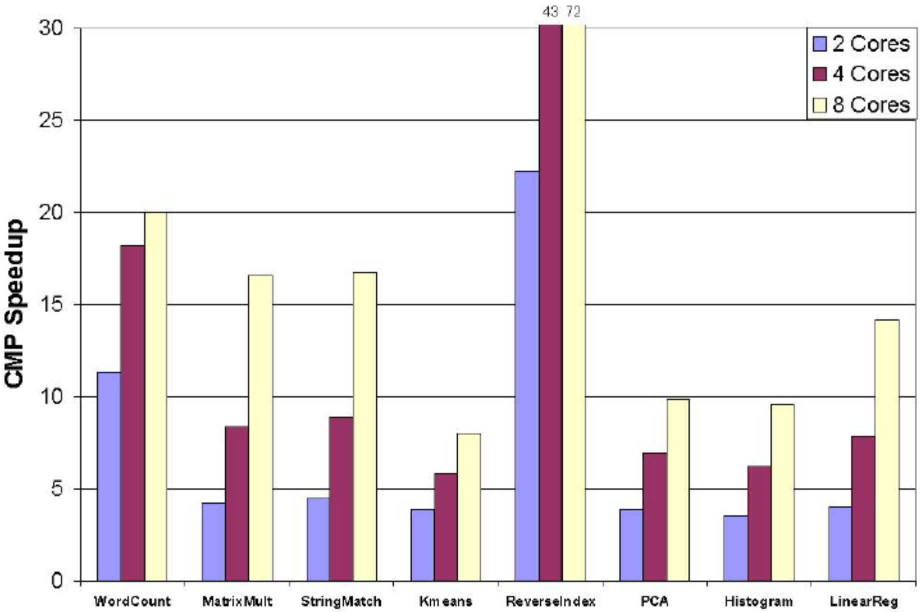
	Description	Data Sets	Code Size Ratio	
			Pthreads	**Phoenix**
Word Count	Determine frequency of words in a file	S:10MB, M:50MB, L:100MB	1.8	0.9
Matrix Multiply	Dense integer matrix multiplication	S:100x100, M:500x500, L:1000x1000	1.8	2.2
Reverse Index	Build reverse index for links in HTML files	S:100MB, 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

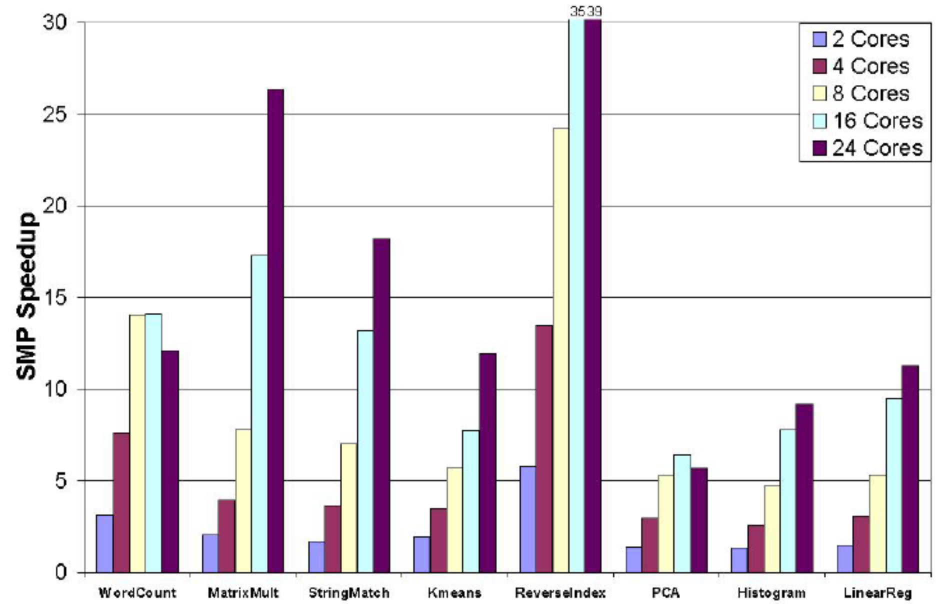
Evaluation
Speedup for different # of processors





Evaluation
Speedup for different # of processors

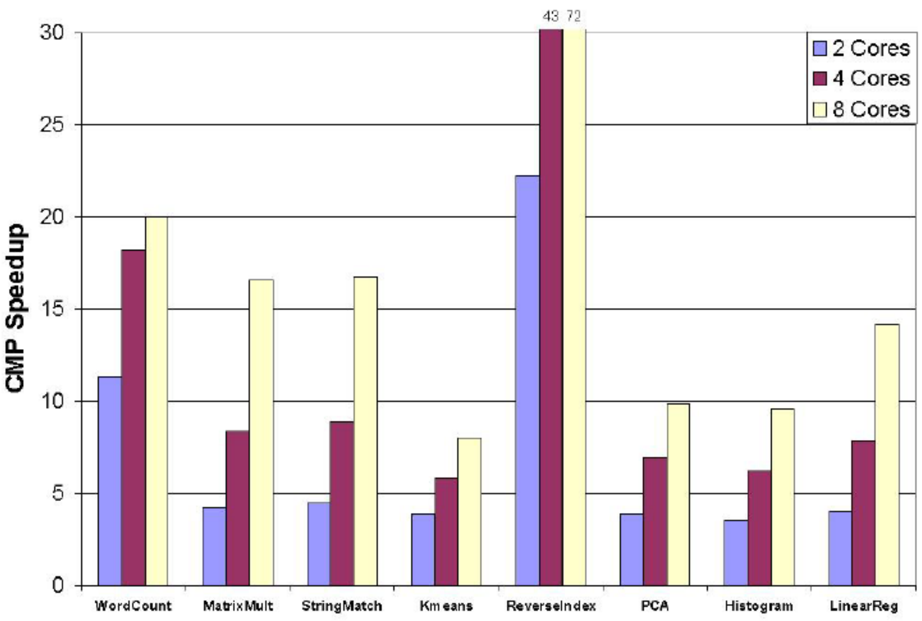


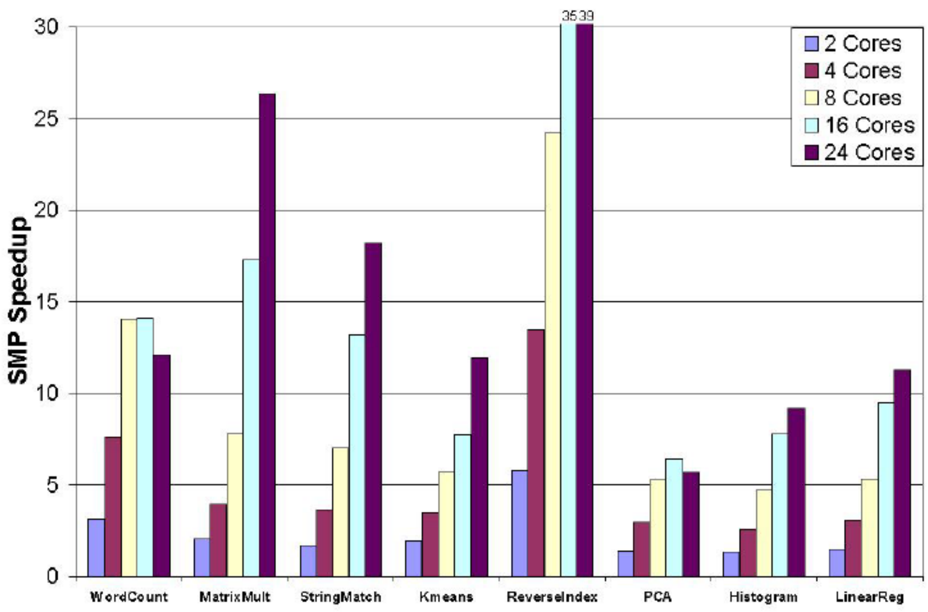


- WordCount, MatrixMultiply, StringMatch, LinearRegression
  key-based structure → significant speedups
- Kmeans, PCA, Histogram
  significant overheads due to unnatrual key-based structure

Evaluation

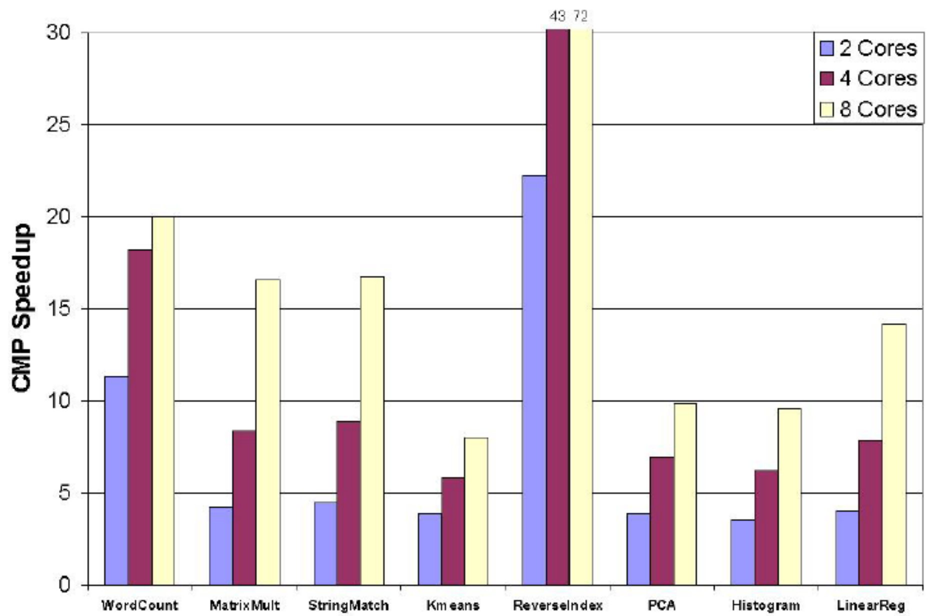
Speedup for different # of processors

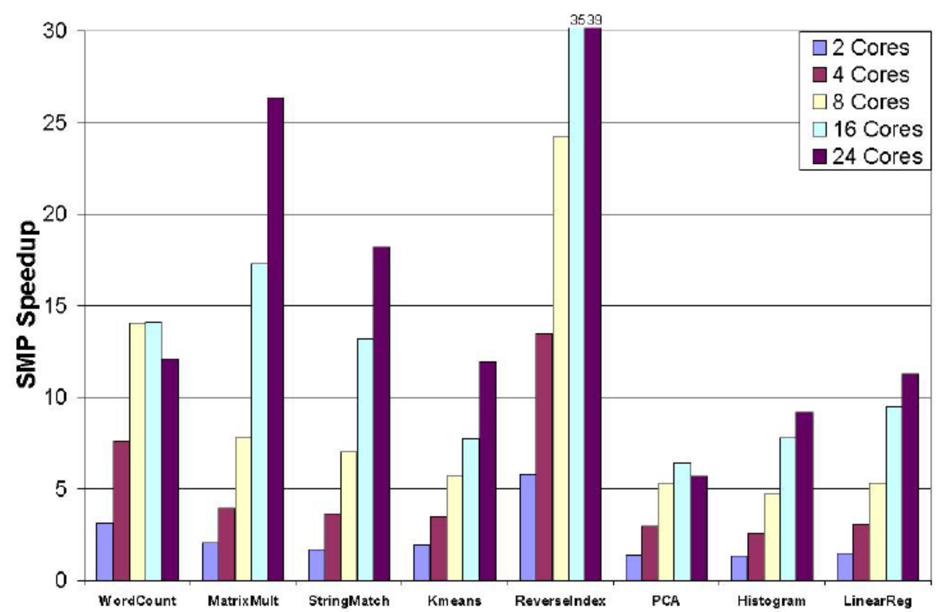




- ReverseIndex
  - Heaps become increasingly smaller over time
  - Reduced merging overhead due to additional cores and caching

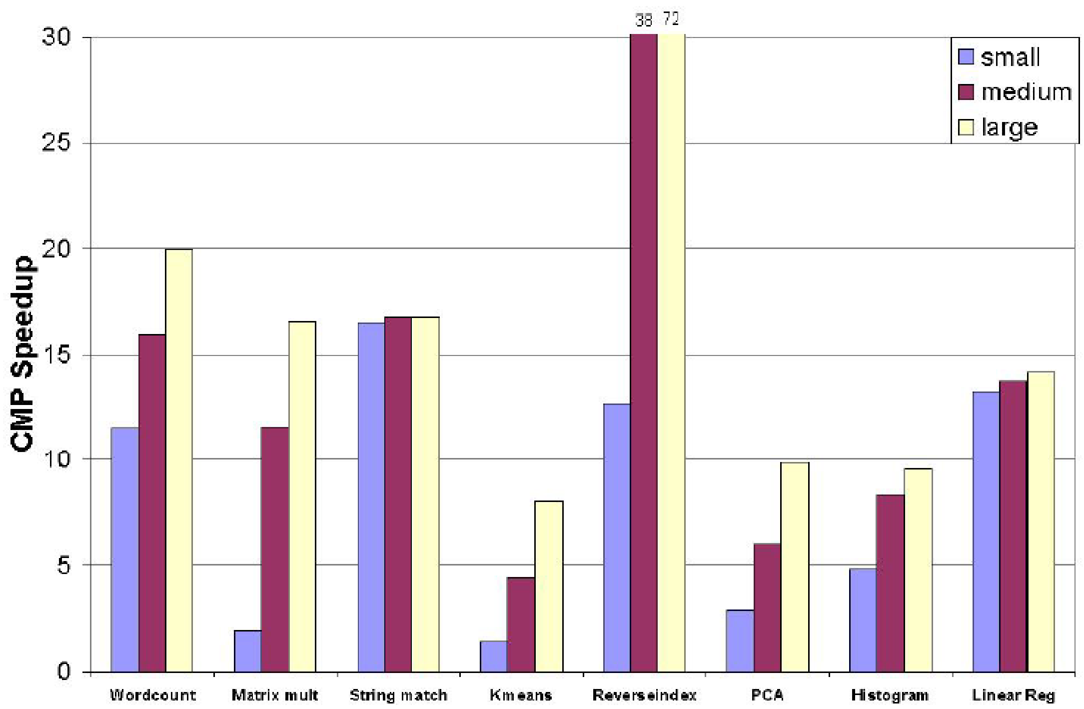
Evaluation
Speedup for different # of processors



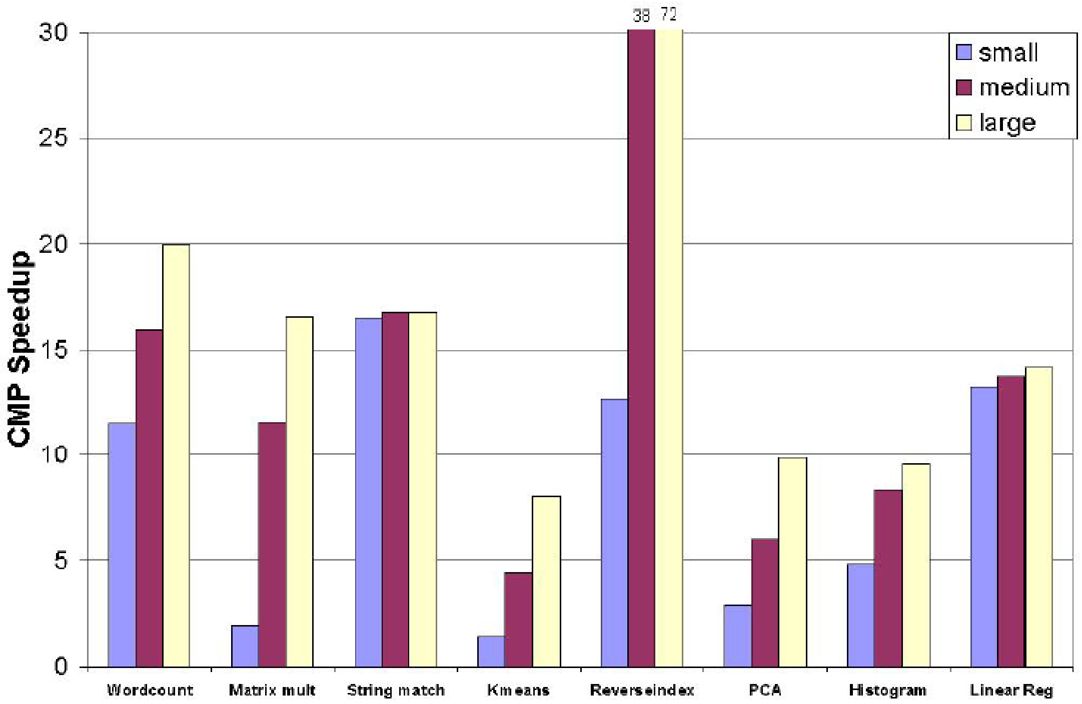


- MatrixMultiply
  - Caching effects (beneficial sharing in the CMP, increased cache capacity in the SMP with more cores)

Evaluation
Speedup for different dataset sizes

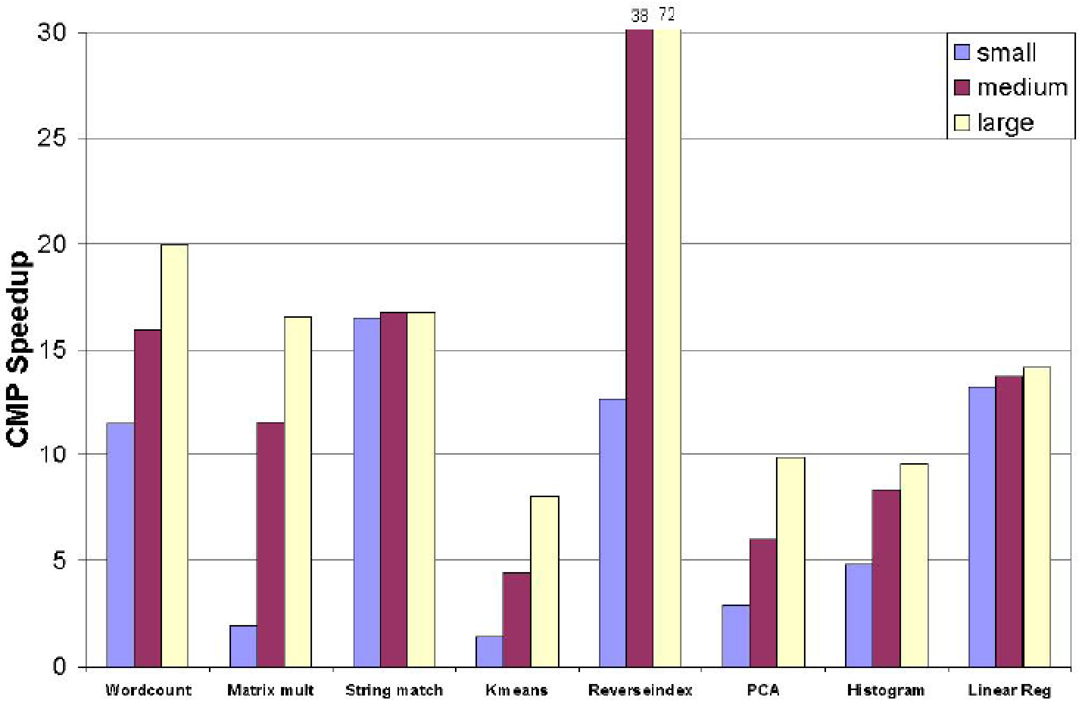


Evaluation
Speedup for different dataset sizes



- Larger data sets allow the phoenix runtime to better armotize its overheads for task management, buffer allocation and sorting.
- Caching effects are more significant.
- Load imbalance is more rare

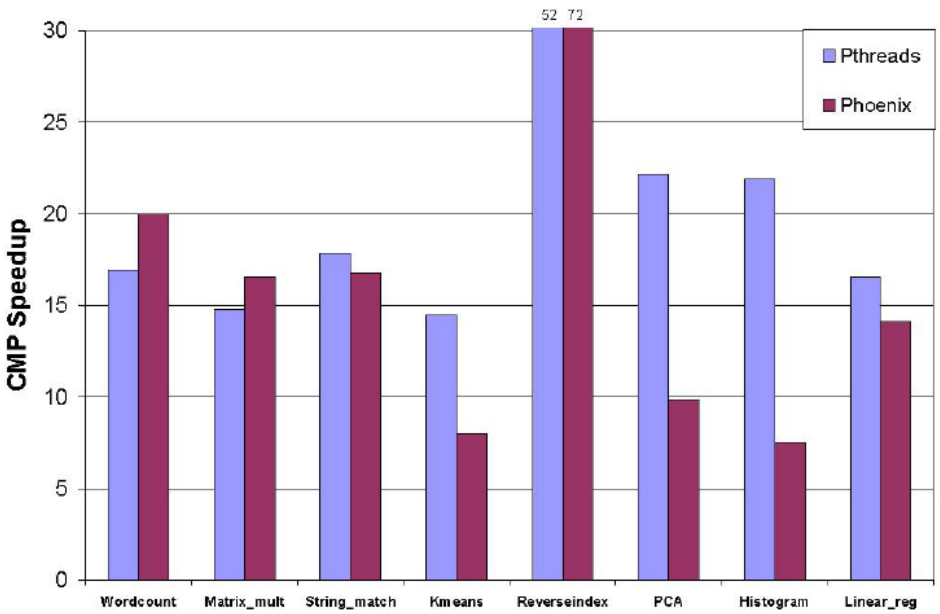
Evaluation
Speedup for different dataset sizes

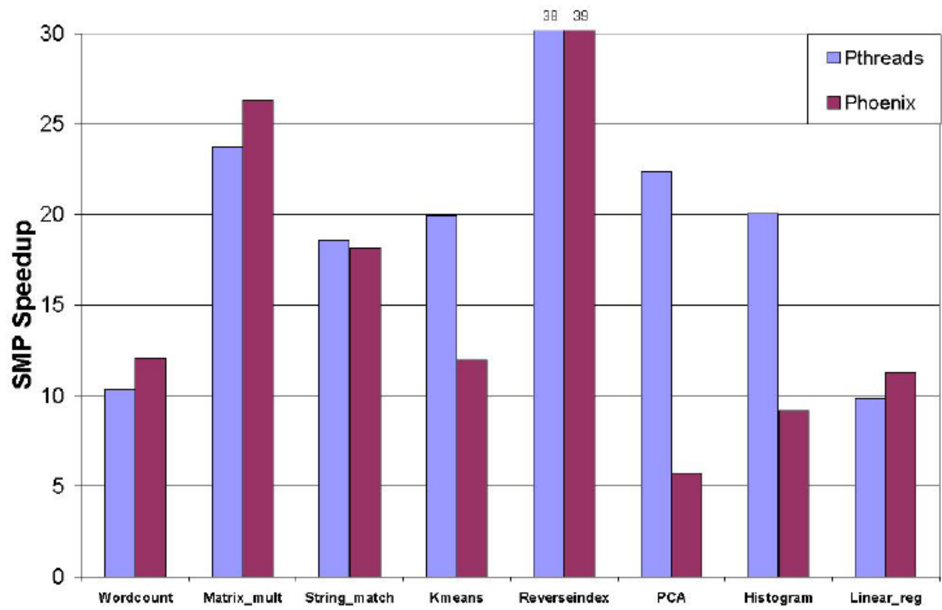


- StringMatch, LinearRegression
  - Even their small sets contain a large number of elements
  - Significant amount of per-element computation in their dataset

Evaluation

Speedup vs. P-threads





Evaluation
Fault injection

- Fault injection experiment
  - Failure affects the execution and buffers for the tasks, but does not corrupt the runtime or its data structures
  - Runtime increases by
    - 9-14% for 1 permanent fault
      (mostly depending on at which point the fault occured)
    - <0.5% for 1-2 transient faults

Conclusion

Conclusion

- Goal: Evaluating MapReduce for shared-memory systems.
  - Given an efficient implementation, MapReduce is an attractive model for some classes of computation.
  - Leads to good parallel efficiency with simple code
    - Dynamically managed without any programmer effort
  - MapReduce performs subobtimally for applications that are difficult to express with its model anyway...