Phoenix Rebirth: 
Scalable MapReduce on a Large-Scale 
Shared-Memory System

Richard Yoo, Anthony Romano, Christos Kozyrakis 
Stanford University

http://mapreduce.stanford.edu
Talk in a Nutshell

- Scaling a shared-memory MapReduce system on a 256-thread machine with NUMA characteristics

- **Major challenges & solutions**
  - Memory mgmt and locality => locality-aware task distribution
  - Data structure design => mechanisms to tolerate NUMA latencies
  - Interactions with the OS => thread pool and concurrent allocators

- **Results & lessons learnt**
  - Improved speedup by up to 19x (average 2.5x)
  - Scalability of the OS still the major bottleneck
Background
MapReduce and Phoenix

MapReduce

- A functional parallel programming framework for large clusters
- Users only provide map / reduce functions
  - Map: processes input data to generate intermediate key / value pairs
  - Reduce: merges intermediate pairs with the same key
- Runtime for MapReduce
  - Automatically parallelizes computation
  - Manages data distribution / result collection

Phoenix: shared-memory implementation of MapReduce

- An efficient programming model for both CMPs and SMPs [HPCA’07]
Phoenix on a 256-Thread System

- 4 UltraSPARC T2+ chips connected by a single hub chip
  1. Large number of threads (256 HW threads)
  2. Non-uniform memory access (NUMA) characteristics
     - 300 cycles to access local memory, +100 cycles for remote memory
The Problem: Application Scalability

- Baseline Phoenix scales well on a single socket machine
- Performance plummets with multiple sockets & large thread counts

Speedup on a 4-Socket UltraSPARC T2+
The Problem: OS Scalability

- OS / libraries exhibit NUMA effects as well
  - Latency increases rapidly when crossing chip boundary
  - Similar behavior on a 32-core Opteron running Linux
Optimizing the Phoenix Runtime on a Large-Scale NUMA System
Optimization Approach

- Focus on the unique position of runtimes in a software stack
  - Runtimes exhibit complex interactions with user code & OS

- Optimization approach should be multi-layered as well
  - **Algorithm** should be NUMA aware
  - **Implementation** should be optimized around NUMA challenges
  - **OS interaction** should be minimized as much as possible
Algorithmic Optimizations

Algorithmic Level

Implementation Level

OS Interaction Level
Algorithmic Optimizations (contd.)

Runtime algorithm itself should be NUMA-aware

- Problem: original Phoenix did not distinguish local vs. remote threads
  - On Solaris, the physical frames for `mmap()` ed data spread out across multiple *locality groups* (a chip + a dedicated memory channel)
  - Blind task assignment can have local threads work on remote data
Algorithmic Optimizations (contd.)

- **Solution: locality-aware task distribution**
  - Utilize per-locality group task queues
  - Distribute tasks according to their locality group
  - Threads work on their local task queue first, then perform task stealing
Implementation Optimizations

- Algorithmic Level
- Implementation Level
- OS Interaction Level

Diagram:
- App
- Phoenix Runtime
- OS
- HW
Implementation Optimizations (contd.)

Runtime implementation should handle large data sets efficiently

- Problem: Phoenix core data structure not efficient at handling large-scale data

- Map Phase
  - Each column of pointers amounts to a fixed-size hash table
  - `keys_array` and `vals_array` all thread-local

```
map thread id
num_map_threads

hash("orange")
num_reduce_tasks

2-D array of pointers
```

```
keys_array
“apple”
“banana”
“orange”
“pear”
```

- Too many keys
- Buffer reallocations

Yoo, Phoenix2

October 6, 2009
Reduce Phase

- Each row amounts to one reduce task
- Mismatch in access pattern results in remote accesses
Solution 1: make the hash bucket count user-tunable
  • Adjust the bucket count to get few keys per bucket
Solution 2: implement iterator interface to \texttt{vals\_array}

- Removed copying / allocating the large value array
- Buffer implemented as distributed chunks of memory
- Implemented prefetch mechanism behind the interface
Other Optimizations Tried

- **Replace hash table with more sophisticated data structures**
  - Large amount of access traffic
  - Simple changes negated the performance improvement
    - E.g., excessive pointer indirection

- **Combiners**
  - Only works for commutative and associative reduce functions
  - Perform local reduction at the end of the map phase
  - Little difference once the prefetcher was in place
    - Could be good for energy

- **See paper for details**
OS Interaction Optimizations

- Algorithmic Level
- Implementation Level
- OS Interaction Level
OS Interaction Optimizations (contd.)

Runtimes should deliberately manage OS interactions

1. Memory management => memory allocator performance
   • Problem: large, unpredictable amount of intermediate / final data
   • Solution
     ▪ Sensitivity study on various memory allocators
     ▪ At high thread count, allocator performance limited by sbrk()

2. Thread creation => mmap()
   • Problem: stack deallocation (munmap()) in thread join
   • Solution
     ▪ Implement thread pool
     ▪ Reuse threads over various MapReduce phases and instances
Results
Experiment Settings

- 4-Socket UltraSPARC T2+

- Workloads released in the original Phoenix
  - Input set significantly increased to stress the large-scale machine

- Solaris 5.10, GCC 4.2.1 –O3

- Similar performance improvements and challenges on a 32-thread Opteron system (8-sockets, quad-core chips) running Linux
Scalability Summary

- **Significant scalability improvement**

Scalability of the Optimized Version

- histogram
- kmeans
- linear_regression
- matrix_multiply
- pca
- string_match
- word_count

Yoo, Phoenix2

October 6, 2009
Execution Time Improvement

- Optimizations more effective for NUMA

Relative Speedup over the Original Phoenix

- Little variation
  - average: 1.5x, max: 2.8x

- Significant improvement
  - average: 2.53x, max: 19x
Analysis: Thread Pool

- **kmeans performs a sequence of MapReduces**
  - 160 iterations, 163,840 threads
- **Thread pool effectively reduces the number of calls to munmap()**

### Number of Calls to munmap() on kmeans

<table>
<thead>
<tr>
<th>threads</th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>16</td>
<td>1,947</td>
<td>13</td>
</tr>
<tr>
<td>32</td>
<td>4,499</td>
<td>18</td>
</tr>
<tr>
<td>64</td>
<td>9,956</td>
<td>33</td>
</tr>
<tr>
<td>128</td>
<td>14,661</td>
<td>44</td>
</tr>
<tr>
<td>256</td>
<td>14,697</td>
<td>102</td>
</tr>
</tbody>
</table>

**kmeans Performance Improvement due to Thread Pool**

3.47x improvement
Analysis: Locality-Aware Task Distribution

- Local group hit rate (% of tasks supplied from local memory)
- Significant locality group hit rate improvement under NUMA environment

- Forced misses result in similar hit rate

- Improved hit rates = improved performance

Locality Group Hit Rate on `string_matches`

- Locality group hit rate (% of tasks supplied from local memory)
- Significant locality group hit rate improvement under NUMA environment
No single hash table size worked for all the workloads

- Some workloads generated only a small / fixed number of unique keys
- For those that did benefit, the improvement was not consistent

Recommended values provided for each application
Why Are Some Applications Not Scaling?
Non-Scalable Workloads

Non-scalable workloads shared two common trends

1. Significant idle time increase
2. Increased portion of kernel time over total useful computation
Profiler Analysis

- **histogram**
  - 64% execution time spent idling for data page fault

- **linear_regression**
  - 63% execution time spent idling for data page fault

- **word_count**
  - 28% of its execution time in `sbrk()` called inside the memory allocator
  - 27% of execution time idling for data pages

- **Memory allocator and `mmap()` turned out to be the bottleneck**

- **Not the physical I/O problem**
  - OS buffer cache warmed up by repeating the same experiment with the same input
Memory Allocator Scalability

- `sbrk()` scalability a major issue
  - A single user-level lock serialized accesses
  - Per-address space locks protected in-kernel virtual memory objects
- `mmap()` even worse
**mmap() Scalability**

- **Microbenchmark:** `mmap()` user file and calculate the sum by streaming through data chunks

![Graph showing scalability of mmap()](Image)

- `mmap()` alone does not scale
- Kernel lock serialization on per process page table
Conclusion

- Multi-layered optimization approach proved to be effective
  - Average 2.5x speedup, maximum 19x

- OS scalability issues need to be addressed for further scalability
  - Memory management and I/O
  - Opens up a new research opportunity
Questions?

- The Phoenix System for MapReduce Programming, v2.0
  - Publicly available at http://mapreduce.stanford.edu