Evaluating MapReduce for Multicore and Multiprocessor Systems

Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, Christos Kozyrakis

Computer Systems Laboratory
Stanford University
Google’s MapReduce

- A general-purpose environment for large-scale data processing
  - Programming model (API) and runtime system for large clusters
  - Functional representation of data parallel tasks

- Easy to use & very successful within Google
  - Indexing system, distributed grep & sort, document clustering, machine learning, statistical machine translation, …

- MapReduce supports
  - Automatic parallelization and distribution
    - Abstracts parallelization, synchronization, and communication issues
  - Fault tolerance
    - Task monitoring and replication
  - I/O scheduling
WordCount Example [OSDI’04]

// input: a document
// intermediate output: key=word; value=1

Map(void *input) {
    for each word w in input
        EmitIntermediate(w, 1); // <key, val> intermediate pairs
}

// intermediate output: key=word; value=1
// output: key=word; value=occurrences

Reduce(String key, Iterator values) {
    int result = 0;
    for each v in values
        result += v;
    Emit(key, result); // Final output sorted by key
}
The Phoenix System

Question: is MapReduce applicable to multicore programming?
- What is the performance?
- Is the performance scalable & portable?
- Does it help with locality and fault management?
- How does it compare to other parallel programming approaches?

Phoenix: a shared-memory implementation of MapReduce
- Uses threads instead of cluster nodes for parallelism
- Communicates through shared memory instead of network messages
  - Works with CMP and SMP systems
- Current version works with C/C++ and uses P-threads
  - Easy to port to other languages or thread environments
The Phoenix API

- **System-defined functions**
  - int `phoenix_scheduler` (scheduler_args_t *args)
    - Initializes the runtime system
  - void `emit_intermediate` (void *key, void *val, int key_size)
  - void `emit` (void *key, void *val)

- **User-defined functions**
  - void (*`map_t`) (map_args_t *args)
    - Map function applied on each input element
  - void (*`reduce_t`) (void *key, void **buffer, int count)
    - Reduce function applied on intermediate pairs with same key
  - int (*`key_cmp_t`) (const void *key1, const void *key2)
    - Function that compares two keys
  - int (*`splitter_t`) (void *input, int size, map_args_t *args)
    - Splits input data across Map tasks (optional)

Simple & narrow API. Similar to Google’s API
The Phoenix Runtime

- Orchestrates program execution across multiple threads
  - Initiates and terminates threads (workers)
  - Assigns map & reduce tasks to workers
  - Handles buffer allocation and communication

- Key runtime features
  - Dynamic scheduling of tasks for load balancing
  - Communication through pointer exchange (when possible)
  - Locality optimization through granularity adjustment
  - Support for failure recovery

- Details of parallel execution are hidden from programmer
  - Low-level threading, communication, scheduling, …
Phoenix Execution Overview

Map Stage

Worker 1

Reduce Stage

Worker 1

Merge Stage

Worker 1

Input

Split

Worker N

Worker M

Worker L

Output

C. Ranger et al, HPCA’13, February 2007
Input Data Splitting

- Divides input data to chunks for map tasks
  - Small chunk → map overhead; large chunk → locality issues
- Phoenix: chunk size determined by cache size for locality
  - Or programmer provides custom splitter
Map Stage

- Each task applies map function to an input chunk
  - Phoenix: typically 100s of tasks multiplexed to available workers
  - No reduce tasks are started before map tasks complete
- Intermediate pairs partitioned to reduce queues based on keys
  - Partitioning can introduce significant overhead!
Reduce Stage

- Each task processes a key set from the reduce queues
  - Dynamic scheduling used for reduce tasks as well
- A poor partition function can lead to significant imbalance
  - Default partition function based on key hashing
  - Programmer can provide custom partition function
Merge Stage

- Combines reduce output queues to single sorted output
  - May be unnecessary for some applications
  - But merge time tends to be small compared to map/reduce time
- Phoenix: binary merging of reduce queues into single queue
  - Overhead increases with number of reduce tasks
Potential Performance Detractors

- **Significant detractors**
  - **Partitioning overhead**: communication and grouping requirements
  - **Model overhead**: particularly due to calls to emit/emit_intermediate
  - **Key management**: some apps do not naturally associate keys with data
  - **Repeated Map/Reduce invocations**: necessary for some apps

- **Practically insignificant issues**
  - **Final merging & sorting**: insignificant compared to other tasks
  - **Buffer management**: extensive (re-)use of pre-allocated buffers
  - **Reduce imbalance**: handled through dynamic scheduling
  - **Serialized input splitting**: most map tasks involve non-trivial work
Phoenix Fault Tolerance

- **Focus:** transient or permanent errors in workers

- **Error detection:** worker time-out
  - Execution time of similar tasks used as yardstick

- **Error recovery**
  - Restart or potentially re-assign affected tasks
  - Handle input/output buffer management

- **Future work**
  - Fault tolerance for the scheduler
  - Error detection and isolation through worker sandboxing
Evaluation Methodology

- **Shared-memory systems**
  - CMP: Niagara-based Sun Fire T1200 (8 CPUs, 4 threads/CPU)
  - SMP: Sun Ultra E6000 (24 CPUs)
    - SMP results similar to CMP → portable performance
    - See paper for details

- **8 applications**
  - Domains: enterprise, scientific, consumer
  - Three code versions: sequential, MapReduce (Phoenix), P-threads
    - Optimized independently

- **Experiments**
  - **Talk**: performance & scalability, Phoenix Vs. P-threads
  - **See paper for**: dependency to data-set size, dependency to input task granularity, (soft & hard) fault injection experiment
Applications

- **Word count** – determine frequency of words in documents
- **String match** – search file with keys for an encrypted word
- **Reverse Index** – build reverse index for links in HTML files
- **Linear regression** – find the best fit line for a set of points
- **Matrix multiply** – dense integer matrix multiplication
  - MapReduce version introduces coarse-grain coordinate variables
- **Kmeans** – clustering algorithm for 3D data points
  - Multiple MapReduce invocation with translation step
- **PCA** – principal component analysis on a matrix
  - MapReduce version introduces coordinate variables
- **Histogram** – frequency of RGB components in images
  - There is no need for keys in original algorithm
Good scalability across all applications
- Absolute speedup depends on importance of various detractors
- Note for CMP: 1 core = 4 threads (4 workers)

Improved locality leads to significant improvements for some apps

At high core counts: some bandwidth saturation or load imbalance
Phoenix Vs. P-threads

- Phoenix equal to P-threads if algorithm matches MapReduce model
  - Note that P-threads’ low-level API is more flexible
- Anecdote: we looked at Phoenix behavior to tune some Pthreads codes
- P-threads is better for algorithms that do not fit MapReduce model
  - Does not use keys, requires multiple MapReduce iterations, …
Conclusions

- **Phoenix: a shared-memory implementation of MapReduce**
  - MapReduce API and runtime system for C/C++
    - Uses threads instead of cluster nodes for parallelism
    - Communicates through shared memory instead of network messages
    - Dynamic scheduling, locality management, fault recovery, …
  - Scalable & portable performance, compares well to P-threads

- **Future work**
  - Improve queue structures
  - Automatic configuration detection
  - Better fault detection
  - Experiment with more systems & apps
Questions?

- Want a copy of Phoenix?
  - Will post the source code at http://csl.stanford.edu/~christos