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 automatically 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

```java
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*

```c
int phoenix_scheduler(scheduler_args_t *args)
• Initializes the runtime system.
• scheduler_args_t provides the needed functions and data pointers.
```

```c
void emit_intermediate(void *key, void *val, int key_size)
• Used in Map to emit intermediate output.
```

```c
void emit(void *key, void *val)
• Used in Reduce to emit a final output pair.
```
Phoenix

Functions defined by the user

```c
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

- 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.
- Reduce tasks dynamically assigned to workers.
- Possibly higher imbalance. (Same key $\rightarrow$ 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 intermediate 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

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

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

## 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>S:10MB, M:50MB, L:100MB</td>
<td>1.8 0.9</td>
</tr>
<tr>
<td><strong>Matrix Multiply</strong></td>
<td>S:100x100, M:500x500, L:1000x1000</td>
<td>1.8 2.2</td>
</tr>
<tr>
<td><strong>Reverse Index</strong></td>
<td>S:100MB, M:500MB, L:1GB</td>
<td>1.5 0.9</td>
</tr>
<tr>
<td><strong>Kmeans</strong></td>
<td>S:10K, M:50K, L:100K points</td>
<td>1.2 1.7</td>
</tr>
<tr>
<td><strong>String Match</strong></td>
<td>S:50MB, M:100MB, L:500MB</td>
<td>1.8 1.5</td>
</tr>
<tr>
<td><strong>PCA</strong></td>
<td>S:500x500, M:1000x1000, L:1500x1500</td>
<td>1.7 2.5</td>
</tr>
<tr>
<td><strong>Histogram</strong></td>
<td>S:100MB, M:400MB, L:1.4GB</td>
<td>2.4 2.2</td>
</tr>
<tr>
<td><strong>Linear Regression</strong></td>
<td>S:50M, M:100M, L:500M</td>
<td>1.7 1.6</td>
</tr>
</tbody>
</table>
Evaluation

Speedup for different # of processors
Evaluation

Speedup for different # of processors

- WordCount, MatrixMultiply, StringMatch, LinearRegression key-based structure $\rightarrow$ significant speedups
- Kmeans, PCA, Histogram significant overheads due to unnatural 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

![Graph showing speedup for different dataset sizes](image)
Evaluation

Speedup for different dataset sizes

- Larger data sets allow the phoenix runtime to better amortize 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**

![Graph showing speedup comparison between Pthreads and Phoenix](image_url)
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 occurred)
    • <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 suboptimally for applications that are difficult to express with its model anyway...