Evaluating MapReduce for Multi-core and Multiprocessor Systems

Shiran Dudy
20/04/14

Outline

- Introduction
- Overview of MapReduce
- Shared-memory implementation
- Evaluation methodology
- Evaluation results
- Conclusions
- Discussion with regard to additional work

Overview of MapReduce

Programming Model

```
class Mapper
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            Emit(term t, count 1)
```

```
class Reducer
    method Reduce(term t, counts [c1, c2, ...])
        sum ← 0
        for all count c ∈ counts [c1, c2, ...] do
            sum ← sum + c
        Emit(term t, count sum)
```

Data-Intensive Text Processing with MapReduce, Jimmy Lin and Chris Dyer, 2010

Overview of MapReduce

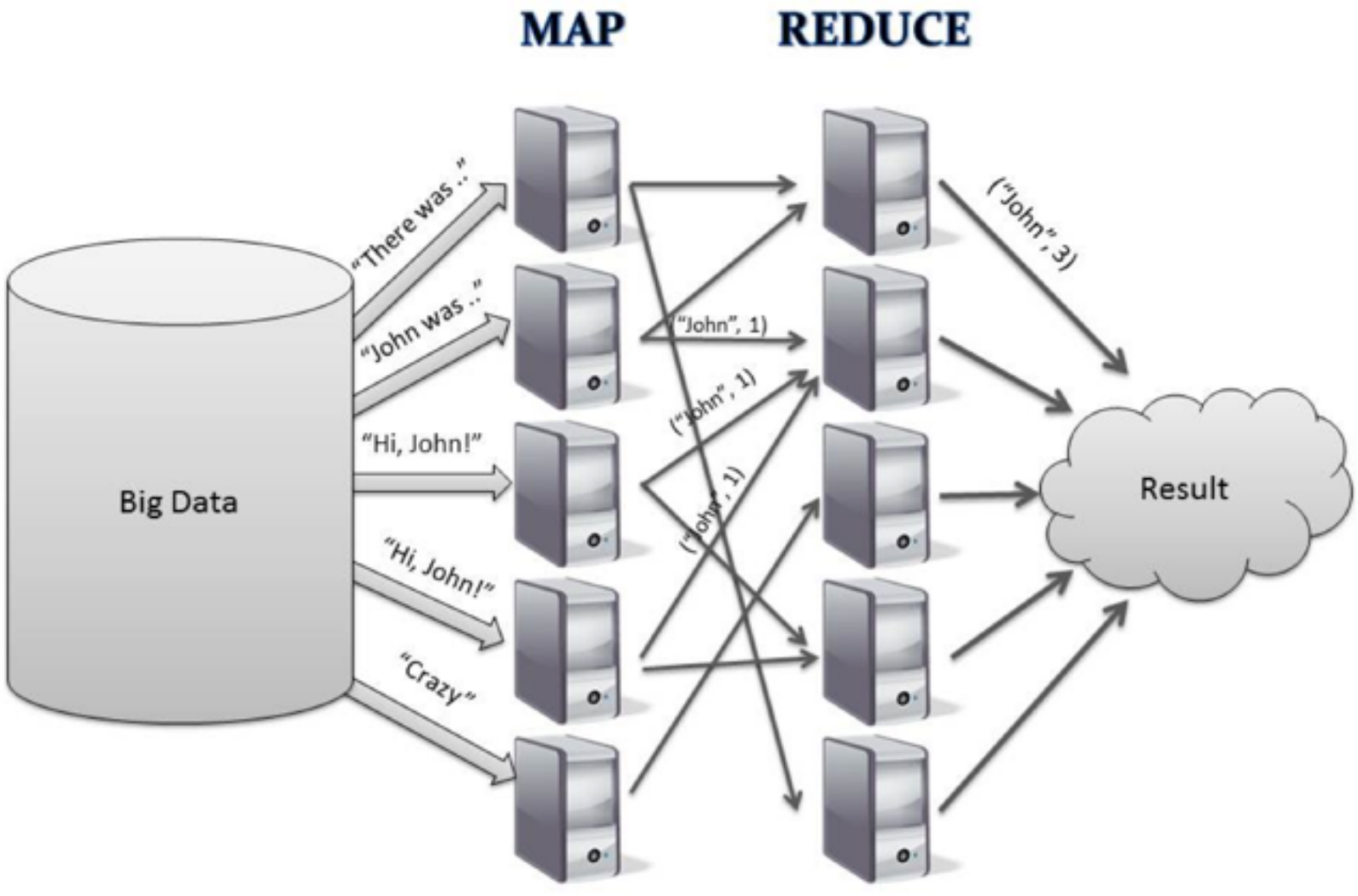
Programming Model

```
class Mapper
    method Initialize
        H ← new AssociativeArray
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            H{t} ← H{t} + 1
    method Close
        for all term t ∈ H do
            Emit(term t, count H{t})
```

Data-Intensive Text Processing with MapReduce, Jimmy Lin and Chris Dyer, 2010

Overview of MapReduce

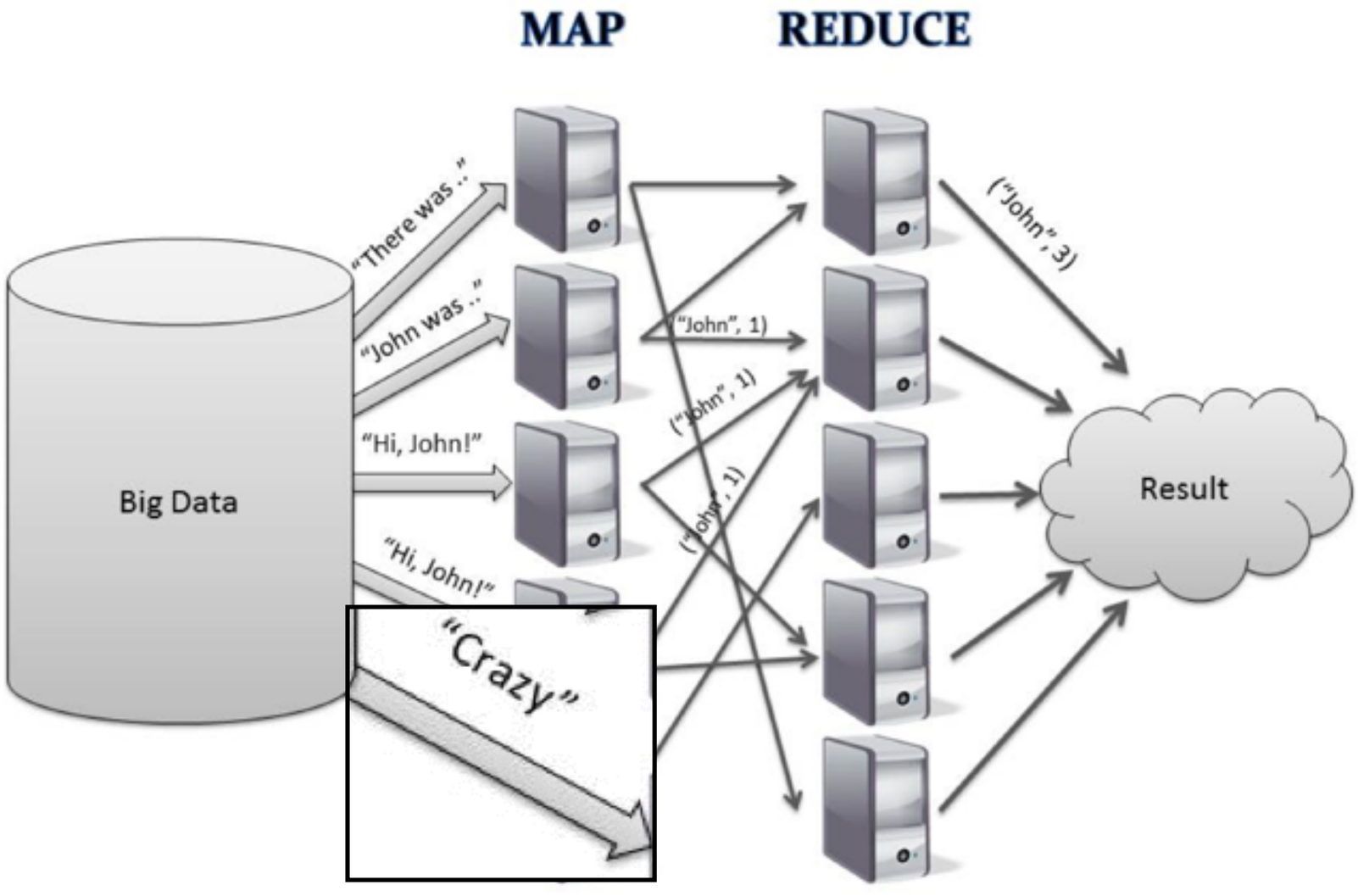
Runtime System



http://dme.rwth-aachen.de/de/research/projects/mapreduce

Overview of MapReduce

Runtime System

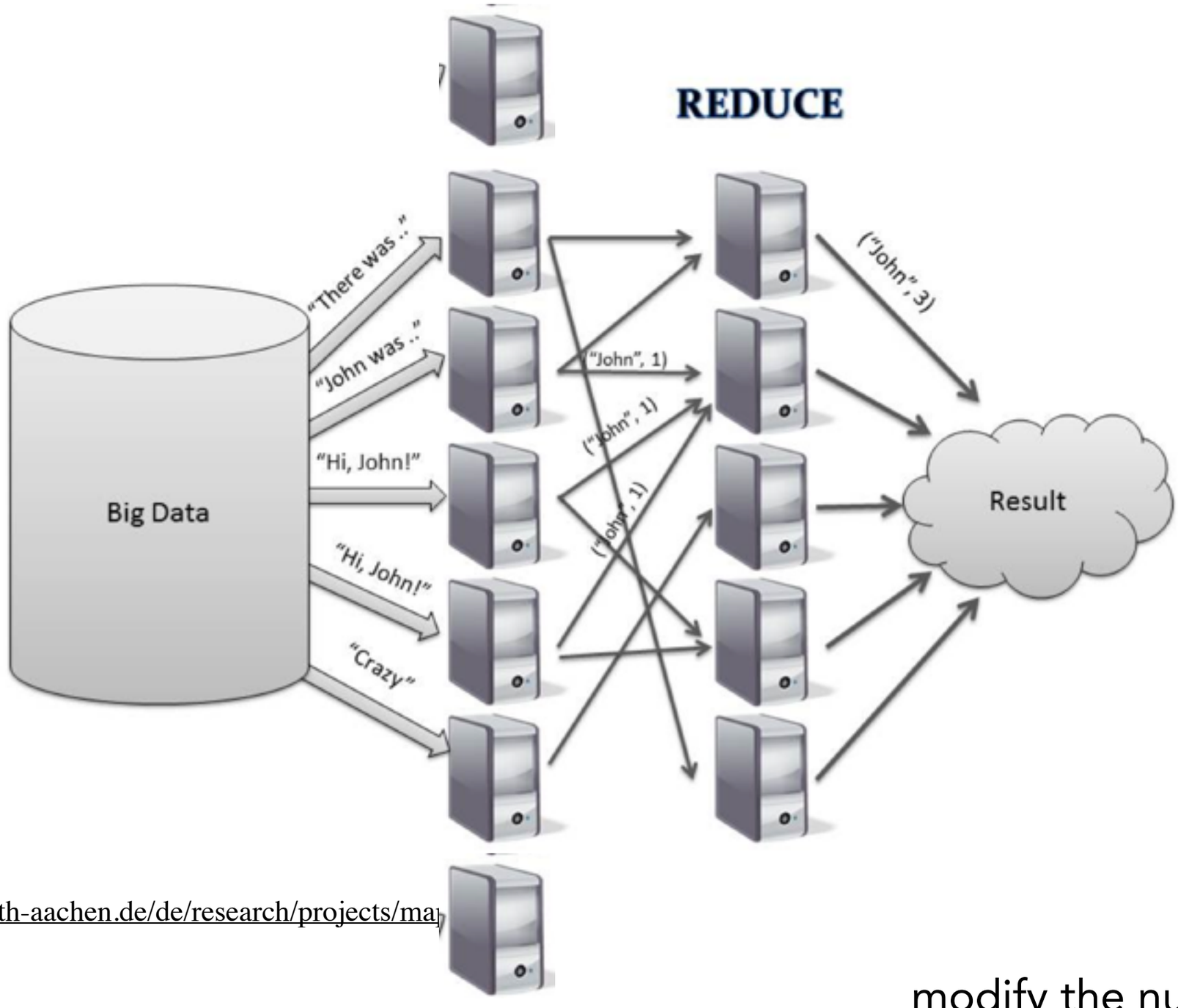


http://dme.rwth-aachen.de/de/research/projects/mapreduce

modify the size of the unit

Overview of MapReduce

Runtime System

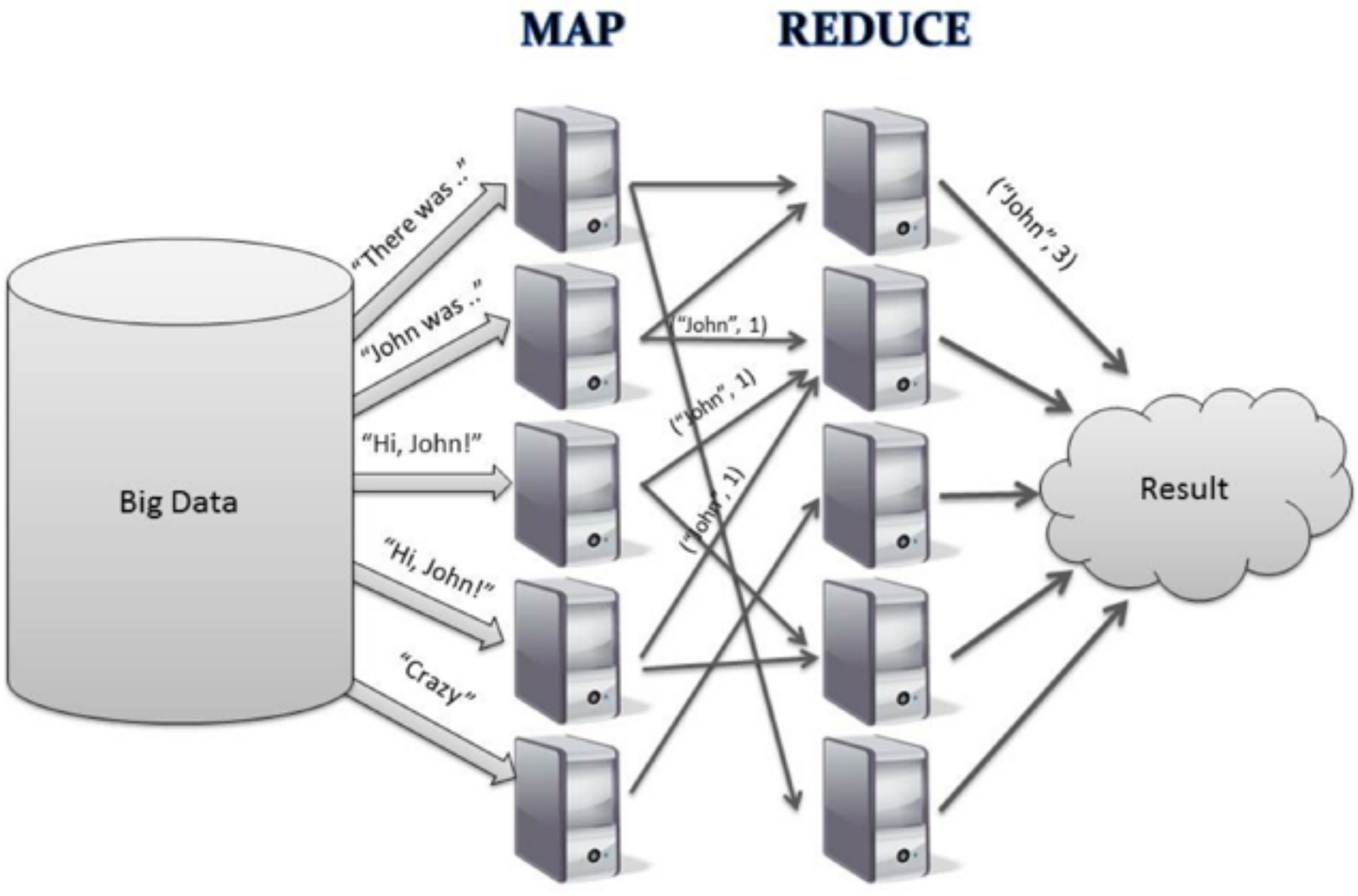


http://dme.rwth-aachen.de/de/research/projects/ma

modify the number of nodes

Overview of MapReduce

Runtime System



http://dme.rwth-aachen.de/de/research/projects/mapreduce

assign to itself the next task while processing the current

Overview of MapReduce

Runtime System

Coping with failures

- re assign a Map/Reduce when detected a node has failed
- re execute the specific Map/Reduce tasks when there's a memory corruption
- dynamically adjust the number of nodes it uses due to a hostile environment (heat, power failure)

Overview of MapReduce

Runtime System

How big is the overhead?

The Shared Memory Implementation

The Phoenix System

API

Runtime

- Basic operation and control flow
- Buffer management
- Fault recovery
- Concurrency and locality management

The Shared Memory Implementation

The Phoenix System

API

Runtime

- Basic operation and control flow
- Buffer management
- Fault recovery
- Concurrency and locality management

The Shared Memory Implementation

The Phoenix System - API

Function Description	R/O
Functions Provided by Runtime	
`int phoenix_scheduler (scheduler_args_t * args)` Initializes the runtime system. The `scheduler_args_t` struct provides the needed function & data pointers	R
`void emit_intermediate(void *key, void *val, int key_size)` Used in Map to emit an intermediate output <key, value> pair. Required if the Reduce is defined	O
`void emit(void *key, void *val)` Used in Reduce to emit a final output pair	O
Functions Defined by User	
`int (*splitter_t)(void *, int, map_args_t *)` Splits the input data across Map tasks. The arguments are the input data pointer, the unit size for each task, and the input buffer pointer for each Map task	R
`void (*map_t)(map_args_t*)` The Map function. Each Map task executes this function on its input	R
`int (*partition_t)(int, void *, int)` Partitions intermediate pair for Reduce tasks based on their keys. The arguments are the number of Reduce tasks, a pointer to the keys, and a the size of the key. Phoenix provides a default partitioning function based on key hashing	O
`void (*reduce_t)(void *, void **, int)` The Reduce function. Each reduce task executes this on its input. The arguments are a pointer to a key, a pointer to the associated values, and value count. If not specified, Phoenix uses a default *identity* function	O
`int (*key_cmp_t)(const void *, const void*)` Function that compares two keys	R

Evaluating MapReduce for Multi-core and Multiprocessor Systems, Colby Ranger @ l Computer Systems Laboratory, Stanford University

The Shared Memory Implementation

The Phoenix System - API

Function Description	R/O
Functions Provided by Runtime	
`int phoenix_scheduler (scheduler_args_t * args)` Initializes the runtime system. The `scheduler_args_t` struct provides the needed function & data pointers	R
`void emit_intermediate(void *key, void *val, int key_size)` Used in Map to emit an intermediate output <key, value> pair. Required if the Reduce is defined	O
`void emit(void *key, void *val)` Used in Reduce to emit a final output pair	O
Functions Defined by User	
`int (*splitter_t)(void *, int, map_args_t *)` Splits the input data across Map tasks. The arguments are the input data pointer, the unit size for each task, and the input buffer pointer for each Map task	R
`void (*map_t)(map_args_t*)` The Map function. Each Map task executes this function on its input	R
`int (*partition_t)(int, void *, int)` Partitions intermediate pair for Reduce tasks based on their keys. The arguments are the number of Reduce tasks, a pointer to the keys, and a the size of the key. Phoenix provides a default partitioning function based on key hashing	O
`void (*reduce_t)(void *, void **, int)` The Reduce function. Each reduce task executes this on its input. The arguments are a pointer to a key, a pointer to the associated values, and value count. If not specified, Phoenix uses a default *identity* function	O
`int (*key_cmp_t)(const void *, const void*)` Function that compares two keys	R

Evaluating MapReduce for Multi-core and Multiprocessor Systems, Colby Ranger @ l Computer Systems Laboratory, Stanford University

The Shared Memory Implementation

The Phoenix System - API

Field	Description
Basic Fields	
`Input_data`	Input data pointer; passed to the Splitter by the runtime
`Data_size`	Input dataset size
`Output_data`	Output data pointer; buffer space allocated by user
`Splitter`	Pointer to Splitter function
`Map`	Pointer to Map function
`Reduce`	Pointer to Reduce function
`Partition`	Pointer to Partition function
`Key_cmp`	Pointer to key compare function
Optional Fields for Performance Tuning	
`Unit_size`	Pairs processed per Map/Reduce task
`L1_cache_size`	L1 data cache size in bytes
`Num_Map_workers`	Maximum number of threads (workers) for Map tasks
`Num_Reduce_workers`	Maximum number of threads (workers) for Reduce tasks
`Num_Merge_workers`	Maximum number of threads (workers) for Merge tasks
`Num_procs`	Maximum number of processors cores used

Evaluating MapReduce for Multi-core and Multiprocessor Systems, Colby Ranger @ 1 Computer Systems Laboratory, Stanford University

The Shared Memory Implementation

The Phoenix System

API

Runtime

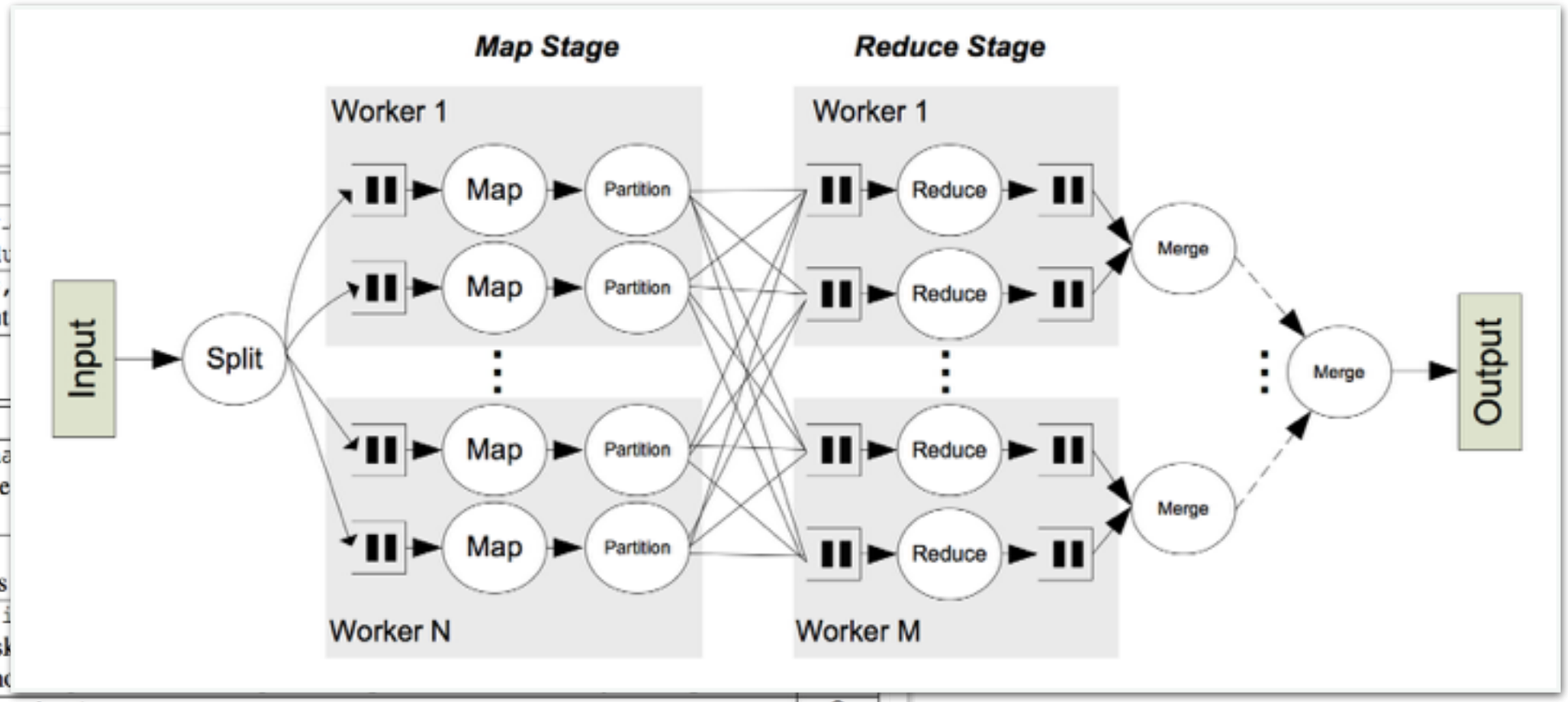
- Basic operation and control flow
- Buffer management
- Fault recovery
- Concurrency and locality management

The Shared Memory Implementation

The Phoenix System - The Runtime

Basic Operation and Control Flow

Function Description	
`int phoenix_scheduler (scheduler_` Initializes the runtime system. The `schedu`	
`void emit_intermediate(void *key,` Used in Map to emit an intermediate output	
`void emit(void *key, void *val)` Used in Reduce to emit a final output pair	
`int (*splitter_t)(void *, int, ma` Splits the input data across Map tasks. The input buffer pointer for each Map task	
`void (*map_t)(map_args_t*)` The Map function. Each Map task executes	
`int (*partition_t)(int, void *, i` Partitions intermediate pair for Reduce task pointer to the keys, and a the size of the key. Phc	
`void (*reduce_t)(void *, void **, int)` The Reduce function. Each reduce task executes this on its input. The arguments are a pointer to a key, a pointer to the associated values, and value count. If not specified, Phoenix uses a default *identity* function	O
`int (*key_cmp_t)(const void *, const void*)` Function that compares two keys	R

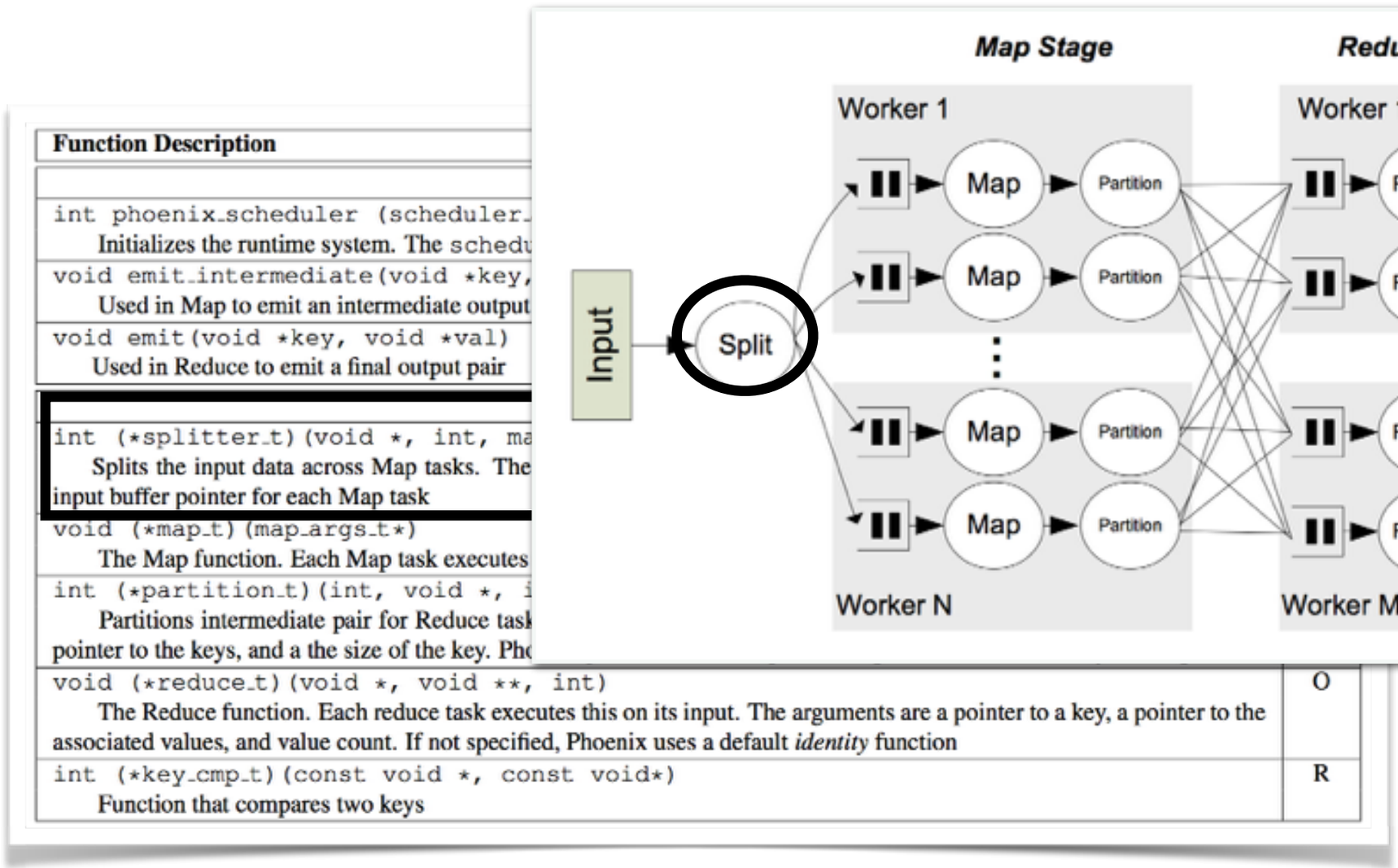


Evaluating MapReduce for Multi-core and Multiprocessor Systems, Colby Ranger @ l Computer Systems Laboratory, Stanford University

The Shared Memory Implementation

The Phoenix System - The Runtime

Basic Operation and Control Flow



Function Description	
`int phoenix_scheduler (scheduler_` Initializes the runtime system. The `schedu`	
`void emit_intermediate(void *key,` Used in Map to emit an intermediate output	
`void emit(void *key, void *val)` Used in Reduce to emit a final output pair	
`int (*splitter_t)(void *, int, ma` Splits the input data across Map tasks. The input buffer pointer for each Map task	
`void (*map_t)(map_args_t*)` The Map function. Each Map task executes	
`int (*partition_t)(int, void *, i` Partitions intermediate pair for Reduce task pointer to the keys, and a the size of the key. Pho	
`void (*reduce_t)(void *, void **, int)` The Reduce function. Each reduce task executes this on its input. The arguments are a pointer to a key, a pointer to the associated values, and value count. If not specified, Phoenix uses a default *identity* function	O
`int (*key_cmp_t)(const void *, const void*)` Function that compares two keys	R

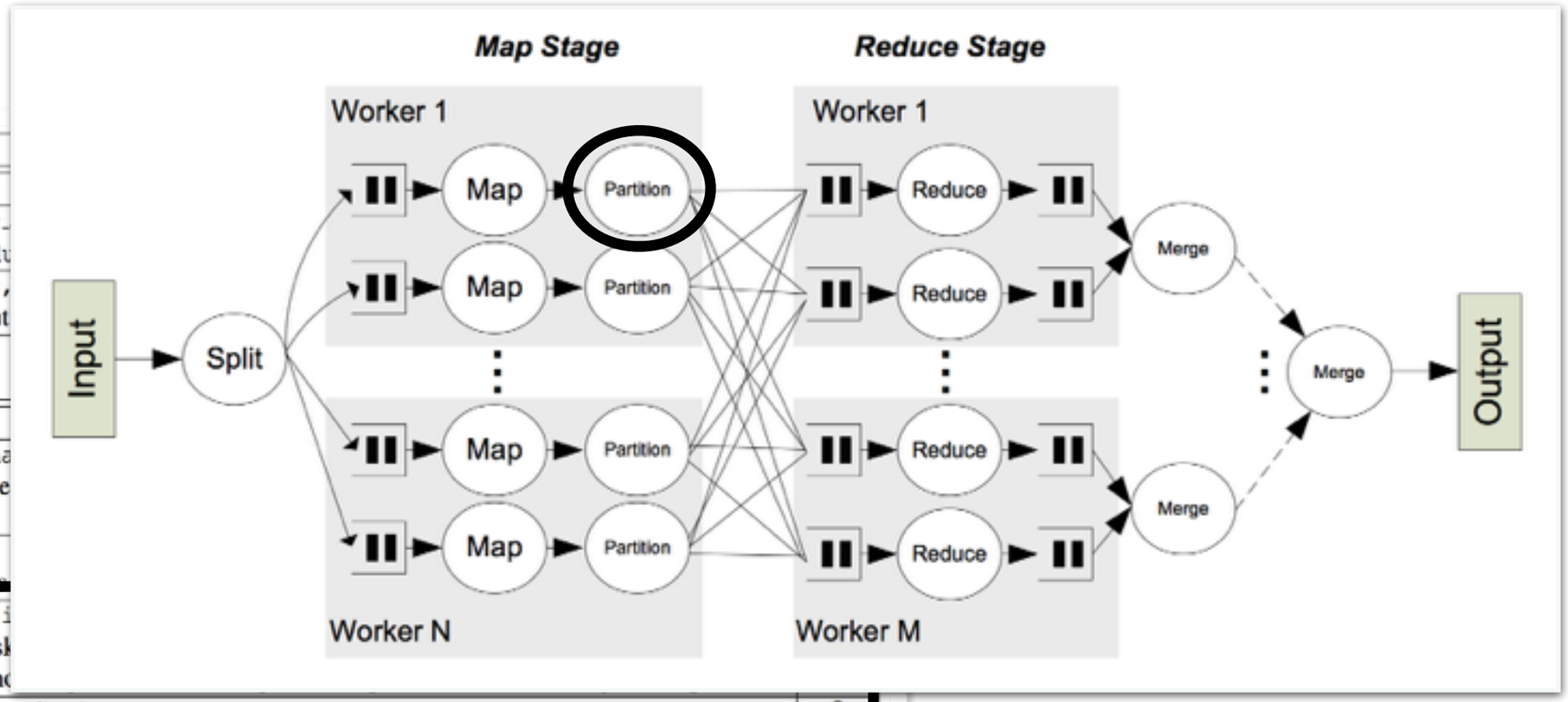
Evaluating MapReduce for Multi-core and Multiprocessor Systems, Colby Ranger @ l Computer Systems Laboratory, Stanford University

The Shared Memory Implementation

The Phoenix System - The Runtime

Basic Operation and Control Flow

Function	Description	Default
`int phoenix_scheduler (scheduler_`	Initializes the runtime system. The `schedu`	
`void emit_intermediate(void *key,`	Used in Map to emit an intermediate output	
`void emit(void *key, void *val)`	Used in Reduce to emit a final output pair	
`int (*splitter_t)(void *, int, ma`	Splits the input data across Map tasks. The input buffer pointer for each Map task	
`void (*map_t)(map_args_t*)`	The Map function. Each Map task executes	
`int (*partition_t)(int, void *, i`	Partitions intermediate pair for Reduce task pointer to the keys, and a the size of the key. Pho	
`void (*reduce_t)(void *, void **, int)`	The Reduce function. Each reduce task executes this on its input. The arguments are a pointer to a key, a pointer to the associated values, and value count. If not specified, Phoenix uses a default *identity* function	C
`int (*key_cmp_t)(const void *, const void*)`	Function that compares two keys	R

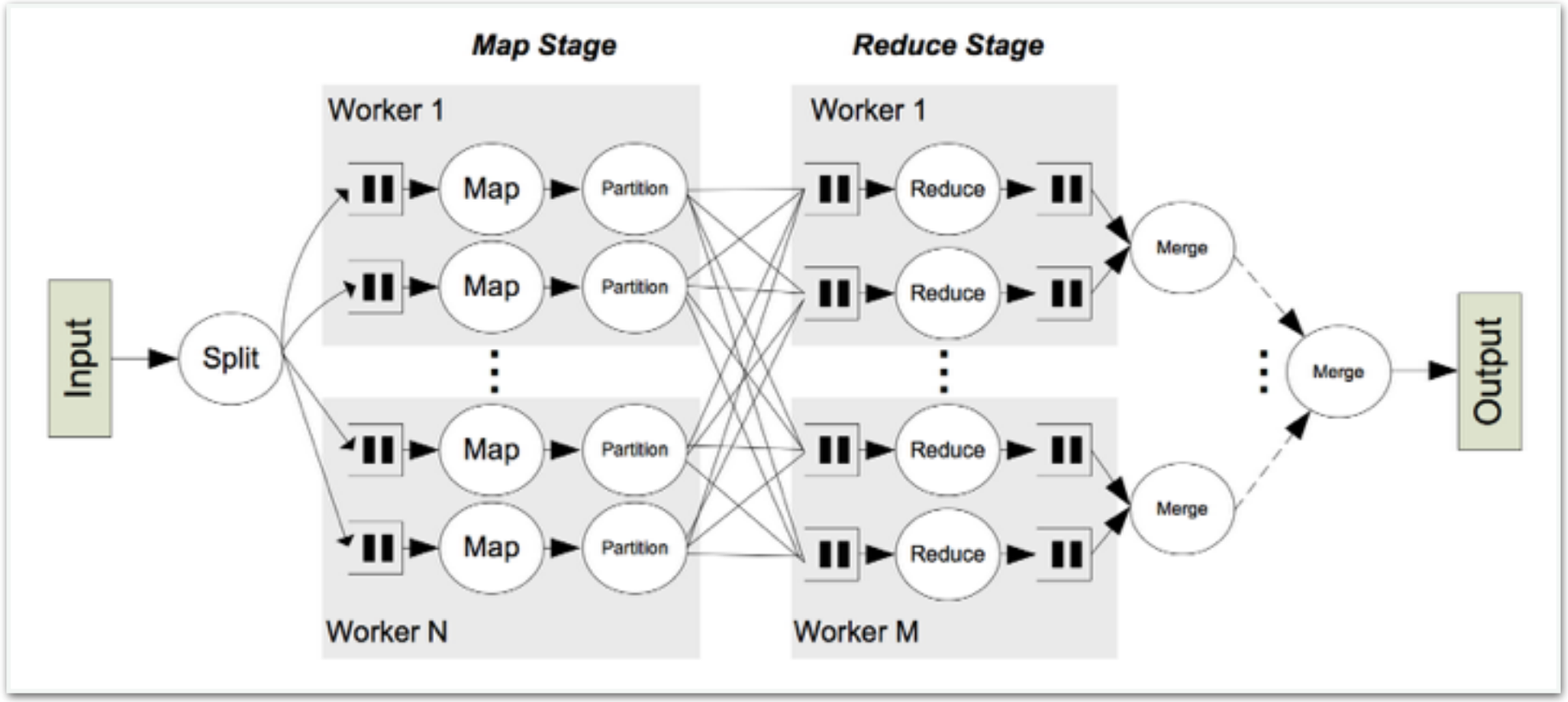


Evaluating MapReduce for Multi-core and Multiprocessor Systems, Colby Ranger @ l Computer Systems Laboratory, Stanford University

The Shared Memory Implementation

The Phoenix System - The Runtime

Buffer Management

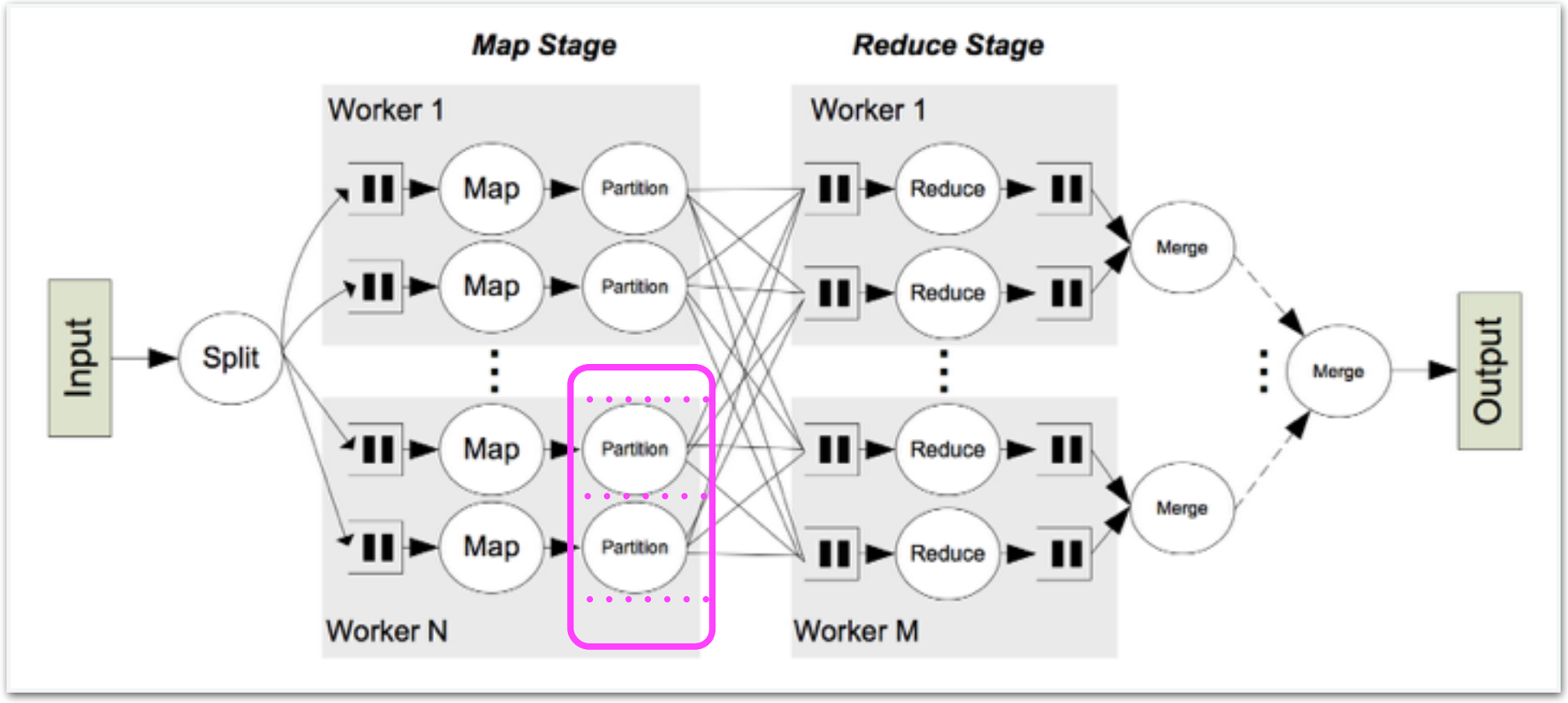


Evaluating MapReduce for Multi-core and Multiprocessor Systems, Colby Ranger @ l Computer Systems Laboratory, Stanford University

The Shared Memory Implementation

The Phoenix System - The Runtime

Buffer Management



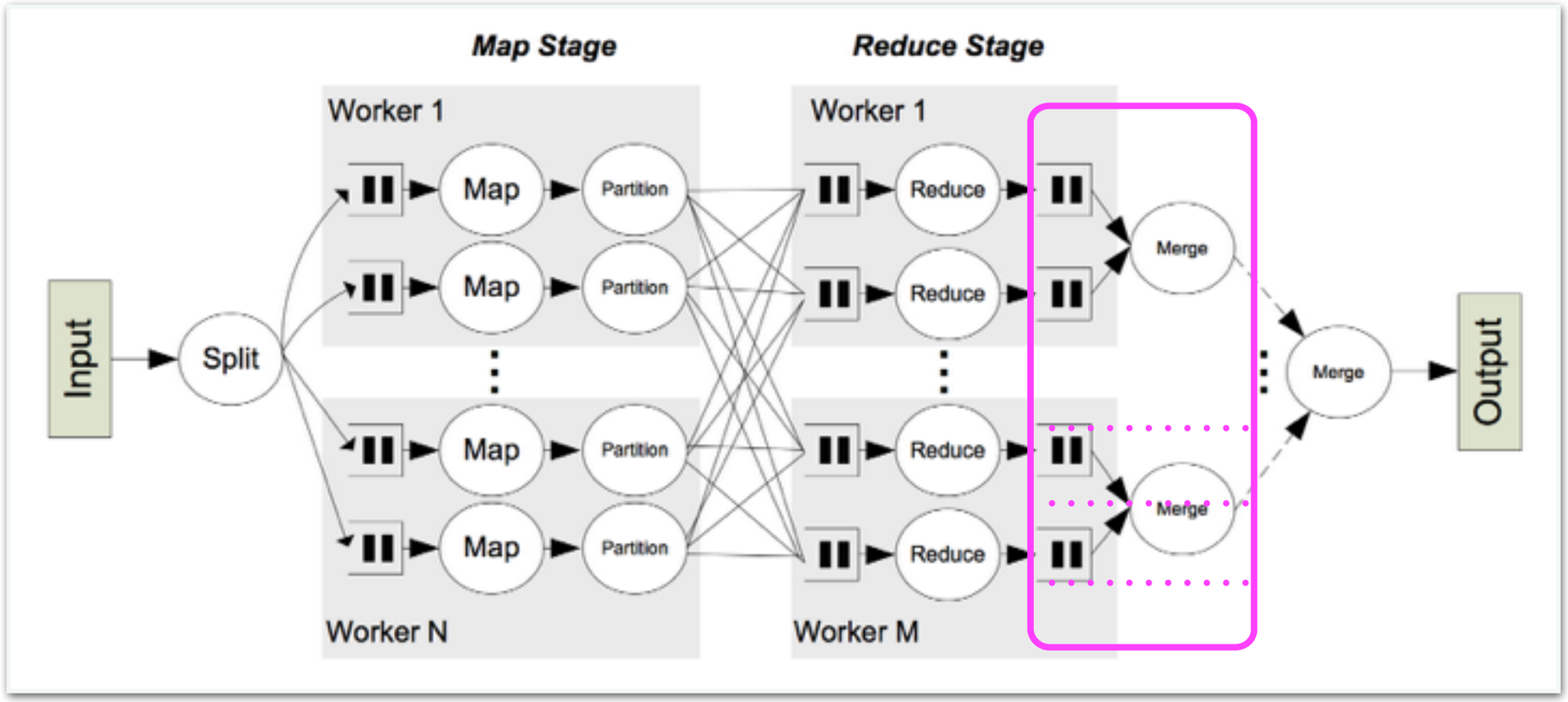
each worker has its own set of buffers resize dynamically

Evaluating MapReduce for Multi-core and Multiprocessor Systems, Colby Ranger @ l Computer Systems Laboratory, Stanford University

The Shared Memory Implementation

The Phoenix System - The Runtime

Buffer Management



Evaluating MapReduce for Multi-core and Multiprocessor Systems, Colby Ranger @ l Computer Systems Laboratory, Stanford University

The Shared Memory Implementation

The Phoenix System - The Runtime

Concurrency and Locality Management

- Number of Cores and Workers/Core
- Task Assignment
- Task Size
- Partition Function

ways to work with the Phoenix
- use a default policy for the specific system which has been developed taking into account its characteristics
- dynamically determine the best policy for each decision by monitoring resource availability and runtime behavior
- allow the programmer to provide application specific policies. Phoenix employs all three approaches in making the scheduling decisions

Evaluating MapReduce for Multi-core and Multiprocessor Systems, Colby Ranger @ l Computer Systems Laboratory, Stanford University

The Shared Memory Implementation

The Phoenix System - The Runtime

Shared Memory Concept

```
counts = {} # keys are words, counts are values

And now let's say that we have a method to count the words in a list:
```

```
1  def count_words(list_of_words):
2     for word in list_of_words:
3        if counts[word]:
4           counts[word] = counts[word] + 1
5        else:
6           counts[word] = 1
```

Methodology

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

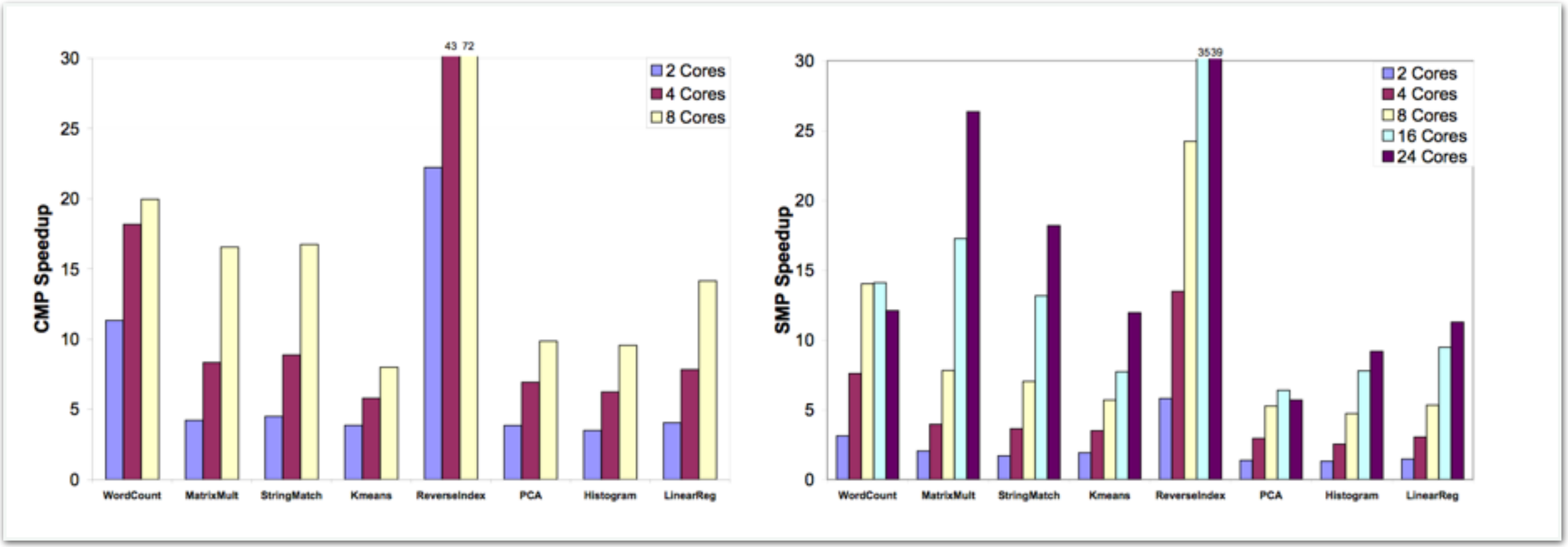
Methodology

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

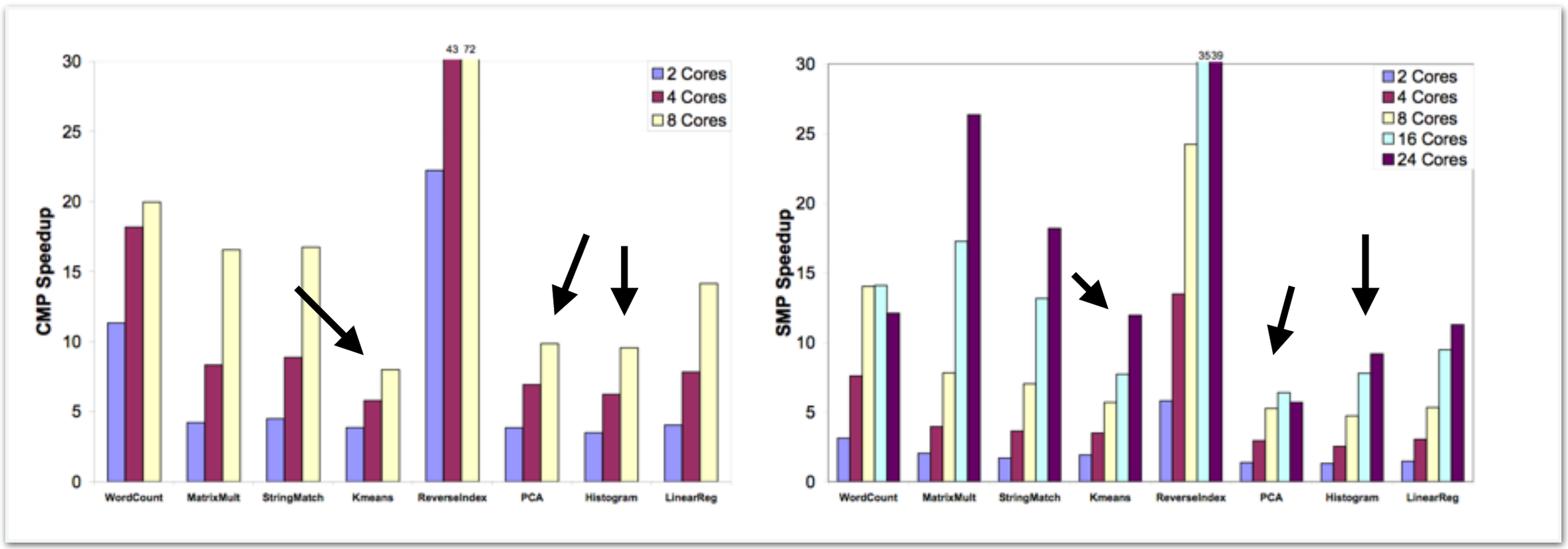
Evaluation

Basic Performance evaluation



Evaluation

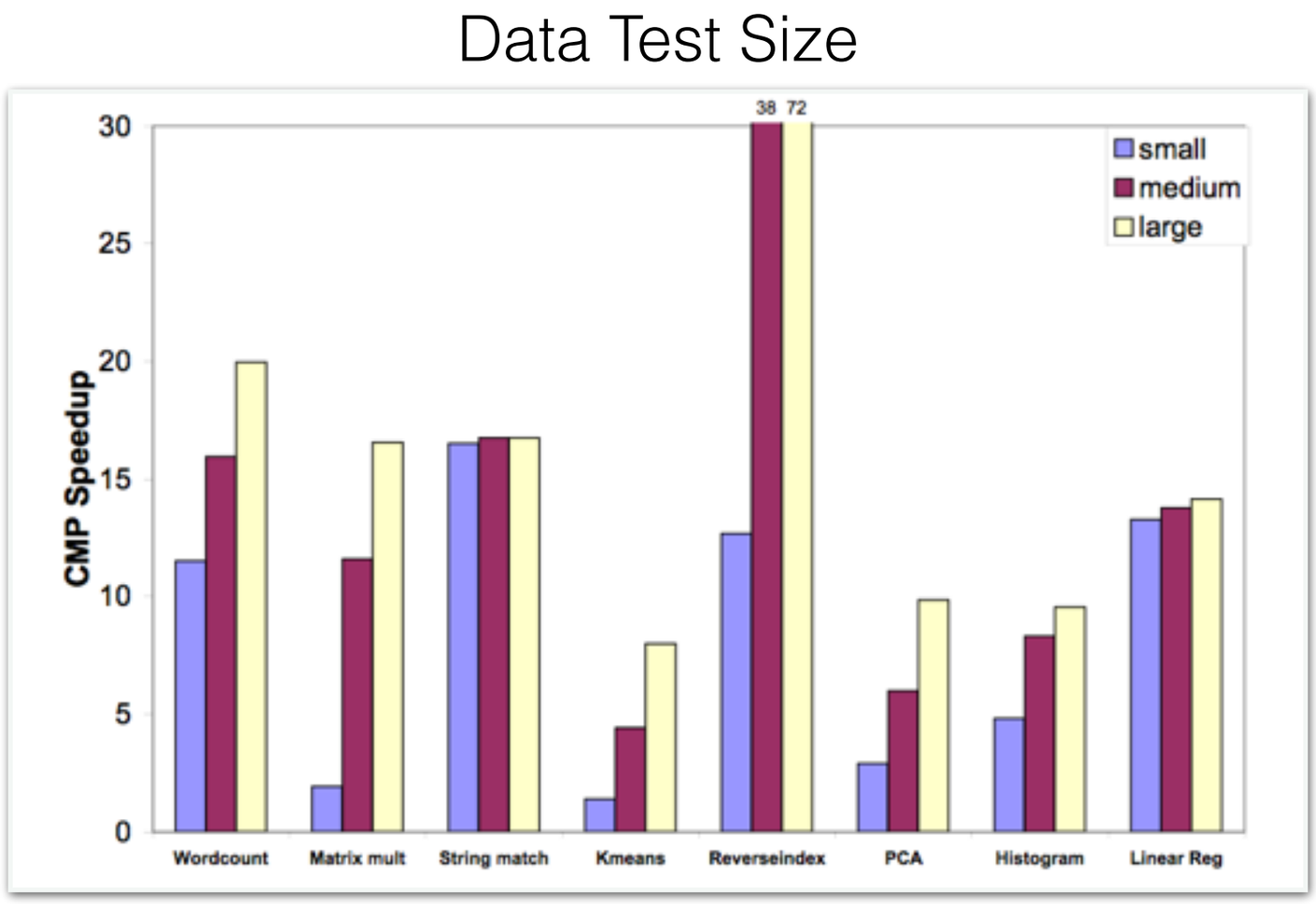
Basic Performance evaluation



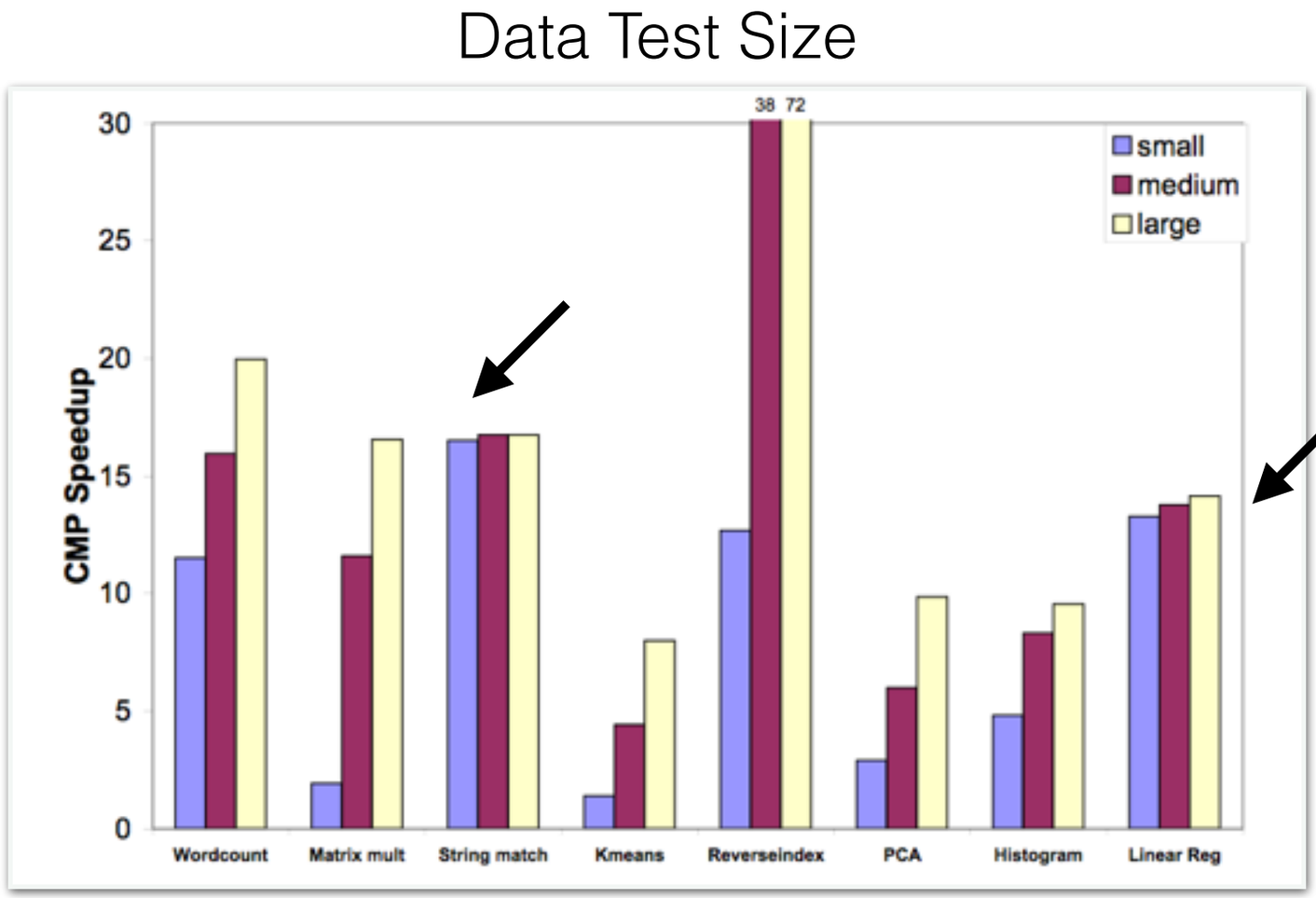
histogram and PCA are not key value naturally structured —> overheads

Evaluating MapReduce for Multi-core and Multiprocessor Systems, Colby Ranger @ l Computer Systems Laboratory, Stanford University

Evaluation


Data Test Size
CMP Speedup
30
25
20
15
10
5
0
38
72
small
medium
large
Wordcount
Matrix mult
String match
Kmeans
Reverseindex
PCA
Histogram
Linear Reg

Evaluation

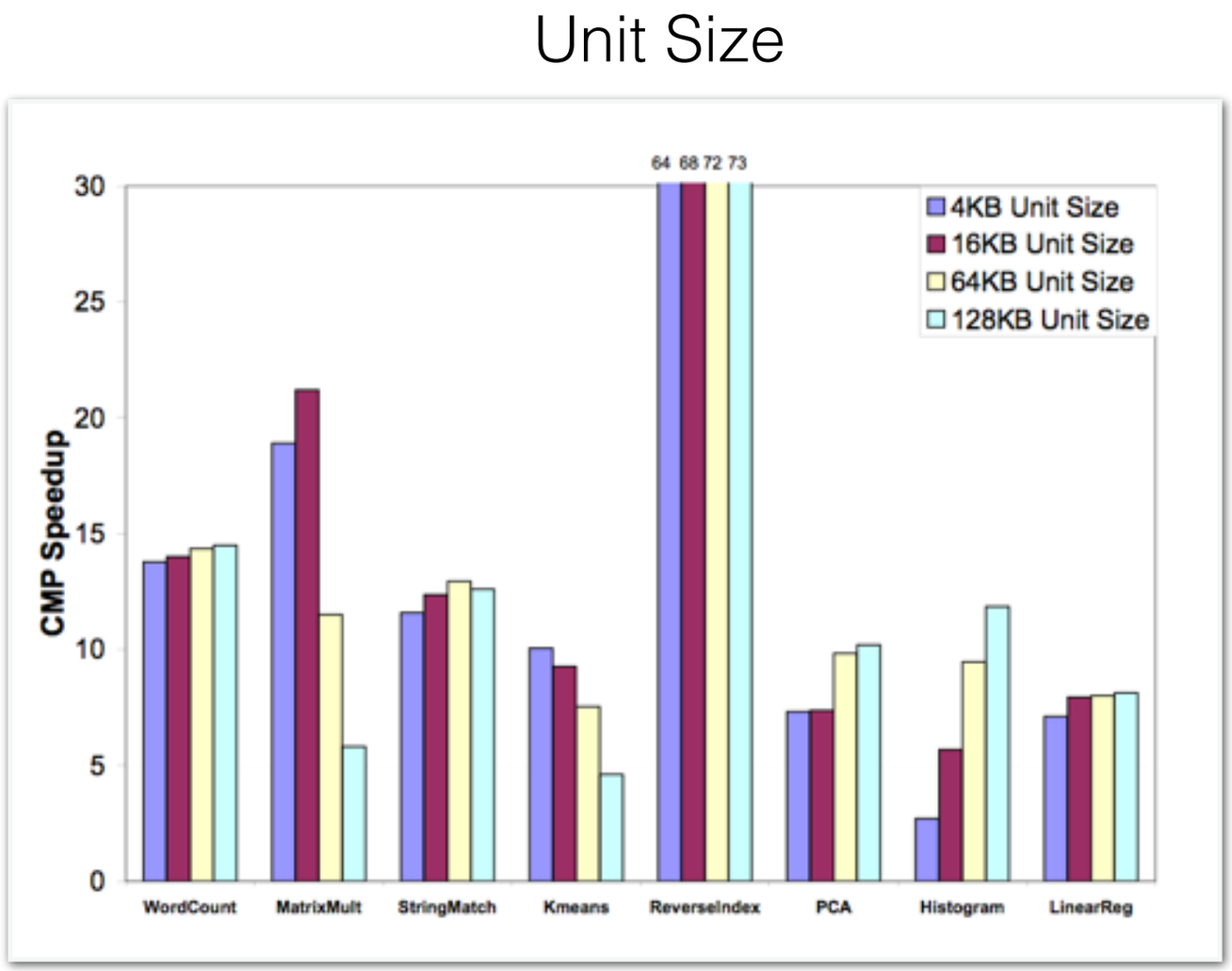


why linearReg and Matinv?

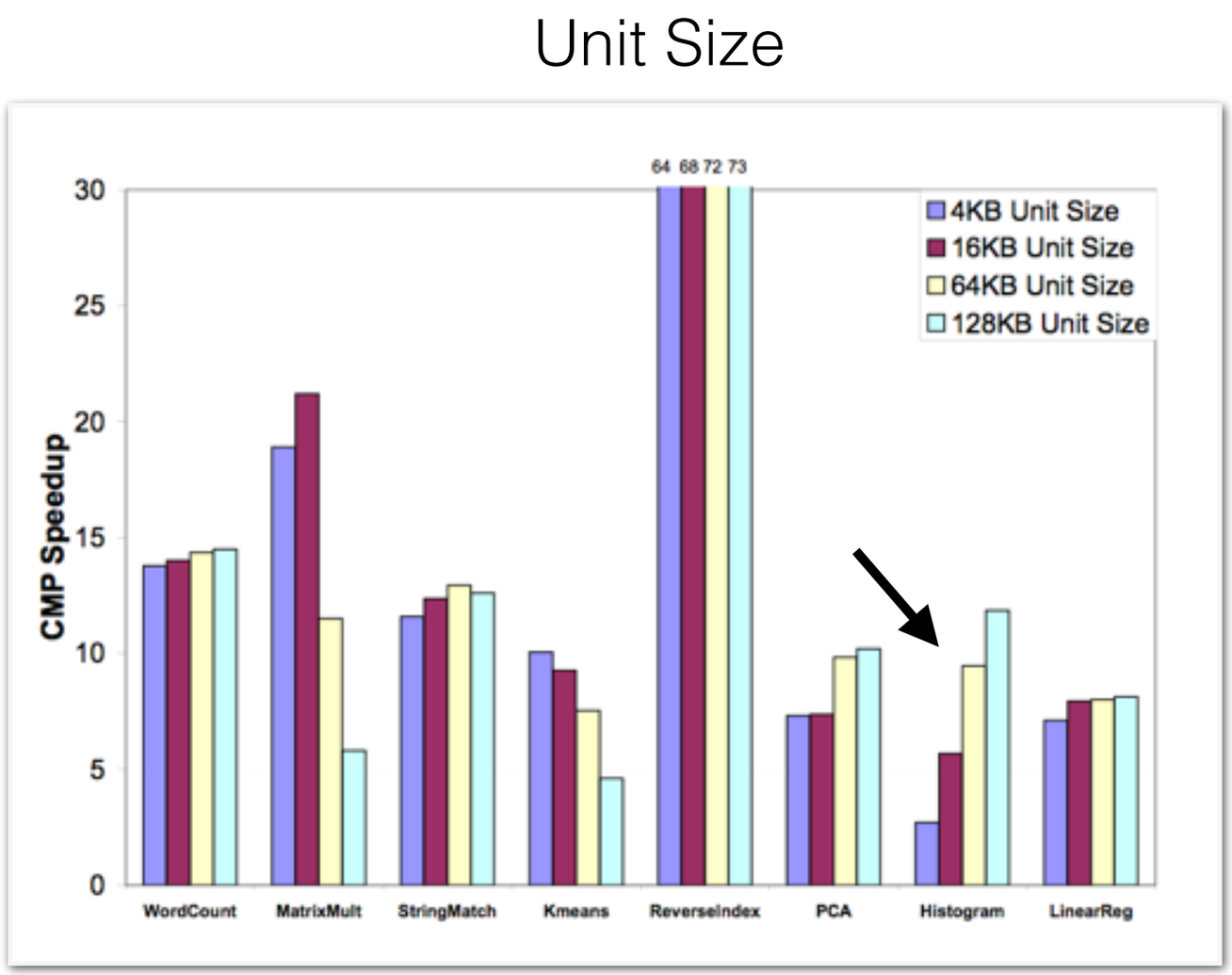
for small data set many operations- a lot of computation per element—> sufficient to fully utilize all available parallel resources

Evaluating MapReduce for Multi-core and Multiprocessor Systems, Colby Ranger @ l Computer Systems Laboratory, Stanford University

Evaluation

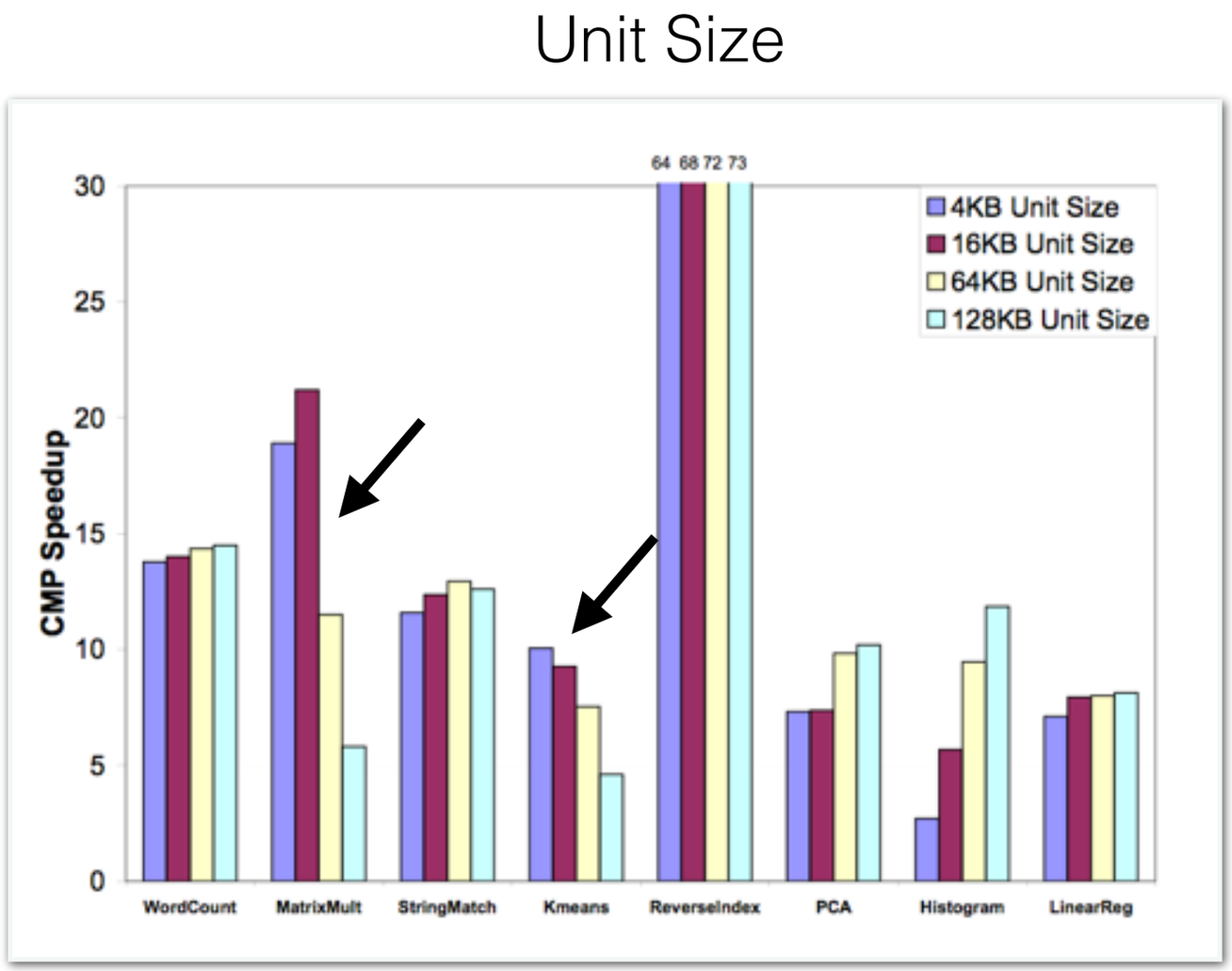

Unit Size
CMP Speedup
30
25
20
15
10
5
0
64 68 72 73
4KB Unit Size
16KB Unit Size
64KB Unit Size
128KB Unit Size
WordCount
MatrixMult
StringMatch
Kmeans
ReverseIndex
PCA
Histogram
LinearReg

Evaluation



in histogram it reduces the number of inter-mediate values to merge across task

Evaluation

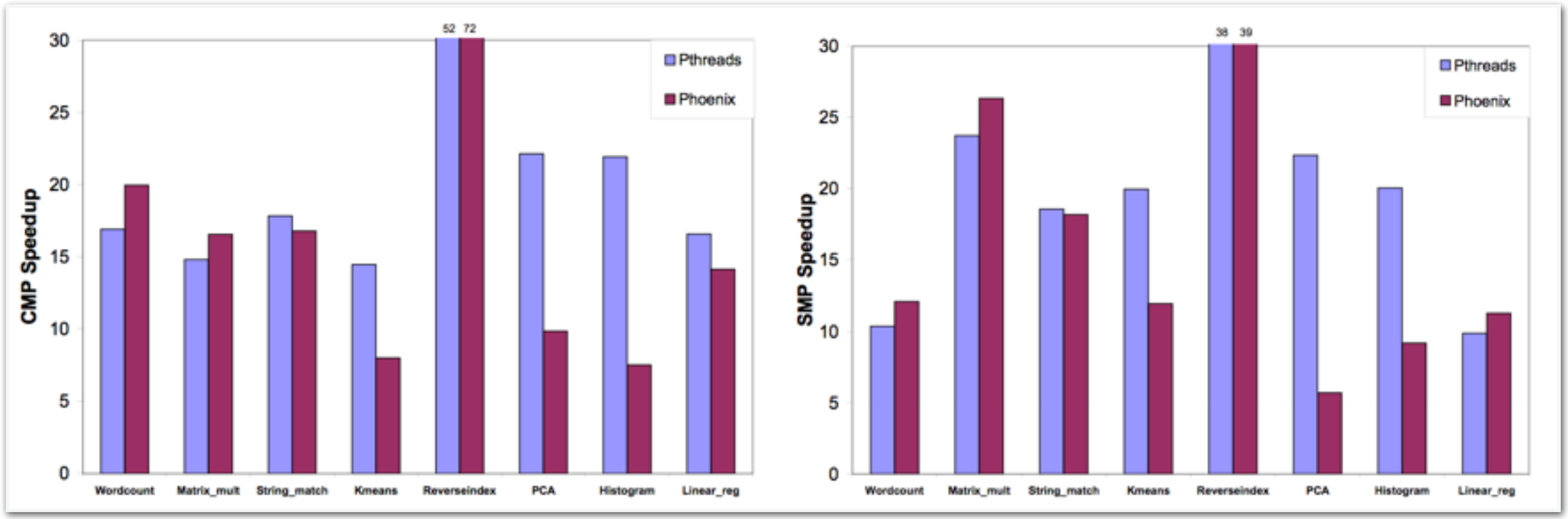


Kmeans and Matinv applications with short term temporal locality allow tasks to operate on data within their L1 cache or the data for all the active tasks to fit in the shared L2

Evaluating MapReduce for Multi-core and Multiprocessor Systems, Colby Ranger @ 1 Computer Systems Laboratory, Stanford University

Evaluation

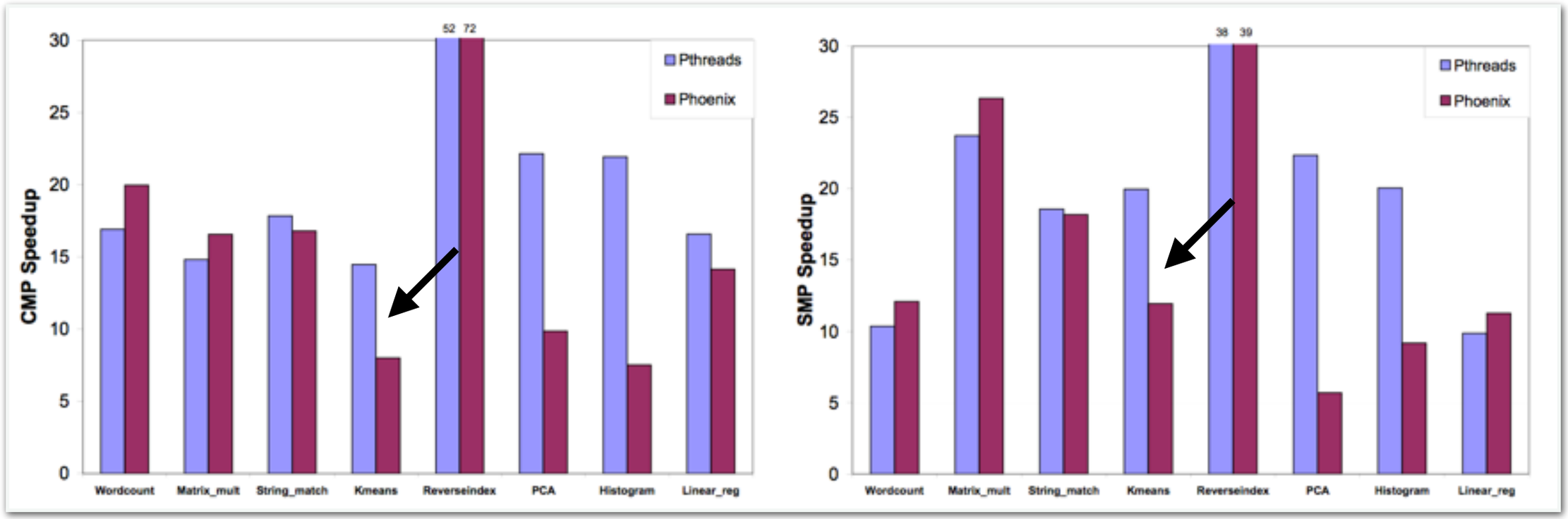
Comparison to Pthreads



Evaluating MapReduce for Multi-core and Multiprocessor Systems, Colby Ranger @ l Computer Systems Laboratory, Stanford University

Evaluation

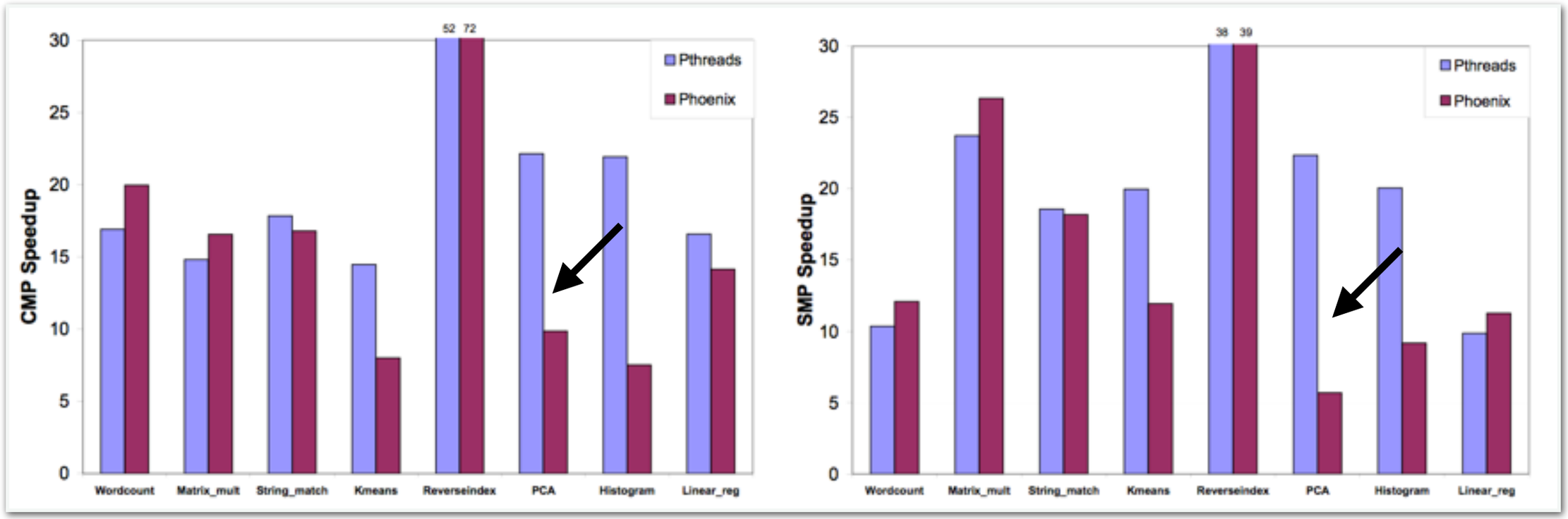
Comparison to Pthreads



Kmeans invokes the Phoenix scheduler iteratively, which introduces significant overhead, translate the output pair format to the input pair format

Evaluation

Comparison to Pthreads



MapReduce code does not use the original array structure Pca must track the coordinates for each data point separately. While the P-threads code uses direct array accesses and does not experience any additional overhead

Evaluation

An intuition?

Evaluation

An intuition?

Conclusion

- Phoenix leads to scalable performance for both multi-core chips and conventional symmetric multiprocessors
- Phoenix automatically handles key scheduling decisions during parallel execution
- Despite runtime overheads, Phoenix leads to similar performance for most applications
- Partition Function