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 Programing 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})
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/mapreduce

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

<table>
<thead>
<tr>
<th>Function Description</th>
<th>R/O</th>
</tr>
</thead>
</table>
| 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 |

<table>
<thead>
<tr>
<th>Functions Provided by Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
</tr>
</tbody>
</table>
| 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 @ 1 Computer Systems Laboratory, Stanford University
The Shared Memory Implementation
The Phoenix System - API

<table>
<thead>
<tr>
<th>Function Description</th>
<th>R/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>int phoenix.scheduler (scheduler.args.t * args)</td>
<td>R</td>
</tr>
<tr>
<td>Initializes the runtime system. The scheduler.args.t struct provides the needed function &amp; data pointers</td>
<td></td>
</tr>
<tr>
<td>void emit.intermediate(void *key, void *val, int key.size)</td>
<td>O</td>
</tr>
<tr>
<td>Used in Map to emit an intermediate output &lt;key, value&gt; pair. Required if the Reduce is defined</td>
<td></td>
</tr>
<tr>
<td>void emit(void *key, void *val)</td>
<td>O</td>
</tr>
<tr>
<td>Used in Reduce to emit a final output pair</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functions Defined by User</th>
<th>R/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>int (*splitter.t)(void *, int, map.args.t *)</td>
<td>R</td>
</tr>
<tr>
<td>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</td>
<td></td>
</tr>
<tr>
<td>void (<em>map.t)(map.args.t</em>)</td>
<td>R</td>
</tr>
<tr>
<td>The Map function. Each Map task executes this function on its input</td>
<td></td>
</tr>
<tr>
<td>int (*partition.t)(int, void *, int)</td>
<td>O</td>
</tr>
<tr>
<td>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</td>
<td></td>
</tr>
<tr>
<td>void (*reduce.t)(void *, void **, int)</td>
<td>O</td>
</tr>
<tr>
<td>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</td>
<td></td>
</tr>
<tr>
<td>int (*key_cmp.t)(const void <em>, const void</em>)</td>
<td>R</td>
</tr>
<tr>
<td>Function that compares two keys</td>
<td></td>
</tr>
</tbody>
</table>
## The Shared Memory Implementation

### The Phoenix System - API

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic Fields</strong></td>
<td></td>
</tr>
<tr>
<td>Input_data</td>
<td>Input data pointer; passed to the Splitter by the runtime</td>
</tr>
<tr>
<td>Data_size</td>
<td>Input dataset size</td>
</tr>
<tr>
<td>Output_data</td>
<td>Output data pointer; buffer space allocated by user</td>
</tr>
<tr>
<td>Splitter</td>
<td>Pointer to Splitter function</td>
</tr>
<tr>
<td>Map</td>
<td>Pointer to Map function</td>
</tr>
<tr>
<td>Reduce</td>
<td>Pointer to Reduce function</td>
</tr>
<tr>
<td>Partition</td>
<td>Pointer to Partition function</td>
</tr>
<tr>
<td>Key_cmp</td>
<td>Pointer to key compare function</td>
</tr>
<tr>
<td><strong>Optional Fields for Performance Tuning</strong></td>
<td></td>
</tr>
<tr>
<td>Unit_size</td>
<td>Pairs processed per Map/Reduce task</td>
</tr>
<tr>
<td>L1_cache.size</td>
<td>L1 data cache size in bytes</td>
</tr>
<tr>
<td>Num_Map_workers</td>
<td>Maximum number of threads (workers) for Map tasks</td>
</tr>
<tr>
<td>Num_Reduce_workers</td>
<td>Maximum number of threads (workers) for Reduce tasks</td>
</tr>
<tr>
<td>Num_Merge_workers</td>
<td>Maximum number of threads (workers) for Merge tasks</td>
</tr>
<tr>
<td>Num_procs</td>
<td>Maximum number of processors cores used</td>
</tr>
</tbody>
</table>
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

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
The Shared Memory Implementation

The Phoenix System - The Runtime

Basic Operation and Control Flow

---

Evaluating MapReduce for Multi-core and Multiprocessor Systems, Colby Ranger @ 1 Computer Systems Laboratory, Stanford University
The Shared Memory Implementation
The Phoenix System - The Runtime
Buffer Management
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 @ I Computer Systems Laboratory, Stanford University
The Shared Memory Implementation
The Phoenix System - The Runtime
Buffer Management
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
counts = {} # keys are words, counts are values

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

```python
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

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

## Applications

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

---

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

Basic Performance evaluation

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

Basic Performance evaluation

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

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

Data Test Size

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

Unit Size

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

Unit Size

in histogram it reduces the number of intermediate 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 cache.

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

Comparison to Pthreads
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