Optimizing MapReduce for Multicore Architectures

Yandong Mao Robert Morris M. Frans Kaashoek
Massachusetts Institute of Technology, Cambridge, MA

Abstract

MapReduce is a programming model for data-parallel programs originally intended for data centers. MapReduce simplifies parallel programming, hiding synchronization and task management. These properties make it a promising programming model for future processors with many cores, and existing MapReduce libraries such as Phoenix have demonstrated that applications written with MapReduce perform competitively with those written with Pthreads [11].

This paper explores the design of the MapReduce data structures for grouping intermediate key/value pairs, which is often a performance bottleneck on multicore processors. The paper finds the best choice depends on workload characteristics, such as the number of keys used by the application, the degree of repetition of keys, etc. This paper also introduces a new MapReduce library, Metis, with a compromise data structure designed to perform well for most workloads. Experiments with the Phoenix benchmarks on a 16-core AMD-based server show that Metis' data structure performs better than simpler alternatives, including Phoenix.

1 Introduction

MapReduce [3] helps programmers to write certain kinds of data-parallel applications. The programmer writes a Map and a Reduce function that must obey some restrictions, and in return the MapReduce library can run the application in parallel automatically: the programmer does not have to write code for synchronization or to manage parallel tasks. MapReduce was initially proposed for applications distributed over multiple computers, but the Phoenix library [11] shows that MapReduce can help applications on a single multicore machine perform competitively with hand-crafted Pthreads applications. This paper describes a new MapReduce library for multicore processors that achieves better performance than Phoenix by using better data structures for grouping the key/value pairs generated by Map and consumed by Reduce.

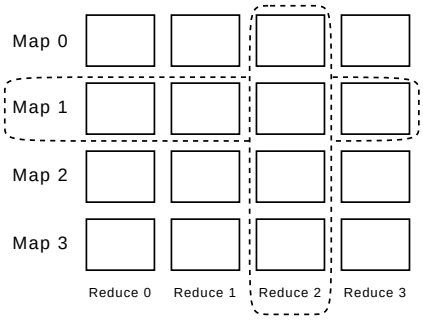

Figure 1: A map computation produces a row of key/value pairs. A reduce computation consumes a column of key/value pairs.

The focus of this paper is MapReduce applications which involve a relatively large number of intermediate key/value pairs and a relatively low amount of computation, that is, situations in which the run time is not dominated by the application code in Map and Reduce, but by the overhead of the library itself. Word count and creating an inverted index are example applications for which the design of the library matters, while matrix multiply isn't. The paper further restricts itself to situations in which the inputs and outputs are in memory (i.e., in the file system buffer cache) so that the network and disk systems are not bottlenecks. This configuration is one that we use for sorting and counting data generated by experiments in other research projects. Even with these restrictions, engineering a MapReduce library for multicore processors is surprisingly challenging.

A MapReduce library organizes an application into three phases: Map, Reduce, and Merge. The library partitions the input into a number of "splits," and concur-

rently runs a programmer-supplied Map function on each split. Each instance of Map produces intermediate output in the form of a set of key/value pairs (see Figure 1). When the Maps have completed, the MapReduce library invokes a programmer-supplied Reduce function, once for each distinct key generated by the Map functions, supplying each instance of Reduce with all the values generated for the corresponding key. Each Reduce generates a set of key/value pairs as output. The different Reduce instances can execute concurrently. When the Reduces have finished, the library enters a Merge phase to sort the aggregated output of the Reduces by key; the sort generates the final output of the application.

The core challenge that we encountered in engineering a MapReduce library is the organization of MapReduce intermediate data—the matrix in Figure 1. The organization of the Map output is critical to the performance of many MapReduce applications, since the entire body of intermediate data must be reorganized between the Map and Reduce phases: Map produces data in the same order as the input, while Reduce must consume data grouped by key. In a data center this operation is dominated by the performance of the network, but when running on single multicore processor the performance is dominated by the operations on the data structure that holds intermediate data.

A strategy for storing intermediate MapReduce data must consider multiple factors in order to perform well on multicore processors. Concurrent Map instances should avoid touching the same data, to avoid locking and cache contention costs. Inserting key/value Map output should be fast, and for many applications looking up existing keys should also be fast in order to support quick coalescing of Map output for the same key. The Reduce phase must consider all instances of a given key at the same time, so ideally those instances would be grouped together in the Map output. Finally, the Reduce invocations would ideally generate output in an order consistent with that required by the overall application, to minimize Merge cost.

Different MapReduce applications stress different performance aspects of the intermediate data structure. In particular, the application Map function's tendency to repeat the same key in its output governs the relative importance of the data structure's lookup and insert operations and the ability of the data structure to group key/value pairs with the same key.

This paper presents a new MapReduce library, called Metis, whose intermediate data structure is a hash table with a b+tree in each entry. In the common case Metis can exploit the hash table's $O(1)$ operation costs and avoid use of the b+trees. For workloads with unexpected key distributions, Metis falls back on the b+tree's $O(\log n)$ operations. The result is competitive performance across a wide range of workloads.

We have implemented Metis in C on Linux and Windows. The implementation includes multiple optimizations: support for Map output combiners, a lock-free scheme to schedule map and reduce work, immutable strings for fast key comparisons, and use of the scalable Streamflow [13] memory allocator. Some of these optimizations are obvious (e.g., using a combiner function, which the Google MapReduce library already has, but Phoenix doesn't), but others took us a while to discover (e.g., the fact that commonly-used memory allocators are not scalable and that this property is important in MapReduce applications).

An evaluation on 16 cores of standard MapReduce benchmarks shows that Metis achieves equal or better performance than Phoenix, and that for some applications this improvement is an order of magnitude. This improvement makes the most difference for applications such as inverted index creation that produce large numbers of intermediate key/value pairs. Metis can achieve good performance without manual parameter tuning, which Phoenix requires.

The main contribution is how to engineer a MapReduce library that runs efficiently on a commodity multicore computer. More specifically, the contributions are: 1) the observation that the data structure used to group key/value pairs is a primary performance bottleneck on multicore processors; 2) a new MapReduce library, Metis, that performs this grouping efficiently without application-specific tuning; 3) an evaluation showing that Metis is more efficient than Phoenix; and 4) a set of experience tidbits from running MapReduce applications on different operating systems and processors, and experimenting with optimizations.

The rest of the paper is organized as follows. Section 2 provides a brief description of MapReduce from a parallel programming perspective. Section 3 describes the Metis design. Section 4 summarizes the salient implementation details. Section 5 evaluates Metis versus Phoenix running on a machine with AMD processors and provides an analysis of the benefits of Metis design decisions. Section 6 summarizes our experience with Metis and MapReduce. Section 7 relates our work to previous work. Section 8 summarizes our conclusions.

2 Background

This section introduces the MapReduce model and outlines the design of Phoenix, an existing multiprocessor MapReduce library.

2.1 MapReduce Programming Model

A MapReduce library requires the programmer to cast the computation in the form of two functions, Map and Reduce. The library partitions the input into a number of "splits", and calls Map on each split. The library typically has a pool of Map worker threads, one per core, that repeatedly take a split from a work list and call Map. Each Map call's output is a set of "intermediate" key/value pairs. When all the splits have been processed by Map, the library calls Reduce once for each distinct key produced by the Map calls, passing Reduce the set of all values produced by the Maps for that key. Again, the library has a pool of Reduce worker threads, one per core. Each Reduce generates a set of output key/value pairs, and the library's Merge phase sorts them by key to produce the final output. The programmer must ensure that the Map and Reduce functions have no side effects.

The charm of MapReduce is that, for algorithms that can fit that form, the library hides all the concurrency from the programmer. For example, one can count the number of occurrences of each word in a body of text as follows. The WordCount Map function parses the input, generating an intermediate key/value pair for each word, whose key is the word and whose value is 1. The Reduce function (which the library calls once per distinct word) emits a key/value pair whose key is the word, and whose value is the number of values. The library gets parallel speedup by running many Maps concurrently, each on a different part of the input file, and by running many Reduces concurrently on different words. The Merge phase combines and sorts the outputs of all the Reduces.

2.2 Phoenix

Phoenix is a MapReduce library for multi-core and multi-processor systems. Phoenix stores the intermediate key/value pairs produced by the Map calls in a matrix. The matrix has one row per split of the input, and a fixed number of columns (256 by default). Each row acts as a hash table with an entry per column. Phoenix puts each key/value pair produced by a Map in the row of the input split, and in a column determined by a hash of the key. Each Phoenix Reduce worker thread processes an entire column at a time.

This arrangement limits contention in two ways. First, different Map workers write their intermediate results to different areas of memory. Second, each Map worker partitions its results according to the Reduce thread that will consume each result. The result is that the Map workers do not contend to produce output, and the Reduce workers contend to find work on column granularity. This arrangement parallelizes a sort that groups keys well.

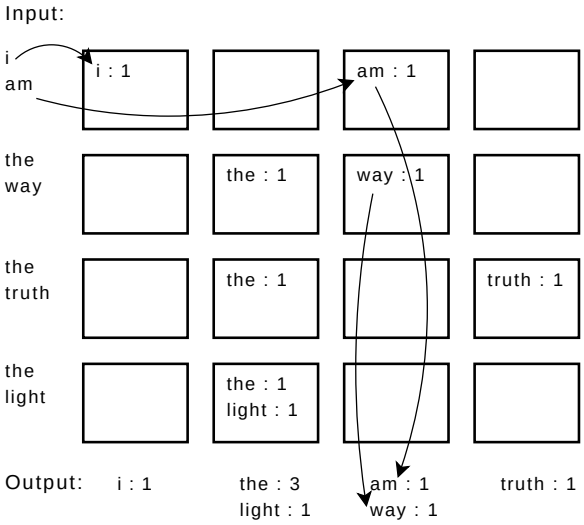

Figure 2: Example of the Phoenix WordCount application processing the input "I am the way, the truth, the light."

Figure 2 shows an example of how data flows through a Phoenix-based WordCount application. The horizontal arrows represent a Map worker hashing intermediate key/value pairs into its row's columns, chosen with a hash of the key. The vertical arrows represent a Reduce worker reading the hash table entries assigned to it. Note that all the instances of "the" are processed by the same Reduce worker.

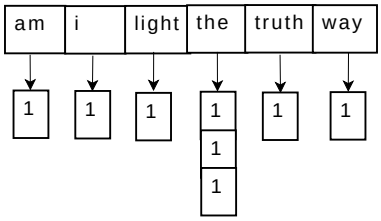

Figure 3: Example of a Phoenix hash table entry filled with the output of WordCount's Map for the input "I am the way, the truth, the light."

Inside each entry of a hash table, Phoenix stores the key/value pairs in an array sorted by key (see Figure 3). An entry's array has at most one element for each distinct key; if there is more than one pair with the same key, the values are "grouped" into a list attached to that key. This grouping is convenient for the Reduce phase, since the Reduce function takes all the values for a key as an argument. The cost for Map to insert a key/value pair depends on whether the key is already in the relevant hash table entry's list. If it is, the cost is $O(1)$ to hash to

the correct column, $O(\log k_b)$ to find the existing key in the array (assuming k_b distinct keys in the hash table entry), and $O(1)$ to insert the new value in the key's list. If the key is not already in the table, the cost is $O(1)$ to hash to the correct column and $O(k_b)$ to move array entries down to make room for the new key. The result is that, for workloads with many keys, the linear insert cost dominates total run-time.

3 Design

This section explores the main challenges in multicore MapReduce performance and describes a design, called Metis, that addresses those challenges. Metis inherits much of its design from Phoenix but differs in a number of aspects to achieve better performance on multicore processors.

3.1 The Problem

The core problem that a MapReduce library solves is conveying the output of the Map phase to the Reduce phase, by way of the intermediate data structure holding the Map output. The library's design must trade off among the following requirements for the intermediate data structure:

- The different Map threads should not modify the same data structures in order to avoid locking and cache line contention costs. This suggests that there should be a separate output data structure for each Map thread.
- Some workloads cause Map to produce many key/value pairs with the same key. For such workloads, the intermediate data structure must efficiently group together all pairs with the same key, in order to later pass them together to Reduce.
- If there are many keys, the Reduce threads may benefit from partitioning the keys among themselves in bulk to avoid scheduling work per key. The set of keys a given Reduce thread is responsible for would ideally be grouped together, in the same order, in the outputs of all the Maps.
- The Merge phase will run fastest if the Reduces generate output in the same order as that required for the application's final output, and thus if each Reduce thread processes keys in the same order as the final output.
- The data structure must allow for load-balancing the Map and Reduce work over the worker threads.

The best trade-off among these requirements depends on the properties of the application and its input (the "workload"). The following are the main relevant properties of the workload:

- Number of pairs: the MapReduce library's intermediate data structure affects performance only for workloads in which Map emits many key/value pairs.
- Key repetition: a large number of Map output pairs may involve many distinct keys, each of which appears only a few times, or a few keys, each of which appears many times (typically in the outputs of many Map threads). If there are relatively few repeated keys, the main operation the intermediate data structure must support is partitioning the keys for consumption by Reduce workers. If there are many repeated keys, it is beneficial for the output data structure to group pairs with the same key efficiently, so a key lookup operation is critical for good performance.
- Large output: some workloads have a Reduce phase that emits many items, so that the final Merge has significant cost. For these workloads, Reducing keys in final output order may reduce the cost of the Merge sort. For these workloads it may be beneficial for the intermediate data structure to store keys in sorted order.
- Predictability: some workloads generate roughly the same distribution of Map output key frequencies for all parts of the Map input, while others have key statistics that vary in different parts of the input. Predictability helps when the intermediate data structure must be sized in advance (for example, for a hash table).
- Load balance: some workloads involve roughly the same amount of Reduce processing for each key, while others have skewed amounts of Reduce work per key. The MapReduce library must try its best to load-balance the latter workloads.

3.2 Design Options

To provide comparison points for Metis' intermediate data structure, consider the following possible ways to store Map output. Each one of them works well for certain workloads.

Each Map thread could store its output in a separate **hash table**. All the hash tables should be the same size, and all Map threads should use the same hash function, so that the Reduce threads can easily partition the work. Each hash table entry would contain a linked list

of key/value pairs whose keys are hashed to that entry. If the hash tables have enough entries, collisions will be rare and the lists will be short, so that lookup and insert will have cost $O(1)$ (i.e., be independent of the number of keys). However, this property requires that the hash table have enough entries to avoid collisions, which implies that the MapReduce library be able to predict the total number of keys. The hash table's $O(1)$ lookups make it particularly attractive for workloads with many repeated keys.

Each Map thread could store its output by appending each key/value pair to an **append-only buffer**, and then sorting that buffer by key at the end of the Map phase. The Reduce threads would have to inspect the buffers to agree on a partition of the keys. Sorting the buffer all at once has better locality than (for example) inserting many keys into a tree. The sort, however, has time determined by the total number of pairs, rather than unique keys, so an append-only buffer is mainly attractive for workloads that have few repeated keys.

Each Map thread could store its output in a **tree** indexed by key. The Reduce threads would have to inspect the trees to agree on a partition of keys. A tree has reasonably fast lookup and insert times ($O(\log k)$, where k is the number of keys), and does not require prediction of the number of keys. Thus a tree is attractive for workloads with repeated keys and with unpredictable number of keys.

The append-only buffer and tree data structures have the added attraction that they sort the pairs by key, and may allow the Reduce threads to process the pairs in key order, thus producing output in key order and reducing or eliminating the time required in the Merge phase to sort the output. A hash table, in contrast, does not naturally order the keys.

3.3 Metis

No one of the data structures mentioned above works well for all workloads. A hash table performs poorly when the number of keys is hard to predict. An append-only buffer performs poorly with many repeated keys. A tree's $O(\log k)$ operations are slower than a hash table's $O(1)$ operations for predictable workloads.

Metis uses a "hash+tree" data structure to get the good properties of both a hash table and a tree. The hash+tree data structure consists, for each Map thread, of a hash table with a separate b+tree in each entry of the hash table (see Figure 4). The hash table has a fixed number of entries, chosen after a prediction phase (see Section 3.4) to be proportional to the predicted total number of distinct keys. All the Map threads use the same size hash table and the same hash function. The hash table's $O(1)$ lookups support efficient grouping of pairs

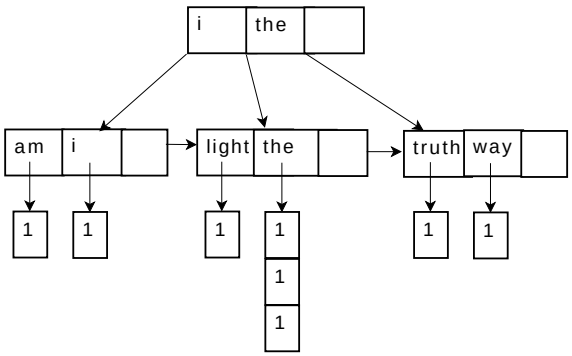

Figure 4: Example of the b+tree in a Metis hash entry filled with the input "I am the way, the truth, the light." from the WordCount application.

with the same key since an existing key can be found quickly. When Map emits a key/value pair, the MapReduce library hashes the key to find the hash table entry, then uses the key to descend the b+tree in that entry. If a b+tree entry for the key already exists, the library appends the new value to the existing value(s) in the entry. Otherwise the library creates a new b+tree entry for the new key/value pair.

When all the Map work is finished, the library partitions the hash table entries. Each Reduce thread repeatedly picks an unprocessed partition and processes the same range of hash table entries in all the Maps' outputs. Small partitions allow better load balance and incur little overhead.

For the Merge phase Metis uses a scalable sorting algorithm: Parallel Sorting by Regular Sampling (PSRS) [14]. The run-time of PSRS is $O(\frac{k}{c} \log k)$, where c is the number of cores and k is the number of Reduce output items. The reason that PSRS scales linearly is that it ignores the bucket boundaries, so the number of buckets doesn't affect its run-time. The advantage that PSRS has over mergesort is that PSRS can use all cores for the whole sort, while mergesort leaves cores idle during its final phase.

The hash+tree data structure works well for many workloads. If the number of keys is predictable and they hash uniformly, then the hash table will have enough entries to avoid more than a constant number of collisions and b+tree entries per hash entry, and both inserts and lookups will take $O(1)$ time. If the prediction is too small or the keys don't hash uniformly, so that some or all hash table entries contain many keys, then the b+trees in the hash table entries may grow large. In that case the lookup and insert costs will be $O(\log k)$—that is, no worse than using a single tree per Map thread. If the prediction is too large, the overhead mainly comes from creating and

destroying the hash table, which is small compared with the total run-time.

The only situation where hash+tree is not attractive is workloads with few repeated keys. For these workloads, an append-only buffer would have better performance for two reasons. First, quicksort on the append-only buffer has better locality than inserting into a b+tree. Second, the Merge phase is more efficient for an append-only buffer than for hash+tree, because the Reduce output is already sorted. However, section 5 shows that Metis sacrifices little performance by using hash+tree rather than append.

Map	Reduce	Merge
$O(\frac{p}{c})$ or $O(\frac{p}{c} \log k)$	$O(\frac{p}{c})$	$O(\frac{k}{c} \log k)$

Figure 5: Computational complexity of Metis.

Figure 5 summarizes the computational complexity of Metis. p is the number of pairs of Map output, k is the number of distinct keys of Map output, and c is the number of cores. For the Map phase, the first item is the computational complexity when the prediction succeeds to limit the collisions to a constant. The second item is the computational complexity when Metis must make significant use of the b+trees in the hash table entries.

3.4 Prediction

To ensure $O(1)$ lookup and insert cost for the hash part of its hash+tree data structure, Metis must ensure that the number of keys per hash entry remains constant as the total number of keys grows. Metis does this by predicting the total number of distinct keys that a typical Map will generate (k_p), and creating each per-Map hash table with a number of entries proportional to k_p.

To predict k_p, Metis runs Map over the first 7% of the input file and finds the number n of distinct keys that Map produced. It then computes k_p as $\frac{n}{0.07}$. Metis sets the size of each per-Map-thread hash table to $0.1 * k_p$, so that each hash table entry is likely to contain 10 keys.

4 Implementation

We have implemented Metis in C on Linux and Windows. The application calls the library start function with the following parameters: the name of the input file, and the names of the functions that implement Map, Reduce, and Merge. Merge performs the final sort of the output of all the Reduces.

The start function creates one POSIX thread per core, and assigns it to a core using *sched_setaffinity*. Each thread processes Map input splits until there is no more data in the input file. When all the input splits have been processed, the threads call Reduce for each key in the Map output. When all keys have been Reduced, the threads Merge the Reduce output to produce the final output.

Metis employs no locks. Each Map input split has dedicated memory (a row in Figure 2) into which it can emit data. To assign Map output to Reduce threads, Metis maintains a column counter, which is initially zero. Each time a thread looks for more Reduce work, it increases the column counter by one using an atomic add instruction. Each value of the column counter corresponds to a range of Map output hash table entries for the Reduce thread to process.

Based on many hours of experimenting with applications, discovering scalability bottlenecks, and tracking down their sources, we refined the implementation in the following ways:

Lock-free splitting MapReduce application splits the input file into chunks (one per Map Task). The splitting can either be determined by the application or Metis. Applications can use algorithms to exploit data locality. In this case, the applications choose the split size to meet the requirement of the algorithm. For example, in MatrixMultiply, with a block-based algorithm (which comes with Phoenix single-core implementation), the locality can be exploited and the performance can be improved by 95% percent.

For applications that scan each chunk only once, locality of each chunk is not critical. In this case, the chunk size can be determined by Metis. Metis' Map threads allocate work (splits of the input file) by atomically incrementing an input file offset counter by the split size.

Memory allocator The three stages (Map, Reduce, and Merge) all require allocating, reallocating and freeing many chunks of memory. Multiple worker threads may concurrently issue memory requests. Thus, an efficient memory management scheme is important for good application performance. After some experiments, we used Streamflow [13], which is an efficient and scalable memory allocator for multi-threaded programs. It performs synchronization-free operations in most cases for local allocation and deallocation.

Keycopy function When a key/value pair is emitted during the Map phase, if the key is a pointer into the memory, the application must guarantee that the memory is not freed until the completion of MapReduce library. This is unnecessary for a key which already appeared; furthermore, keeping the memory around limits the size of the input file to the size of virtual memory. Worse, when the input file is large enough that it incurs paging, a key comparison may trigger a read to the disk, which is significantly slower compared with in-memory comparison.

```
void *keycopy(void *key, size_t len);
```

Application	Description	Keys	Pairs per key
MatrixMultiply(MM)	Multiply two integer matrices	0	0
PCA	Principle component analysis on a matrix	Input matrix	1
InvertedIndex(II)	Build reverse index for words	Unique words	# words
WordCount(WC)	Count occurrence of Words	Unique words	# words
StringMatch(SM)	Find an encrypted word from files with keys	4	# map input splits
LinearRegression(LR)	Calculate the best fit line for points	5	# map input splits
Histogram (Hist)	Compute frequency of image components	$3X256$	# map input splits
Kmeans	Classifying 3D points into groups	K	Points/K

Figure 6: The benchmarks, with input sizes and number of distinct keys and pairs in the Map output.

Metis provides a keycopy function, which copies the key on its first occurrence. As the above prototype shows, the keycopy function receives the key to copy and the length of the key, and returns the address of a new key. In this way, the processed input can be freed so that Metis can handle input files larger than virtual memory size, and the key comparison is done in the memory.

Combiner function The hash entries holding intermediate key/value pairs may have many values per key. Many values increase the memory footprint and place pressure on hardware caches. Metis, like the Google MapReduce library, allows an application to provide an optional Combiner function. The Combiner decreases the memory pressure by reducing the values corresponding to the same key as early as possible. Without a Combiner function, the library needs to keep these values until the Reduce phase. The Reduce function in many applications, however, does not necessarily need all values of a key before being able to apply the Reduce function.

```
size_t combiner(void *key, void **vals,
size_t len);
```

In Metis, Combiner is usually identical to the Reduce function in the Reduce phase. There are two possible places to apply the Combiner when merging the values of the same key: (1) Inserting key/value pairs in the Map phase; (2) Merging all key/value pairs with the same key emitted by all Map worker threads. Currently, Metis applies the Combiner only in the Map phase. Since there are only a few key/value pairs after applying the Combiner in the Map phase, the memory pressure in Reduce phase is not intensive.

Output compare function The output of Phoenix is an array of key/value pairs sorted by key. If the user requires the output sorted by fields other than keys, an additional MapReduce iteration must be executed to sort the output. For example, in the output of WordCount, the key is word and the value is the occurrences of the word in the file. After that, another MapReduce iteration is used to sort the output by occurrences.

```
typedef struct {
    void *key;
    void *val;
} keyval_t;
int outcmp(keyval_t *kv1, keyval_t *kv2);
```

Metis allows the application to provide an optional comparison function for the final output, as shown above. After the key/value pairs are reduced, the Reduce phase and Merge phase use the output comparison function to sort the output pairs, which reduces two MapReduce iterations to one for applications that don't require the output to be sorted by keys.

5 Evaluation

This section evaluates Metis' performance by exploring five issues. First, it demonstrates that Metis provides good parallel speedup as the number of cores increases. Second, it shows that Metis' absolute performance is better than that of Phoenix for workloads where the MapReduce library's performance matters. Third, it investigates the reasons for Metis' good performance, focusing on the interaction between workload and Map output data structure, and showing that Metis' hybrid strategy performs as well as the best individual data structure for each kind of workload. Fourth, it measures the accuracy and overhead of Metis' prediction algorithm. Finally, we compare the single-core performance of Metis and Hadoop for completeness.

5.1 Experimental Method

The measurements are taken on an AMD 16-core machine with 64 Gbyte memory running Linux with kernel version 2.6.25. The machine has four quad-core 2.0 GHz AMD chips. Each of the 16 cores has a 64 KByte L1 data cache, and 512KByte L2 cache. The four cores on each chip share a 2 MByte L3 cache. All caches have a block size of 64 bytes. All the software uses the 64-bit instruction set.

The benchmark applications are derived from those supplied by Phoenix, and summarized in Figure 6. For

StringMatch, LinearRegression and Histogram, the number of keys is fixed and the number of pairs per key depends on the number of map input splits. The results for sections other than 5.4 use the workloads described in Figure 7. These are chosen to ensure the applications run for a moderately long time, but still fit in memory. The inputs are in the in-memory buffer cache.

Application	Input size	Keys	Pairs
MM	2048X2048	0	0
PCA	2048X2048	2.1 M	2.1 M
II	100 MB	9.5 M	13.7 M
WC	300 MB	1 M	51 M
SM	1 GB	4	1024
LR	4 GB	5	1280
Hist	2.6 GB	768	197 K
Kmeans	Points = 5M, K = 16	16	5 M

Figure 7: The input size, number of Key and Key/Value Pairs of the inputs. M stands for million, and K stands for thousand.

All experiments run with 256 Map input splits, except for MatrixMultiply. MatrixMultiply uses a blocked-based algorithm, which splits the input matrix into 4175 blocks to exploit locality. The performance of other benchmarks is not sensitive to the number of map input splits.

5.2 Parallel Speedup

Figure 8 shows the speedup with increasing number of cores for each benchmark, when using Metis. The speedup is the run-time of an optimized non-parallel version divided by the run-time for the application on the indicated number of cores. The optimized non-parallel version of each benchmark does not use MapReduce, but is a separately written dedicated application. The "1 core" bar for each benchmark indicates performance of the Metis-based benchmark, and is typically less than 1.0 because the optimized non-parallel version is faster than the Metis-based version.

Histogram, LinearRegression, StringMatch, and MatrixMultiply have run-times that are dominated by the application logic, not the MapReduce library. This can be seen in Figure 9, which shows the fraction of single-core run-time spent in the library vs the application. StringMatch and MatrixMultiply achieve a good speedup because the application code is highly concurrent. Histogram and LinearRegression scale well until 8 cores and then drop. The scalability is limited by a lack of concurrency in Linux's page fault handler, which is triggered as the Map phase reads the memory-mapped input file. This problem could be fixed with address ranges [1].

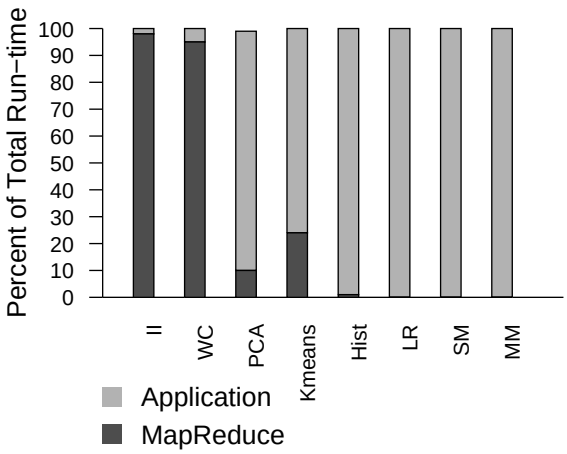

Figure 9: Percent of total run-time spent in the application and in the MapReduce library for the benchmarks, measured with Metis on a single core using performance counters. InvertedIndex and WordCount spend almost all of the time in the MapReduce library manipulating Map output and sorting Reduce output.

Figure 8 shows poor speedup for PCA and Kmeans because they do not fit will into the MapReduce model.

InvertedIndex and WordCount are of the most interest because they spend most of their time in the MapReduce library, as Figure 9 shows. Both achieve more than half-linear parallel speedup on 16 cores. For InvertedIndex, the three phases scale to 13.1x, 8.2x and 13.9x on 16 cores, while WordCount's phases scale to 10.8x, 1.4x, and 9.8x. Both are limited by the poorly scaling Reduce phase. For InvertedIndex the reason is that the Reduce phase copies and merges values from different Map workers, and thus may suffer from Linux' unscalable page fault handler [1]. For WordCount, the run-time of the Reduce phase is quite short with 16 cores, and the overhead of assigning the large number of hash table entries to the Reduce threads may dominate the performance.

5.3 Metis versus Phoenix

Figure 10 compares the throughput of applications using Metis to throughput with Phoenix. All the numbers are normalized to Phoenix, so Phoenix has a throughput of one. InvertedIndex and WordCount are much faster with Metis than with Phoenix, because both produce a large number of Map output pairs which Metis is more efficient at handling. The other benchmarks have about the same performance for Metis and Phoenix because they spend most of their time in application code.

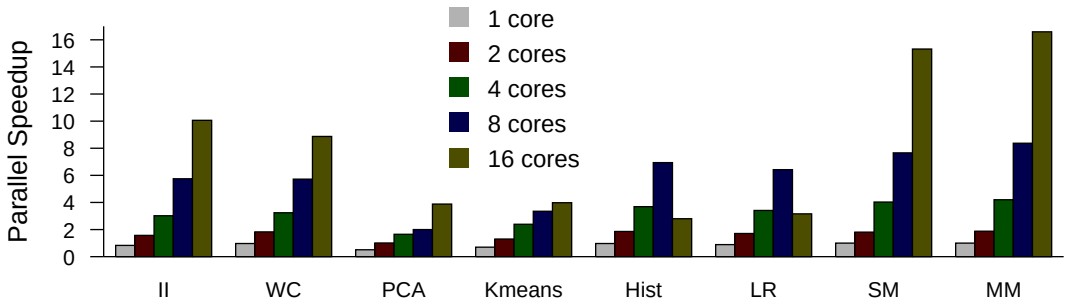

Figure 8: Parallel speedup of the benchmarks with Metis, relative to optimized non-parallel versions of the applications.

Implementation	InvertedIndex	WordCount	PCA	KMeans
Phoenix	31.7	65.3	5.3	3.1
Phoenix+Ext	25.2	51.1	4.2	2.6
Phoenix+Ext+Tuning	12.3	9.9	4.0	2.6
Metis	5.3	5.9	3.7	2.7

Figure 11: Run-time in seconds of InvertedIndex, WordCount, PCA, and KMeans for different implementations on 16 cores. Phoenix+Ext indicates Phoenix with the StreamFlow, KeyCopy, OutCmp, and Combiner optimizations. Phoenix+Ext+Tuning also includes tuning of the number of Reduce tasks for best performance.

In order to explain the difference in performance between Phoenix and Metis, Figure 11 shows how Phoenix's run-time improves as Metis' optimizations are added. The Phoenix+Ext line shows the effect of adding the optimizations from Section 4: a modest increase in throughput, mostly due to Combiner functions and StreamFlow.

The Phoenix+Ext+Tuning line shows the improvement when the number of Phoenix's Map output hash table entries is chosen to yield the highest throughput: the improvement is large for both InvertedIndex and WordCount. Phoenix' default number of entries is 256. Increasing the number of entries improves Map performance because it decreases hash collisions. However, Phoenix's Merge phase sorts the output of the Reduces with a merge-sort that takes time logarithmic in the number of Map output hash entries, since the starting point for the merge-sort is the Reduce output for each Map output hash entry. As a result, increasing the number of entries helps Map but hurts Merge; Phoenix+Ext+Tuning chooses the best trade-off.

Metis' use of PSRS in Merge avoids any performance penalty due to the number of Map output hash entries, and as a result Metis can use enough hash entries to reduce the number of hash collisions to an insignificant level. The result is the increase in performance in Figure 11 between the Phoenix+Ext+Tuning line and the Metis line.

5.4 Workloads and data structures

This section demonstrates that Metis' hash+tree data structure for Map output is a good design choice, by comparing its performance to the data structures outlined in Section 3.2 (a per-Map worker append-sort only buffer which doest not group pairs after the sort, a per-Map worker append-group buffer which does grouping after the sort, a per-Map worker single b+tree, and a per-Map worker fixed-size hash table with 256 b+tree entries). These designs are compared using InvertedIndex with four input workloads (all tests run with 16 cores):

Workload	Input size	Keys	Pairs
Few Keys & Many Dups	100 MB	108 K	13.3 M
Many Keys & Few Dups	100 MB	9.5 M	13.7 M
Many Keys & Many Dups	800 MB	513 K	106.7 M
Unpredictable	500 MB	457 K	102.4 M

Figure 12 shows throughput with the "Few keys & Many dups" workload. Append-sort and append-group perform the most poorly because it cannot take advantage of the fact that there are few keys; for example, the time append-sort and append-group spend sorting is $O(p \log p)$ while b+tree spends only $O(p \log k)$ time inserting. Append-group performs slightly better than append-sort since grouping key/value pairs of the map output at Map phase has better locality than grouping at Reduce phase. Metis performs best because it has $O(1)$ lookup operations rather than $O(\log k)$.

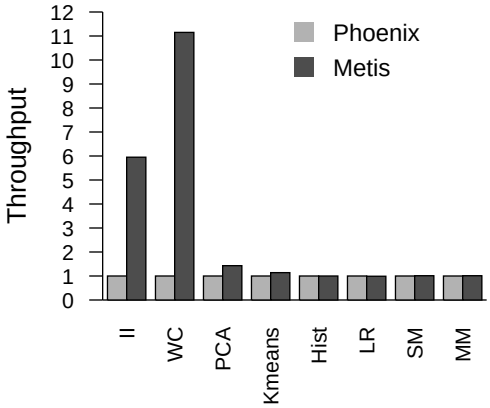

Figure 10: Comparison of Metis throughput to Phoenix throughput. The bar lengths are rate of work divided by Phoenix's rate of work, so that all the Phoenix bars have length one. Measured on 16 cores.

Figure 12: Throughput of InvertedIndex with different data structures using the "Few keys & Many dups" input.

Figure 13 shows throughput with the "Many keys & Few dups" workload. Append-sort, append-group and b+tree perform similarly: they all have $O(\log k)$ sorting times, and they all cause the Reduce output to be in nearly sorted order, which makes Merge fast. Metis runs somewhat slower than either, because its Reduce output is not sorted and requires significant work in the Merge phase.

Figure 14 shows throughput with the "Many keys & Many dups" workload. Append-sort and append-group perform the worst due to the same reason for "Few keys & Many dups" workload. The other three data structures do allow grouping, but are lookup-intensive due to the large number of repeated keys. Metis has the fastest lookup operation ($O(1)$), so unlike "Few keys & Many dups", it performs greatly better than b+tree and fixed sized hash table.

Figure 15 shows the throughput of InvertedIndex with

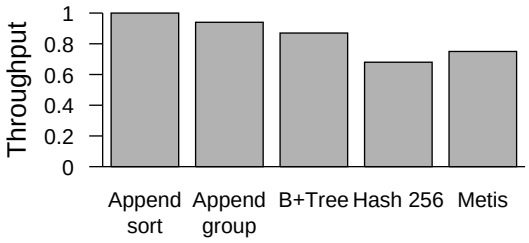

Figure 13: Throughput of InvertedIndex with different data structures using the "Many keys & Few dups" workload.

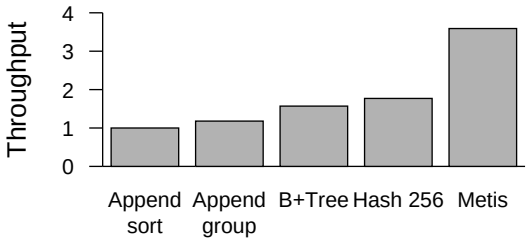

Figure 14: Throughput of InvertedIndex with different data structures using the "Many keys & Many dups" workload.

different data structures using the "Unpredictable" workload, whose input has 457,000 keys but which Metis incorrectly predicts to have 2.8 million keys. Metis is still attractive for this workload since the extra cost of allocating an overly large hash table is small.

5.5 Prediction

We evaluate the accuracy and overhead of sampling for InvertedIndex, WordCount, and Kmeans, for which the MapReduce library is critical to the performance. As Figure 16 shows, since the input of InvertedIndex and PCA has few repeated keys, each Map worker is expected to receive a subset of the keys, which is about $\frac{1}{16}$ of the total number of keys (593 K and 131 K). For WordCount and KMeans, the input contains more repeated keys than core number, so that each Map worker would process almost all the keys.

The "PK" column shows the predicted number of keys per Map worker. For WordCount, the prediction is notably larger than expectation because the keys are not uniformly distributed. The fact that the input file contains more repeated keys at the beginning of the file, which is used by Metis for sampling, results in over prediction.

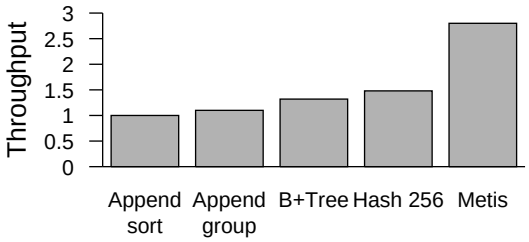

Figure 15: Throughput of InvertedIndex with different data structures using the "Unpredictable" workload. The input contains 457K keys, but Metis predicts that each Map worker would hold 2.8M keys.

	Total Keys	EK	PK	Overhead
II	9.49 M	593 K	697 K	0.8%
WC	1 M	1 M	2.67 M	6.3%
PCA	2.10 M	131 K	131 K	1.2%
KMeans	16	16	17	2.7%

Figure 16: Accuracy and overhead of prediction with 16 cores. "EK" stands for "Expected Keys per Map worker", while "PK" stands for "Predicted Keys per Map worker".

For other benchmarks processing uniform inputs, Metis' sampling scheme can adapt to the input, and the overhead of prediction is less than 3%.

5.6 Metis versus Hadoop

Hadoop [15] is an open-sourced MapReduce implementation for clusters. To facilitate debugging, Hadoop provides a standalone operation to run in a non-distributed mode. We compare the performance of WordCount on standalone Hadoop (version 0.20.1) and Metis.

For WordCount with a 10MB input containing 1M key and 1.3M pairs, single-core Metis runs in 4 seconds, and single-core Hadoop runs in 31 seconds. Hadoop cannot easily be configured to use parallel workers on a single multicore machine. This comparison is not entirely fair, however, since Hadoop uses Java and hasn't focussed on single-node performance. If Metis were used in a cluster environment, we expect Metis' good per-node performance would provide good cluster performance as well.

6 Discussion

This section summarizes the more interesting insights gained from investigating the performance of Metis and Phoenix.

6.1 Locality

The number of Map input splits in Metis is mostly driven by the goal of balancing load across the cores. We found out, as also pointed out in the Phoenix evaluation (see section 5.3 of [11]), that choosing a split size that allows the split to fit in a core's cache has little impact on performance (except for MatrixMultiply), because there is little locality; the working set of the applications is much bigger than the cores' caches.

We also observed that Kmeans has no short term temporal locality in access pattern. Kmeans with Phoenix runs faster with smaller unit size because the Map phase is extremely imbalanced with larger unit size. Since the input is small (1.2 MB), with a unit size of 128KB, there are only 9 Map splits for 8 cores. Metis splits the input by 256 by default to avoid this case.

Since the grouping-by-key operation is a global sort, we investigated in isolation different ways of performing sorts that might have better locality. For example, we explored using quicksort on an input of the size of an L2 cache and then use merge sort on the sorted L2 buffers. We observed some benefits in isolation but small compared to the overall run-time of an application. Nevertheless, we believe that optimizations of these kind are likely to be important when processors have more cores. With more cores, it might also become important to make a distinction between nearby caches and far away ones, and perhaps perform a hierarchical sort.

6.2 Processor

The results in this paper are not much different for the AMD and Intel processors. Performance on the 16-core AMD computer is slightly better than on the 16-core Intel computer. The primary reason is that the AMD Opteron processors are faster processors than the Intel ones (quad-core Xeon processors), and run at a higher clock rate (2.0 GHz versus 1.6 GHz). MatrixMultiply and InvertedIndex are the exceptions, with the MatrixMultiply difference being most pronounced (10.7s versus 7.9s on 16 cores). The Intel processors win on this application because they have bigger and faster caches, and MatrixMultiply has good memory locality.

6.3 Operating system

The impact of the operating system is minimal for MapReduce applications, since they interact with the operating system infrequently. There are three areas, however, where operating system functions have some impact: memory allocation, address space management, and file I/O.

Memory allocator In initial evaluations of WordCount on Windows we found that it performed surpris-

ingly badly, spending a large amount of time in the memory allocation library. The Windows library has a number of flags, and we tried several different configurations of memory management to find what might be the major cause of the problem: contention of the heap-lock, inappropriate algorithm in heap management, or something else. Of five different configurations we tried the configuration that sets the "Low Fragment Heap" (LFH) flag is the best. Using a heap per thread instead of shared global heap brings only a small performance improvement, indicating contention for the heap-lock is not intensive when LFH is enabled. The 32-bit version of WordCount performs slightly better than the 64-bit one, due to the increased memory usage in the 64-bit version. By contrast, shared-heap without LFH option scales rather badly in both 32-bit and 64-bit, due to the false sharing of caches among multiple worker threads. On Linux we used the Streamflow allocator instead of the standard libc allocator, because Streamflow is more efficient.

Address space management In Phoenix and in earlier versions of Metis, the Map, Reduce, and Merge phase each create a new thread per core and delete the threads at the end of the phase. Both on Windows and Linux, when freeing the stack of a thread, the operating system updates the process's address space. This update can be an expensive operation on Windows (because of a global lock) and to a lesser degree on Linux (because of soft page faults). To avoid any thread-related overhead, Metis creates threads once in the beginning and reuses them in each phase.

File I/O The Histogram and LinearRegression application, which have few keys, spend little time in the MapReduce library and most of their time in application functions. The main system overhead, which is small compared to the application run-time overhead, is locking overhead in the page fault handler in the operating system when reading the input file. With a larger number of cores, operating systems may need more tuning to ensure that they don't become a performance bottleneck for applications.

6.4 Fault tolerance

Unlike MapReduce libraries for data centers, Metis doesn't support restarting Map and Reduce computations when a Map or Reduce fails. As suggested by the Phoenix paper, such a feature could be important if cores could fail independently. Since the failure model for large-scale multicore processors hasn't been worked out yet, we have not implemented this feature and assume that the unit of failure is a complete computer. A fault-tolerant master process can reschedule the tasks assigned to the failed computer to another computer in the data center.

7 Related work

This paper shows that the global operation that groups key/value pairs by keys is a bottleneck on multicore processors, and proposes various optimizations. This global operation can also be a bottleneck in data centers and necessitate special care in that environment as well. On a multicore machine, this bottleneck appears as cache misses and memory accesses. In a data center, it appears as the exchange of messages and load on core switches [3]. Some systems have been designed to improve locality in the data center. For example, the Hadoop scheduler [15] is aware of the switch topology and uses that knowledge when scheduling to keep load local and off the core switches.

The Phoenix MapReduce library [11] is the library that is most related to Metis. Phoenix has been implemented on Solaris and evaluated on a 24-processor SMP and the UltraSparc T1 processor. The evaluation demonstrates that applications that fit the MapReduce model can perform competitively with hand-tuned parallel code using Pthreads. Metis augments the Phoenix work in several ways. First, it focuses on the challenge of handling the intermediate key/value pairs. It combines several different Map output data structures that have $O(n \log n)$ performance, or better for specific workloads; Phoenix's implementation has $O(n^2)$ performance in the worst case. Second, it shows that no single data structure is best, and that the Combiner function is also important on multicore processors, since it reduces the pressure of storing intermediate values. Third, the evaluation demonstrates the benefits of Metis on commodity multicore processors, and provides a more detailed analysis of which applications can benefit from MapReduce.

The core challenge the paper addresses is the design of an intermediate data structure that, in essence, supports an efficient global, parallel sort/group-by function for MapReduce applications. Like much of the previous work on sorting, Metis uses a data partitioning scheme to achieve good locality for individual Map and Reduce computations, and adopts key sampling from sophisticated sorting algorithms (e.g., [14]). The main contribution of Metis is what data structure to use when, by providing insight in the structure of the workloads induced by MapReduce applications. These insights result in substantial performance improvements for MapReduce applications and are likely to be important for any MapReduce implementation, as processors with large number of cores become prevalent.

Pavlo et al. compared Hadoop with databases implementations for the data center [9], and conclude that MapReduce implementers can learn much from database implementers. In this respect, Metis's contributions is similar: databases support efficient group-by operators

and that is important for certain MapReduce applications too.

A number of implementations extend or augment the Google's MapReduce model. Lämmel [7] describes an implementation of MapReduce in Haskell. Map-Reduce-Merge [2] adds an additional Merge step after Reduce to facilitate the joining of datasets output by different MapReduce jobs. Hadoop [15] is open-source and implemented in Java, and, as mentioned earlier, optimizes for the switch topology. Dryad [6] generalizes MapReduce into an acyclic dataflow graph. Mars [5] extends MapReduce to graphics processors.

MapReduce has been augmented with languages to make it even easier to program. For example, Google Sawzall [10] and Yahoo! Pig [8]. DryadLINQ [16] is a high-level language for programming Dryad in LINQ, a language for specifying queries.

Several papers have looked at scaling systems on multicore computers. Gough et al. [4] show how carefully organizing the fields of C structures can reduce both false sharing and cache line bouncing in the Linux kernel. Others have looked at special runtimes (e.g., McRT [12]) and operating systems (e.g., Corey [1]) to scale systems well with many core processors. Yet others have proposed special hardware-based features (e.g., Synchronization State Buffer [17]) to achieve good performance. Our work is in the same spirit, but tailored to MapReduce applications on commodity hardware and operating systems.

8 Summary

This paper studies the behavior of MapReduce on commodity multicore processors, and proposes the Metis library. The paper's main insight is that the organization of the intermediate values produced by Map invocations and consumed by Reduce invocations is central to achieving good performance on multicore processors. Metis stores these intermediate values using an efficient data structure consisting of a hash table per Map thread with a b+tree in each hash entry. As a result, Metis can achieve better performance than Phoenix on MapReduce applications that interact with the library frequently (e.g., applications with many keys).

We have found Metis useful in practice, using it on our 16-core computers for counting and sorting gigabytes of data generated as part of another research project. As computers with more cores become prevalent, we hope that others will find Metis useful too. We will make the code for Metis publicly available.

Acknowledgements This material is based upon work supported by the National Science Foundation under Grant No. 0834415.

References

[1] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, New York, NY, USA, Dec 2008. ACM.

[2] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified data processing on large clusters. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on management of data*, pages 1029 – 1040, New York, NY, USA, 2007. ACM.

[3] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, New York, NY, USA, Dec 2004. ACM.

[4] C. Gough, S. Siddha, and K. Chen. Kernel scalability — expanding the horizon beyond fine grain locks. In *Proceedings of the Ottawa Linux Symposium 2007*, pages 153–165, Ottawa, Ontario, June 2007.

[5] B. He, W. Fang, N. K. Govindaraju, Q. Luo, and T. Wang. Mars: a MapReduce framework on graphics processors. Technical Report HKUST-CS07-14, Department of Computer Science, HKUST, Nov 2007.

[6] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 59–72, New York, NY, USA, Apr 2007. ACM.

[7] R. Lämmel. Google's MapReduce programming model – revisited. Accepted for publication in the Science of Computer Programming Journal; Online since 2 January, 2006; 42 pages, 2006 – 2007.

[8] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *ACM SIGMOD 2008 International Conference on Management of Data*, June 2008.

[9] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. Dewitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD 09: Proceedings of the 2009 ACM SIGMOD International Conference*. ACM, June 2009.

[10] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: parallel analysis with Sawzall. *Scientific Programming Journal*, 13(4):277 – 298, 2005.

[11] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of HPCA*. IEEE Computer Society, 2007.

[12] B. Saha, A.-R. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. L. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, and J. Fang. Enabling scalability and performance in a large-scale CMP environment. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 73–86, New York, NY, USA, Apr 2007. ACM.

[13] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 2006 ACM SIGPLAN International Symposium on Memory Management*, 2006.

[14] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14:362 – 372, 1992.

[15] Yahoo! `http://developer.yahoo.com/hadoop/`.

[16] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, New York, NY, USA, Dec 2008. ACM.

[17] W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao. Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures. In *ISCA'07: Proceedings of the 34th annual international symposium on computer architecture*, pages 35–45, NY, NY, USA, 2007. ACM.