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
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};

    /* 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

```c
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};
    /* 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]);
    }
}
```

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
[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
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

<table>
<thead>
<tr>
<th>Container type</th>
<th><em>::</em> (word count)</th>
<th>*::k (histogram)</th>
<th>1:1 (matrix operations)</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable-size hash table</td>
<td>array with fixed mapping</td>
<td>shared array</td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th></th>
<th>Container::get()</th>
<th>Container::put()</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable-size hash table</td>
<td>thread-local hash table</td>
<td>rehash table to # of reduce tasks</td>
</tr>
<tr>
<td>array</td>
<td>thread-local array</td>
<td>swap pointer to global memory</td>
</tr>
<tr>
<td>shared array</td>
<td>pointer to global array</td>
<td>-</td>
</tr>
</tbody>
</table>
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 \texttt{MapReduceSort}<
\begin{align*}
\text{Histogram, pixel, intptr_t, uint64_t,}
\text{array\_container< intptr_t, uint64_t,}
\text{sum\_combiner, 768>} > \\
\text{public:}
\end{align*}
void map(pixel const& p, container& out) 
\begin{align*}
\text{const} \{ \\
\text{emit(out, p.r, 1);} \\
\text{emit(out, p.g+256, 1);} \\
\text{emit(out, p.b+512, 1);} \\
\} \\
\}
\]

.L734:
\begin{align*}
&\text{movzbl -3(%rsi), %eax} \\
&\text{addq $1, (%rbx,%rax,8)} \\
&\text{movzbl -2(%rsi), %eax} \\
&\text{addq $1, 2048(%rbx,%rax,8)} \\
&\text{movzbl -1(%rsi), %eax} \\
&\text{addq $3, %rsi} \\
&\text{addq $1, 4096(%rbx,%rax,8)} \\
&\text{cmpq %rsi, %rdx} \\
&\text{je .L752} \\
&\text{jmp .L734}
\end{align*}
Performance Results
Performance Summary

32-core, 64-HW context Nehalem
Container Sensitivity

- fixed-width hash table (Phoenix 2)
- variable-width hash table
- array
- shared array

Graphs show runtime against the number of threads for different operations:
- histogram
- linear regression
- kmeans
- pca
- word count

Legend:
- fixed-width hash table (Phoenix 2)
- variable-width hash table
- array
- shared array
Combiner Performance

- **buffer_combiner**
- **associative_combiner**

![Graphs showing performance comparison between buffer_combiner and associative_combiner for different tasks (histogram, linear regression, kmeans, pca, word count) with varying number of threads.](image-url)
Function Call Overhead

![Graph showing the comparison between function call overhead with and without inlining for histogram and linear regression tasks. The x-axis represents the logarithm of the number of threads, and the y-axis represents the logarithm of runtime. The graph shows a significant reduction in runtime for both tasks with inlining compared to no function inlining.](image-url)
Performance Summary

All 3 changes contributed to observed higher performance

Average improvement over Phoenix 2: 4.7x
## Code Size Comparison

<table>
<thead>
<tr>
<th></th>
<th>map</th>
<th></th>
<th>reduce</th>
<th></th>
<th>combiner</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P++</td>
<td>P2</td>
<td>P++</td>
<td>P2</td>
<td>P++</td>
</tr>
<tr>
<td>histogram</td>
<td>5</td>
<td>39</td>
<td>0</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>kmeans</td>
<td>30</td>
<td>47</td>
<td>5</td>
<td>33</td>
<td>11</td>
</tr>
<tr>
<td>linear_regression</td>
<td>9</td>
<td>34</td>
<td>0</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>matrix_multiply</td>
<td>12</td>
<td>26</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pca</td>
<td>24</td>
<td>56</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>string_match</td>
<td>31</td>
<td>36</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>word_count</td>
<td>26</td>
<td>53</td>
<td>0</td>
<td>13</td>
<td>0</td>
</tr>
</tbody>
</table>
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