Phoenix++: Modular MapReduce for Shared-Memory Systems

Justin Talbot, Richard Yoo, Christos Kozyrakis

Stanford University

Phoenix

Phoenix [Ranger et al., HPCA 2007]

- Cluster-style MapReduce on **shared-memory**

Phoenix 2 [Yoo et al., IISWC 2009]

- Explore shared-memory-specific details
  - Disk and network I/O no longer the bottleneck
  - Handling NUMA, reducing OS interaction and synchronization

Phoenix++ [today]

- High performance *and* simple code

Outline

1. Limitations of Phoenix
2. Related Work
3. Phoenix++ Design and Implementation
4. Performance Results

Limitations of Phoenix

Limitations of Phoenix

1. Inefficient key-value storage

   Fixed-width hash array + sorted key list

2. Ineffective combiner stage

   Combiner run at the *end* of the map stage

3. Exposed task chunking

   Interface exposes chunks, rather than single tasks

Limitations of Phoenix

```c
void map(pixel p) {
    emit(p.r, 1);
    emit(p.g+256, 1);
    emit(p.b+512, 1);
}
```

```c
void hist_map(map_args_t *args) {
    unsigned char *data = (unsigned char *) args->data;

    /* Manually buffer intermediate results */
    intptr_t red[256] = {0};
    intptr_t green[256] = {0};
    intptr_t blue[256] = {0};

    /* Count occurrences, amounts to manual combine */
    for (int i = 0; i < args->length * 3; i +=3) {
        red[data[i]]++;
        green[data[i+1]]++;
        blue[data[i+2]]++;
    }

    /* Selectively emit key-value pairs */
    for (int i = 0; i < 256; i++) {
        if(red[i] > 0) emit(i, red[i]);
        if(green[i] > 0) emit(i+256, green[i]);
        if(blue[i] > 0) emit(i+512, blue[i]);
    }
}
```

Limitations of Phoenix

```
void map(pixel p) {
    emit(p.r, 1);
    emit(p.g+256, 1);
    emit(p.b+512, 1);
}
```

```
void hist_map(map_args_t *args) {
    unsigned char *data = (unsigned char *) args->data;

    /* Manually buffer intermediate results */
    intptr_t red[256] = {0};
    intptr_t green[256] = {0};
    intptr_t blue[256] = {0};

    ts to manual combine */
    ength * 3; i +=3) {

    }

    /* Selectively emit key-value pairs */
    for (int i = 0; i < 256; i++) {
        if(red[i] > 0) emit(i, red[i]);
        if(green[i] > 0) emit(i+256, green[i]);
        if(blue[i] > 0) emit(i+512, blue[i]);
    }
}
```

histogram: 10x slowdown

linear_regression: 24x slowdown

Previous work

Previous Work

1. Inefficient key-value storage
2. Ineffective combiner stage
3. Exposed task chunking

Previous Work

[Tiled MapReduce, Chen et al. 2010]

1. Inefficient key-value storage

    Overlap map/reduce phases, shrinking working set

    Reduction function must be commutative, associative

2. Ineffective combiner stage

3. Exposed task chunking

Previous Work

[MATE, Jiang et al. 2010]

1. Inefficient key-value storage
   - Reduce run in map stage (as a combiner)
   - Reduction function must be commutative, associative
2. Ineffective combiner stage
   - User manually fuses map and combiner/reduction functions
3. Exposed task chunking

Previous Work

[Metis, Mao et al. 2010]

1. Inefficient key-value storage

   Fixed-width hash table + b-tree

   Estimate hash table width from 7% run

2. Ineffective combiner stage

   Run combiner if value buffer has more than 8 items

3. Exposed task chunking

Phoenix++ Design and Implementation

Design Goals

Pure

- keep map, combiner, reduce functions distinct
- no user-maintained state
- no exposed chunking

Complete

- no arbitrary restrictions on workloads
- handle non-associative reductions

Clean

- simple programmatic interface
- type safe

Fast

- make performance workarounds unnecessary

Design

1. Efficient key-value storage

   Modular storage options: *Containers* and *Combiner objects* abstractions support “mix and match”

2. Effective combiner stage

   Aggressively call combiner after *every* map emit

3. Encapsulated task chunking

   User-exposed functions called with one task at a time

   Compile-time optimizations eliminate overhead

Design: Modular storage options

Key distribution varies by workload

:	(word count)
***:k**	(histogram)
1:1	(matrix operations)

Design: Modular storage options

Key distribution varies by workload

		<u>*Container* type</u>
:	(word count)	variable-size hash table
***:k**	(histogram)	array with fixed mapping
1:1	(matrix operations)	shared array

Design: Modular storage options

```
// Begin map stage (Phoenix++ library)
storage = Container.get()
while(chunk in queue) {
   for(task in chunk) {
         user_map_fn(task.data, storage)
   }
}
Container.put(storage)
// End map stage

// User map function
user_map_fn(…) {
   …
   emit(storage, key, value)
}
```

Design: Modular storage options

```
// Begin map stage (Phoenix++ library)
storage = Container.get()
while(chunk in queue) {
    for(task in chunk) {
        user_map_fn(task.data, storage)
    }
}
Container.put(storage)
// End map stage

// User map function
user_map_fn(…) {
    …
    emit(storage, key, value)
}
```

Design: Modular storage options

```
// Begin map stage (Phoenix++ library)
storage = Container.get()
while(chunk in queue) {
   for(task in chunk) {
         user_map_fn(task.data, storage)
   }
}
Container.put(storage)
// End map stage

// User map function
user_map_fn(…) {
   …
   emit(storage, key, value)
}
```

Design: Modular storage options

```
// Begin map stage (Phoenix++ library)
storage = Container.get()
while(chunk in queue) {
   for(task in chunk) {
        user_map_fn(task.data, storage)
   }
}
Container.put(storage)
// End map stage

// User map function
user_map_fn(…) {
   …
   emit(storage, key, value)
}
```

Design: Modular storage options

```
// Begin map stage (Phoenix++ library)
storage = Container.get()
while(chunk in queue) {
   for(task in chunk) {
        user_map_fn(task.data, storage)
   }
}
Container.put(storage)
// End map stage

// User map function
user_map_fn(…) {
   …
   emit(storage, key, value)
}
```

Design: Modular storage options

	`Container::get()`	`Container::put()`
variable-size hash table	thread-local hash table	rehash table to # of reduce tasks
array	thread-local array	swap pointer to global memory
shared array	pointer to global array	-

Design: Modular storage options

Advantages:

- Storage can be optimized for a particular workload
- Users may provide own container implementation
- Hash tables resize dynamically and independently
- Thread-local storage can be optimized by compiler

Disadvantages:

- Introduces rehash between map and reduce stages

Design: Effective combiners

Combiners are stateful objects in Phoenix++

- Used to store all emitted values with the same key

2 implementations:

- `buffer_combiner`
  - standard MapReduce behavior
- `associative_combiner`:
  - applies associative function on every emit
  - only stores cumulative value

Design: Effective combiners

Advantages:

- associative combiners minimize storage
- associative combiners have no buffer maintenance overhead
- preserve support for non-associative reductions

Design: Encapsulated Chunking

```
// Begin map stage (Phoenix++ library)
thread_local_storage = Container.get()
while(chunk in queue) {
   for(task in chunk) {
         user_map_fn(task_data, thread_local_storage)
   }
}
Container.put(thread_local_storage)
// End map stage
```

Design: Encapsulated Chunking

```
// Begin map stage (Phoenix++ library)
thread_local_storage = Container.get()
while(chunk in queue) {
    for(task in chunk) {
        user_map_fn(task_data, thread_local_storage)
    }
}
Container.put(thread_local_storage)
// End map stage
```

Design: Encapsulated Chunking

Introduces large number of function calls

(also, calling combiner on every emit)

C++ templates to statically inline functions

Design: Encapsulated Chunking

```
class Histogram : public MapReduceSort<
    Histogram, pixel, intptr_t, uint64_t,
    array_container<intptr_t, uint64_t,
    sum_combiner, 768> > {
public:
    void map(pixel const& p, container& out)
          const {
        emit(out, p.r, 1);
        emit(out, p.g+256, 1);
        emit(out, p.b+512, 1);
    }
};
```

```
.L734:
```

emit r

```
    movzbl  -3(%rsi), %eax
    addq    $1, (%rbx,%rax,8)
```

emit g

```
    movzbl  -2(%rsi), %eax
    addq    $1, 2048(%rbx,%rax,8)
```

emit b

```
    movzbl  -1(%rsi), %eax
    addq    $3, %rsi
```

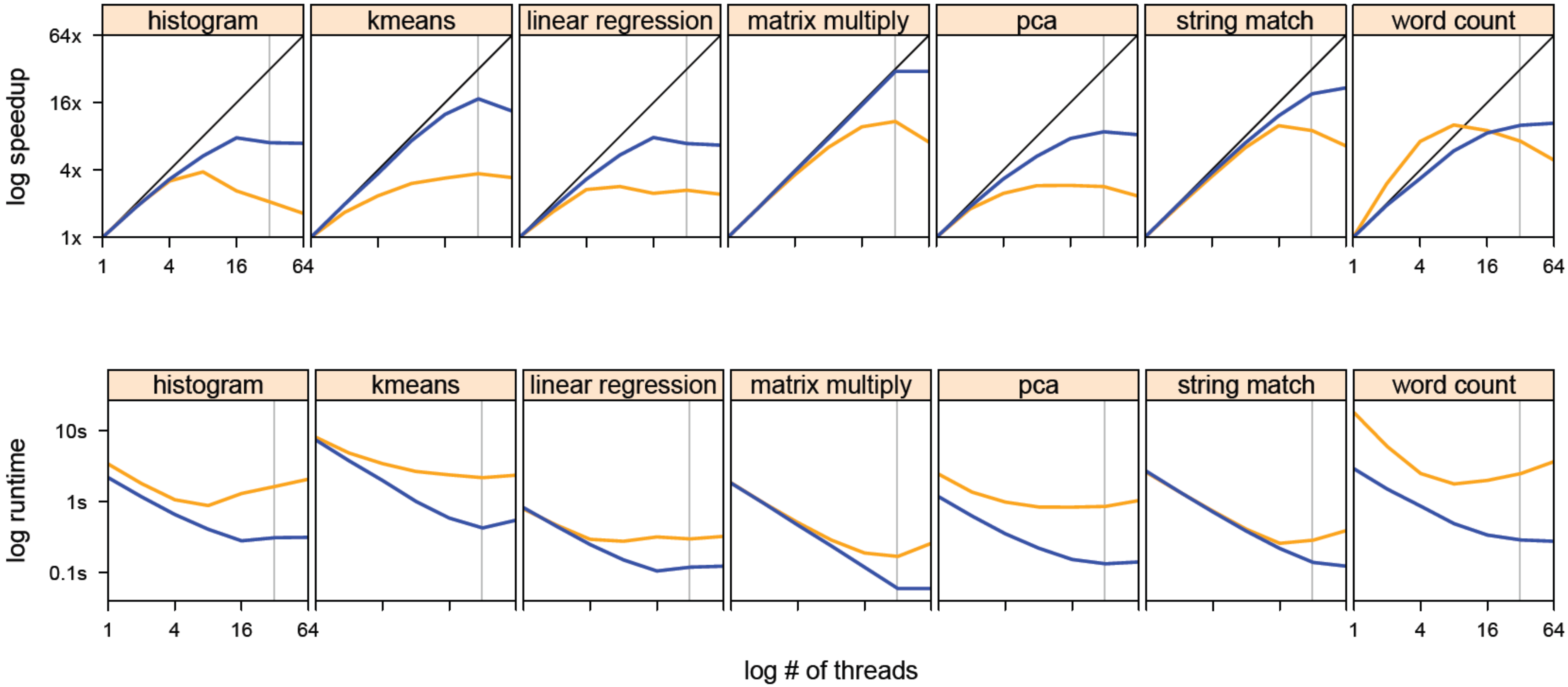
loop over tasks

```
    addq    $1, 4096(%rbx,%rax,8)
    cmpq    %rsi, %rdx
    je      .L752
    jmp     .L734
```

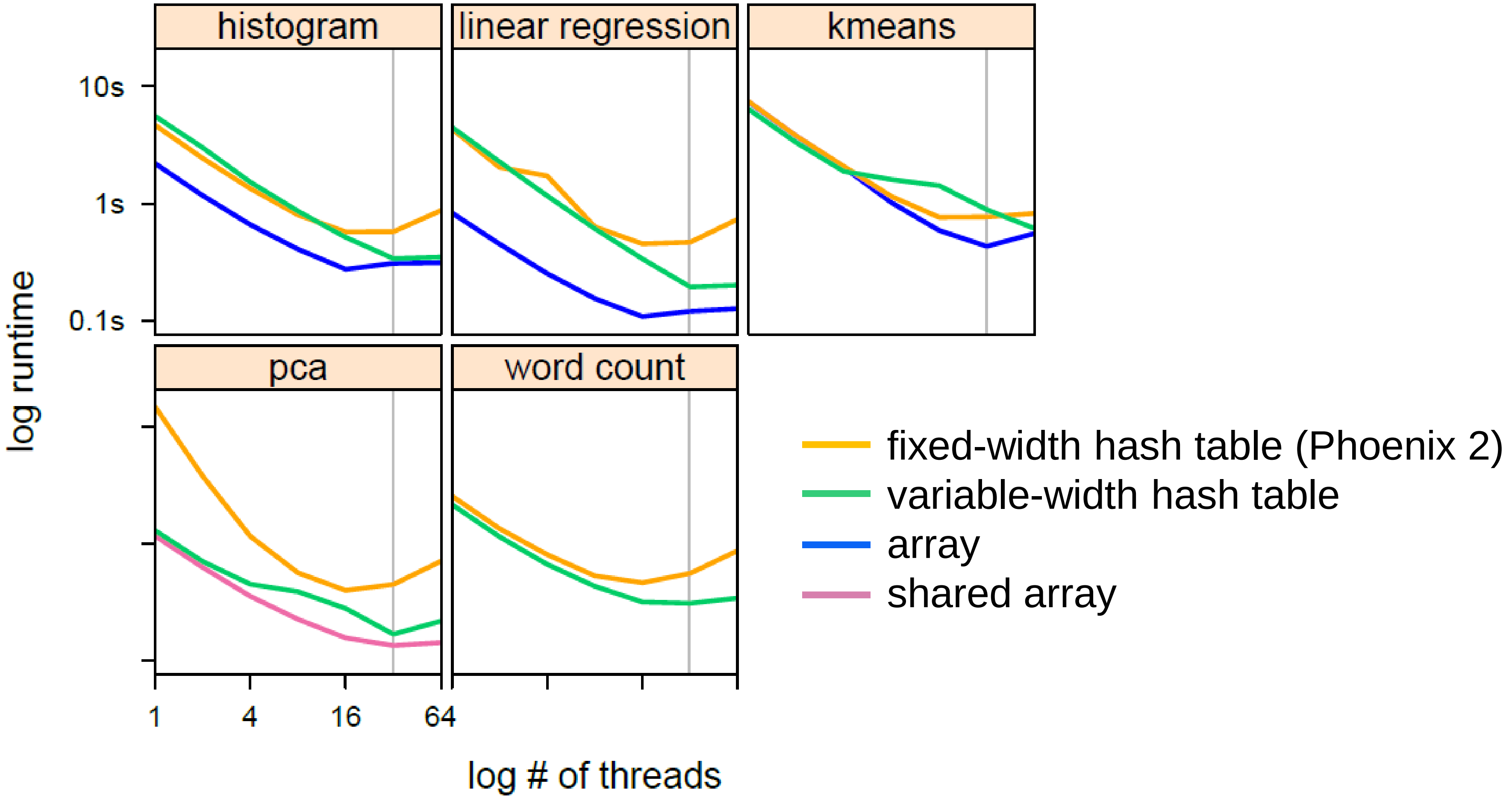
Performance Results

Performance Summary

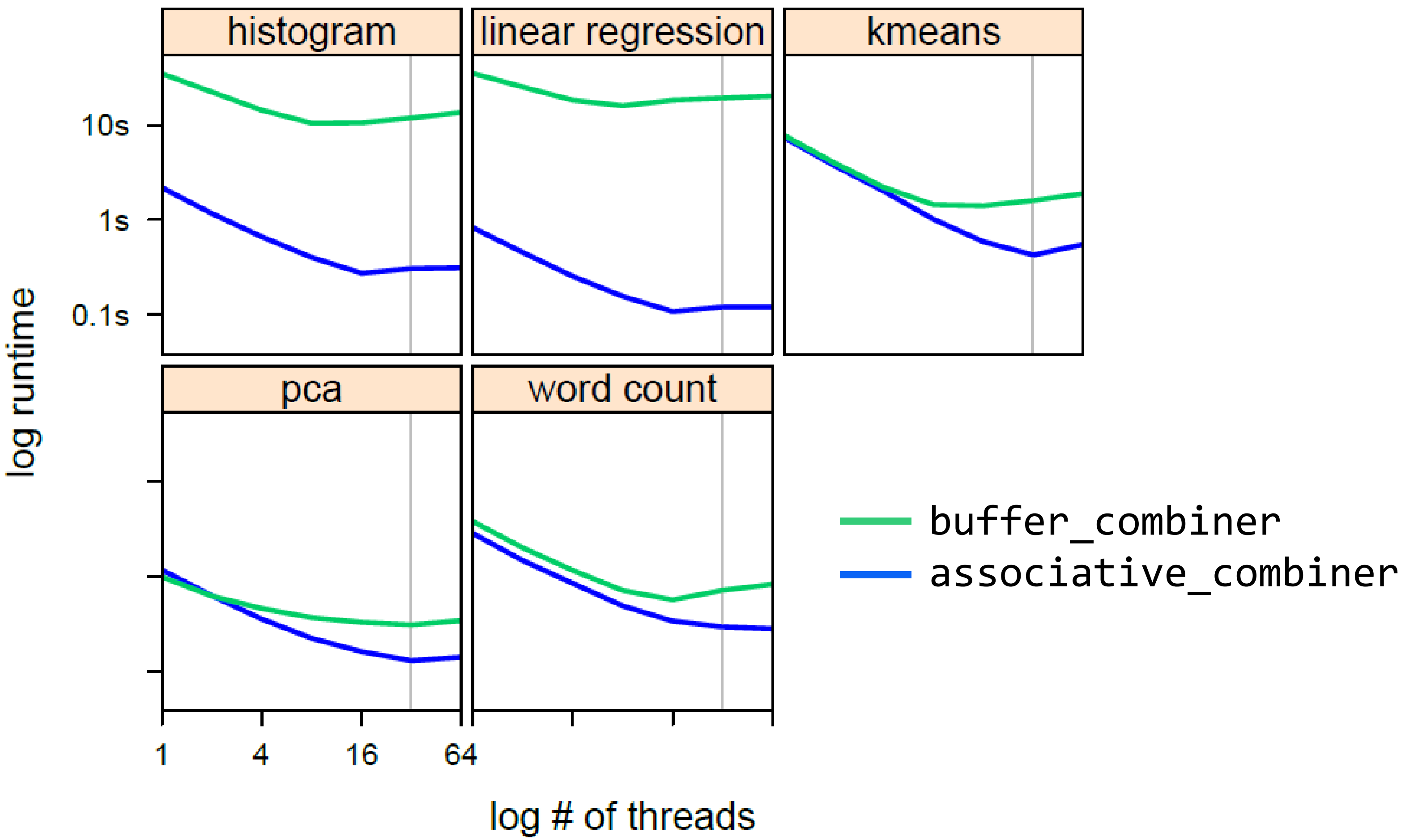
32-core, 64-HW context Nehalem



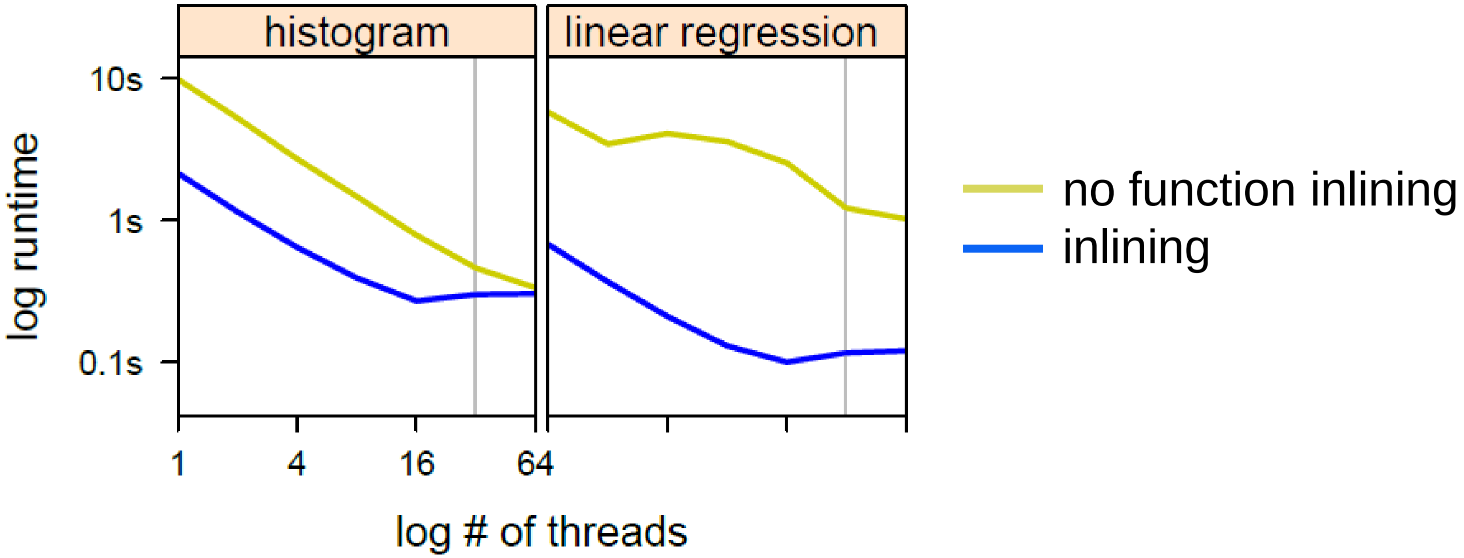
Container Sensitivity



Combiner Performance



Function Call Overhead



Performance Summary

All 3 changes contributed to observed higher performance

Average improvement over Phoenix 2: **4.7x**

Code Size Comparison

	map		reduce		combiner	
	P++	P2	P++	P2	P++	P2
histogram	5	39	0	13	0	11
kmeans	30	47	5	33	11	0
linear_regression	9	34	0	14	0	14
matrix_multiply	12	26	0	0	0	0
pca	24	56	0	0	0	0
string_match	31	36	0	0	0	0
word_count	26	53	0	13	0	11

Summary

Phoenix++

- A modular, flexible, high performance MapReduce library for shared memory machines
- Demonstrated high performance without sacrificing simple, standard MapReduce interface
- Based on adapting pipeline to workload properties and carefully leveraging compiler optimizations for performance

Questions?

Code available at

http://mapreduce.stanford.edu

Justin Talbot: jtalbot@stanford.edu