Language Agnostic MapReduce on Shared-Memory Systems

Erik Selin
School of Information Technology and Engineering
University of Ottawa
Ottawa, Canada K1N 6N5
erik.selin@gmail.com

December 10, 2017

Abstract

Data processing on shared-memory systems is becoming increasingly attractive due to the advent of high core-count CPUs. The MapReduce programing paradigm is a great fit for data processing on high core-count shared-memory systems however there is currently no industry ready runtime available. We present XRT, a novel MapReduce runtime which is designed from the ground up to be language-agnostic, fast and pedantic about resource usage. XRT Mapper and reducer programs can be written in any language since communication with the runtime is done over the standard streams. In addition, XRT is capable of achieving high performance by only using memory for intermediate data storage while maintaining the capacity to fall back to disk based data structures if the intermediate data is too large. In this paper we show that XRT is capable of achieving excellent speedup as worker counts are increased while simultaneously offering a very pleasant language-agnostic programming experience. We believe that XRT is quickly approaching a industry ready state and enables high-performance MapReduce on shared-memory systems.

1 Introduction

MapReduce is a restrictive programming model used to write efficient and highly parallel data processing programs without having to deal with the complexities of parallel programming. In general, clusters of networked servers is used to run MapReduce since individual servers have historically had fairly low core-counts. However, in recent years server core-counts have increased and today 128-core shared-memory system are generally available.

We believe that shared-memory systems with high core-counts provides a excellent environment for modern data processing and that adoption is currently hindered by the lack of a simple to use high performing runtime. Complexities associated with cluster computing can be avoided in shared-memory systems resulting in simpler operations while per-core performance is better since datasets that fit in memory can be processed without disk or network access. In addition, we believe that there has been a lack of interest in developing truly language agnostic MapReduce runtimes and that current shared-memory MapReduce runtimes are not industry ready.

To address these perceived lacks and shortcomings we have developed XRT, a language agnostic, industry ready, MapReduce runtime built from the ground up for high-performance on shared-memory systems. In this paper we conduct a literature review in Section 2, introduce XRT 0.1.0 in Section 3, explore the implementation of XRT in Section 4 and finally evalutes the performance of XRT in Section 5 before concluding the paper in Section 6.

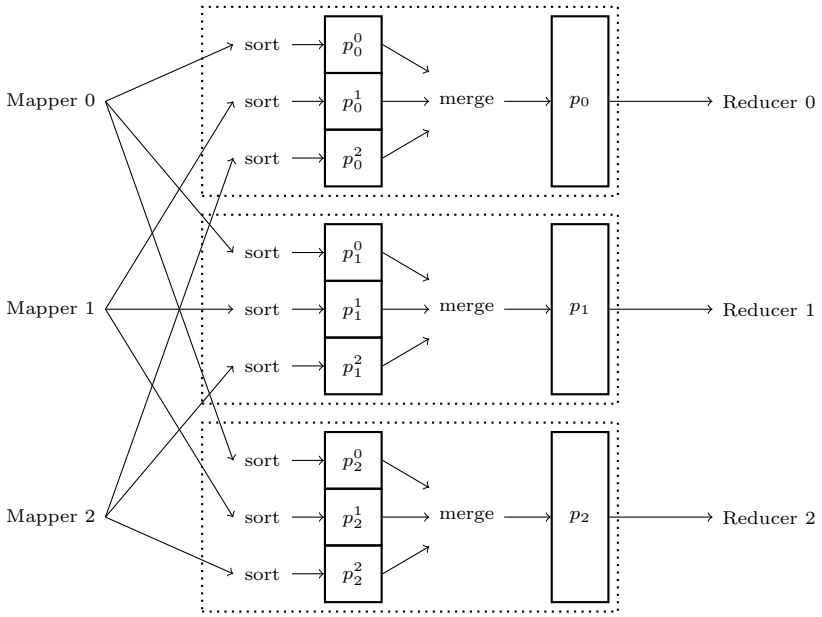

Figure 1: Classical MapReduce programming model.

2 Literature Review

The MapReduce programming model was initially introduced by Google in 2004 [9] and later popularized by Yahoo! through the Apache open-source project Hadoop MapReduce [2]. Hadoop MapReduce provides a MapReduce runtime for networked commodity hardware and its popularity has resulted in the development of runtimes for alternative environments like GPUs [12], FGPA's [20], Coprocessors [16] and shared-memory systems [18] [19] [7] [17] [8].

2.1 Programming Model

In the classical MapReduce programming model [9], illustrated in Figure 1, programmers only need to supply mapper and reducer programs together with the source of the input and the destination of the output, everything else is handled by the runtime. For most MapReduce runtimes this means executing parallel workers running multiple instances of the mapper and reducer programs, reading and distribution of the input, collection and writing of the output and shuffling and sorting intermediate data. A classical MapReduce job starts by splitting the input and running a mapper for each split. Each mapper will consume the records in the split and output key-value pairs which are collected by the runtime and partitioned into sorted buckets. Key-value pairs with the same key end up in the same bucket no matter which mapper produced the pair. Once all mappers are done a reducer is started for each bucket and all key-value pairs in the bucket are processed in order. This means that all key-value pairs with the same key will all be processed by the same reducer. Finally, the output of the reducers becomes the output of the job.

There is a lot of room for implementation specific deviations and optimizations in this description but on a high level all MapReduce runtimes operates by this model [9] [2] [18] [19] [7] [17] [8]. The success of the MapReduce model comes from the ease of writing sequential mapper and reducer programs that are able to accomplish advanced data transformation tasks in parallel when executed through a MapReduce runtime [9].

2.2 Language support

MapReduce runtimes with multi-language or language-agnostic support is not something that has been greatly explored by the MapReduce community. In general MapReduce runtimes are single language: Googles proprietary MapReduce is described as a C++ runtime [9], Hadoop MapReduce provides a runtime for Java [2] and shared-memory runtimes like Phoenix [18], Phoenix++ [19], Metis [17], Ostrich [8] and CilkMR [7] provide runtimes for C or C++.

In fact, the only available language-agnostic MapReduce runtime seems to be Hadoop Streaming. Hadoop Streaming is built upon Hadoop MapReduce and achieves language-agnostic support by replacing the mappers and reducers with externally run processes that communicates with the MapReduce runtime over the standard streams [4]. Unfortunately the MapReduce runtime that power Hadoop Streaming is still the Java optimized Hadoop MapReduce runtime and the resulting performance of Hadoop Streaming is very bad [10] [15] [6]. ShmStreaming is an attempt to increase the performance of Hadoop Streaming by communicating over shared memory instead of the standard streams [14]. The performance of Hadoop Streaming + ShmStreaming is superior to Hadoop Streaming but communication over shared memory is not as straightforward to implement compared to communication over standard streams. In addition, ShmStreaming still suffers from the fact that Hadoop MapReduce is not optimized for interacting with mappers and reducers that are run as external processes.

The Hadoop community has also developed Hadoop Pipes which outperforms Hadoop Streaming but only offers support for C++ [3]. Language support for the python programming language without using Hadoop Streaming is available through the Pydoop project. Pydoop integrates with Hadoop MapReduce by wrapping Hadoop Pipes and offers superior performance versus running python mapper and reducers through Hadoop Streaming [15]. An alternative approach to adding specific language support to Hadoop MapReduce was undertaken in the Perldoop project [6]. Perldoop provides a perl-to-java transpiler specifically built to convert Perl Hadoop MapReduce jobs to Java Hadoop MapReduce jobs. Perldoop can achieve very good performance since it is effectively running regular Java Hadoop MapReduce however there are a lot of limitations since only a subset of very specifically formatted Perl is supported.

Finally, there seems to be no MapReduce runtime that is built explicitly for multi-language or truly language-agnostic MapReduce.

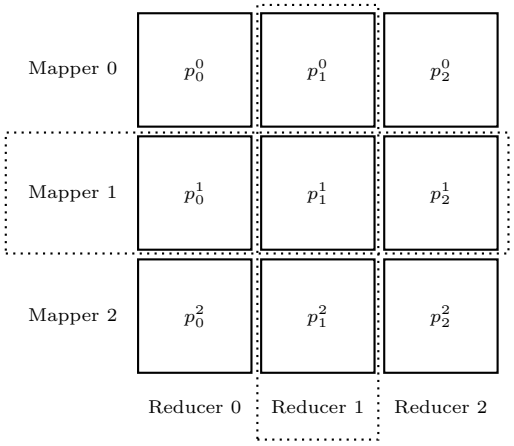

Figure 2: Memory layout in shared-memory MapReduce runtimes.

2.3 Shared-memory

Shared-memory systems are becoming increasingly capable and cost effective for data processing. At the time of writing it is possible to get a on demand system with 128 cores and 3.9TB of memory [1]. Based on recent releases from popular cloud providers the trend of access to high core-count systems is likely to continue and cost per core will likely keep decreasing [1] [5].

MapReduce on shared-memory systems was first explored by the Phoenix [18] project. Besides being first to explore MapReduce on shared-memory systems the Phoenix project also contributed the matrix memory-layout illustrated in Figure 2. This memory layout is used in some form by all shared-memory

runtimes [18] [19] [7] [17] [8] and enables workers to communicate extremely efficiently with the runtime while avoiding contention. The columns in the matrix represent buckets and each entry in the matrix is a buffer responsible for storing the intermediate data produced by a mapper. During the mapper stage each mapper is given access to a row in the matrix and during the reducer stage each reducer is given access to a column in the matrix. This methodology ensures that no communication is required across workers during execution which in turns maximizes throughput.

The Phoenix runtime has gone through multiple iterations and Phoenix 2.0 was the result of performance issues when Phoenix was run on larger shared-memory systems processing larger data sets [21]. In particular the intermediate data buffers that are used to store keys from the key-value pairs produced by the mappers are implemented as sorted arrays in Phoenix. This lead to performance issues since the arrays needed to reallocate whenever they ran out of space and whenever a new key was added all keys coming after the new key had to be shifted. Phoenix 2.0 solved this issue by significantly increasing the number of sorted key arrays so that on average only one key resides in each array [21].

Further work on the Phoenix project has lead to the determination that the intermediate sorted arrays were limiting the potential performance of the runtime. This resulted in the reimplementation of Phoenix in C++ and the start of the Phoenix++ project [19]. Phoenix++ is the poster child of shared-memory MapReduce runtimes optimized for speed and extends the MapReduce API to include containers and combiner objects. Containers exposes the internals of the shuffle stage to the programmer and makes it possible to select or tune the data structure used for shuffling [19]. Combiner objects exposes the internals of the intermediate data buffers and makes it possible to select the data structure used for intermediate data buffering [19]. While the introduction of containers and combiner objects in Phoenix++ does allow for better performance it has been argued that they break the MapReduce abstraction and programmers now need to be intimately familiar with the Phoenix++ internals to pick the correct containers and combiner objects [7].

An alternative approach to dealing with the perceived shortcomings of the Phoenix project is the Metis project. Instead of increasing the configurability of the runtime Metis introduces a compromise intermediate data buffer [17]. Using a more advanced buffer Metis is able to achieve significant performance increase over Phoenix on data processing problems involving very little mapper/reducer computation on datasets with few keys but many duplicates. Metis does not offer any performance increase over Phoenix on data processing jobs involving a significant amount of mapper/reducer computation or on datasets with few duplicate keys [17].

An extension of the Phoenix model is Tiled-MapReduce and its prototype implementation Ostrich. Ostrich extends the model introduced by Phoenix by using the tiling strategy commonly used in the compiler community. Ostrich and Tiled-MapReduce in general splits the input into subsets and runs smaller jobs on each subset. Each smaller job runs mappers that produces a intermediate data buffer like Phoenix but then reduces the partial intermediate data buffer using a combiner into a secondary intermediate data buffer. Once all smaller jobs have run the secondary intermediate data buffers are reduced by the reducers. Since the combiner reduces the amount of intermediate data this approach increases the performance of the runtime and allows for processing of more data. In addition the primary intermediate data buffer that is filled by the mappers can be reused between the smaller jobs. This means that Ostrich is able to avoid a lot of memory allocation calls compared to Phoenix and other shared-memory runtimes [8].

The CilkMR project is one of the newest MapReduce runtimes for shared-memory systems and addresses the key-value pair serialization/deserialization overhead of all the previously discussed runtimes [7]. Instead of operating on key-value pairs CilkMR operates over typed data containers which means that the mappers can pass arbitrary data structures to the reducers. CilkMR is also inspired by Phoenix++ in that it allows the programmers to control intermediate data structures and tune the runtime. The performance of CilkMR is really good on computationally

```
Usage: xrt [--help] [--version] <options>

Resource options:
  --workers <n>     set the number of workers to <n> (default: 4)
  --memory <m>      set the amount of allocated memory (default: 256m)
  --tempdir <dir>   set the temporary directory (default: system temporary directory)

Job options:
  --input <file|dir>  input file or directory
  --mapper <cmd>      mapper command (required)
  --reducer <cmd>     reducer command
  --output <dir>      output directory
```

Figure 3: Usage description of the XRT cli tool

heavy tasks since mappers and reducers can operate on complex data structures. However, Phoenix++ performs better than CilkMR on classical data processing jobs like word-count and string-search [7] [9].

A completely different idea for bringing MapReduce to shared-memory systems is the Hone project. Hone attempts to take advantage of the familiarity of Hadoop MapReduce by providing a Hadoop MapReduce compatible Java API [13]. The goal is to create a runtime that allow you to run the exact same Java code that was written for Hadoop MapReduce. In benchmarks Hone beats Phoenix however Phoenix++ beats Phoenix by a much wider margin and thus it seems like Phoenix++ is likely much faster then Hone.

A last note about recent development in shared-memory MapReduce runtimes is the usage of disk based data structures. In Hadoop MapReduce almost all intermediate data will reside on disk at some point but in all the above mentioned shared-memory runtimes there is never a mention of disk backing. There has been some exploration in extending Metis to include the capacity of spilling intermediate data to disk if the input is too large [11]. However, this optimization was never contributed back to the Metis project.

3 XRT 0.1.0

XRT 0.1.0 is the first release of the XRT MapReduce runtime for shared-memory systems. The objective of XRT is to offer a simple, high-performance, resource-aware and language-agnostic MapReduce runtime. Internally XRT is based on ideas developed in shared-memory MapReduce systems like Phoenix [18] combined with language-agnostic concepts from Hadoop Streaming [4]. In addition, it explores novel topics in the shared-memory MapReduce space like resource awareness and the ability to use secondary storage for datasets that do not fit in memory. XRT is being implemented in the Go language and Figure 3 shows the usage options of the XRT cli tool.

To keep usage simple and enable truly language-agnostic support XRT is built to operate on newline-delimited data. Widely used formats like csv, tsv and newline delimited json can all be used. On a high level XRT follows the classical MapReduce model illustrated in Figure 1 and programming MapReduce jobs involves writting mapper and reducer programs.

XRT mapper programs read newline-delimited records from stdin and writes newline-delimited worker-id/record pairs to stdout. The XRT runtime shuffles the records by the associated worker-id and then sorts the records before passing them to the reducer program.

XRT reducer programs read sorted newline-

delimited records from stdin and writes newline-delimited records to stdout which in turn becomes the output of the MapReduce job.

3.1 Examples

Here are a few common MapReduce tasks to illustrate usage of the XRT MapReduce runtime.

string-search: The mapper program reads records from stdin and only writes records to stdout that matches the search pattern. No reducer program is needed.

word-count: The mapper program reads words from stdin, assigns a worker-id to each word to ensure that all words that are the same end up on the same reducer. The reducer program reads a word from stdin and keeps reading as long as new words is the same as the first word while keeping a count of how many words have been read. When a new word is read from stdin that does not match the first word the reducer writes the current word and the associated count to stdout and then restarts the process with the new word.

sort: The mapper program reads records from stdin and assigns a worker-id to the record such that records ending up on reducer_i are all smaller then records ending up on reducer_{i+1}. The reducer program reads records from stdin in sorted order and simply pass them to stdout as all sorting and partitioning needed has already been completed by the runtime.

4 Implementation

The flow of records through a XRT MapReduce job is split into 9 distinct subroutines out of which 4 are visible to the user (input, mapper, reducer, output) and 5 are internal to the runtime (shuffle, buffers, spilling, merging). In this section the technical details of each subroutine is described in order of execution for a typical XRT job.

input: Input can be a single or multiple files containing newline-delimited records. The XRT runtime enumerates the input files and assigns 64mb chunks from the input to the mapper programs while ensuring that each chunk starts and end at newline boundaries.

mapper: Mapper programs read newline-delimited records from stdin, processes the record, assigns a worker-id to each record and writes the worker-id/processed-record pair to stdout. The worker-id will be used by the runtime to shuffle the record to the appropriate reducer program.

shuffle: Shuffle receives the worker-id/processed-record pairs from the mapper programs and shuffles them to the appropriate buffer based on worker-id.

buffers: Buffers are implemented using the matrix memory model illustrated in Figure 2 which means that for each mapper program there is a associated list of buffers, one for each reducer program. Records are directed to a buffer by the shuffle which can be done lock-free due to the matrix based memory layout. The current implementation of XRT uses a simple byte buffer where pointers to records are prepended and the records themselves are appended.

spilling: Spilling occurs when a buffer becomes full, meaning the free space between the prepended record pointers and the appended records is too small to hold additional pointers and records. During spilling the full buffer is sorted, written out to disk and then cleared.

sort: Sort starts once all mapper programs have terminated. It first sorts the in memory records by only moving the record pointers, reducing the number of bytes that need to be moved around while also making it possible to sort in place. If any buffers spilled then sort uses a external heap-based n-way merge to create a single sorted file for each buffer.

merging: Merging grabs all in-memory buffers and all on-disk spill files associated with a reducer and using a heap-based n-way merge creates a single sorted stream of records.

reducer: Reducer programs read the sorted stream of records from the merging phase which are delivered to the reducer program over stdin. The reducer program processes the records and writes them to stdout.

output: The records written to stdout by the reducer programs are written to a temporary location on disk. Once all reducer programs have finished successfully XRT commits the data by moving the temporary data to the target output directory.

5 Performance

In this section we present the results of running common MapReduce tasks using XRT 0.1.0 on a large shared-memory system.

5.1 System setup

The shared-memory system had 32 cores, 224GB of memory, one 500GB SSD disk and was running a default Ubuntu 16.04 installation. XRT was compiled using Go 1.9.1 and the mapper and reducer programs were implemented in Python and executed by Python 2.7.11. Execution times were collected using the linux time command.

5.2 Dataset

A artificial 50GB dataset was generated and contained 500 million 100-byte records. The dataset was split into 50, 1GB files each containing 10 million 100-byte records. The records were split into a 5-byte key followed by a tab-character followed by a 93-byte payload followed by a newline-character. Every byte in every record was randomly picked from the sequence of lowercase characters in the range a-z.

5.3 Benchmarks

We ran the 3 examples from Section 3.1 as our benchmarks with the following implementation details.

string-search: The mapper program reads records from stdin and only writes records to stdout if the first 5-character of the 93-byte payload are all the character 'a'.

word-count: The mapper program reads records from stdin and sets the worker-id for each record using a hash of the 5-byte key modulo the number of workers. The mapper program only emits the worker-id/key pair as we are only interested in a count by key. The reducer program counts the number of times a key is observed and emits the current key and current count to stdout when a new key is read from stdin.

sort: The mapper program reads records from stdin and assigns a worker-id to each record ensuring that records assigned to worker_i are guaranteed to be smaller then records assigned to worker_{i+1}. The reducer program reads records from stdin and simply write them to stdout directly as they are already in sorted order.

5.4 Results

5.4.1 Baseline

For the first performance evaluation each of the benchmarks were implemented as optimal and sequential non-MapReduce programs. These baseline implementations where then compared with the performance of the XRT MapReduce implementations. When number of XRT workers was set to 1 the baseline implementations where on average 20% faster due to the overhead incurred by the XRT runtime. When the number of workers was increased to 2 XRT was able to completely outperform the optimal sequential implementations.

5.4.2 Speedup

Figure 4 shows the speedup of string-search, word-count and sort as the number of XRT workers is increased. The results show that we are approximately doubling the speedup as we double the core count which is precisely the kind of speedup that we're aiming to achieve. In some cases we are even able to more then double the speedup as we are effectively moving computation out of Python (a dynamic, slower

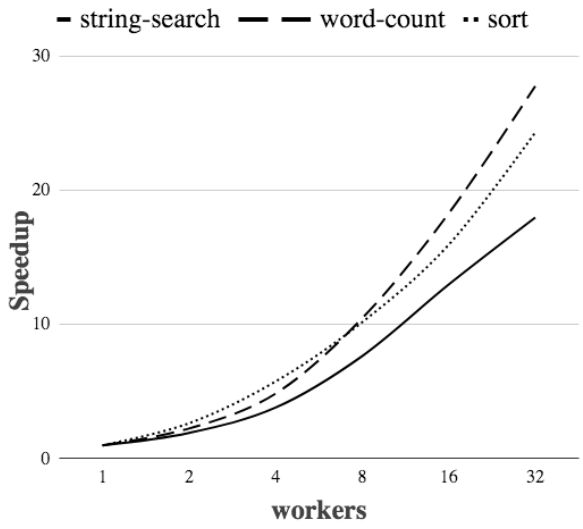

Figure 4: Speedup of XRT as worker counts are increased

language) to Go (a compiled, high performing language). The job with the lowest speedup is string-search which also happens to be the least computationally heavy job and thus is likely bottlenecked by the fact that our benchmarking system only has a single SSD.

We also see that as we approach 32 workers we have a bit of a reduction in speedup across all benchmarks. Again this is likely due to the single SSD becoming a bottleneck.

6 Conclusion

This paper introduced XRT 0.1.0, a novel language-agnostic MapReduce runtime for shared-memory systems. The implementation details of XRT were explored and through our benchmarks we have shown that XRT enables high-performance data processing with a good speedup profile as core-counts increase.

Furthermore, we have shown that XRT addresses the perceived shortcomings and lacks in the shared-memory MapReduce community. XRT is language-agnostic and memory first but can process datasets that do not fit in memory by having first class support for spilling to disk. In addition, we believe XRT to be heading towards a industry-ready state with easy development, deployment and management as well as a robust implementation.

We intend to continue optimizing the various internal components of XRT and run further benchmarks on even larger shared-memory systems. In particular, we are interested in running benchmarks on shared-memory systems with multiple RAID mounted SSD's to address the slight slowdowns we observed in our performance evaluation.

Overall, we believe that we have shown that language-agnostic shared-memory MapReduce is possible and extremely attractive following the advent of high core-count shared-memory systems.

References

[1] The aws x1 instance type. `https://aws.amazon.com/ec2/instance-types/x1/`.

[2] Hadoop. `http://hadoop.apache.org/`.

[3] Hadoop pipes. `https://hadoop.apache.org/docs/r1.2.1/api/org/apache/hadoop/mapred/pipes/package-summary.html`.

[4] Hadoop streaming. `http://hadoop.apache.org/docs/current/hadoop-streaming/HadoopStreaming.html`.

[5] List of google cloud instance types. `https://cloud.google.com/compute/docs/machine-types`.

[6] José M Abuín, Juan C Pichel, Tomás F Pena, Pablo Gamallo, and Marcos García. Perldoop: Efficient execution of perl scripts on hadoop clusters. *2014 IEEE International Conference on Big Data (Big Data)*, pages 766–771, 2014.

[7] Mahwish Arif, Hans Vandierendonck, Dimitrios S. Nikolopoulos, and Bronis R. de Supinski. A scalable and composable map-reduce system. *IEEE International Conference on Big Data (Big Data)*, pages 2233–2242, 2016.

[8] Rong Chen, Haibo Chen, and Binyu Zang. Tiled-mapreduce: Optimizing resource usages of data-parallel applications on multicore with tiling. *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 523–534, 2010.

[9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 2004.

[10] Mengwei Ding, Long Zheng, Yanchao Lu, Li Li, Song Guo, and Minyi Guo. More convenient more overhead: the performance evaluation of hadoop streaming. *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, pages 307–313, 2011.

[11] Tharso Ferreira, Antonio Espinosa, Juan Carlos Moure, and Porfidio Hernández. An optimization for mapreduce frameworks in multicore architectures. *Procedia Computer Science*, 18:2587–2590, 2013.

[12] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. pages 260–269, 2008.

[13] K Ashwin Kumar, Jonathan Gluck, Amol Deshpande, and Jimmy Lin. Hone: Scaling down hadoop on shared-memory systems. *Proceedings of the VLDB Endowment*, 6(12):1354–1357, 2013.

[14] Longbin Lai, Jingyu Zhou, Long Zheng, Huakang Li, Yanchao Lu, Feilong Tang, and Minyi Guo. Shmstreaming: A shared memory approach for improving hadoop streaming performance. *2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, pages 137–144, 2013.

[15] Simone Leo and Gianluigi Zanetti. Pydoop: a python mapreduce and hdfs api for hadoop. *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 819–825, 2010.

[16] Mian Lu, Lei Zhang, Huynh Phung Huynh, Zhongliang Ong, Yun Liang, Bingsheng He, Rick Siow Mong Goh, and Richard Huynh. Optimizing the mapreduce framework on intel xeon phi coprocessor. *2013 IEEE International Conference on Big Data*, pages 125–130, 2013.

[17] Yandong Mao, Robert Morris, and M Frans Kaashoek. Optimizing mapreduce for multicore architectures. *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep*, 2010.

[18] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. *IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, 2007.

[19] Justin Talbot, Richard M Yoo, and Christos Kozyrakis. Phoenix++: modular mapreduce for shared-memory systems. *Proceedings of the second international workshop on MapReduce and its applications*, pages 9–16, 2011.

[20] Zeke Wang, Shuhao Zhang, Bingsheng He, and Wei Zhang. Melia: A mapreduce framework on opencl-based fpgas. *IEEE Transactions on Parallel and Distributed Systems*, 27(12):3547–3560, 2016.

[21] Richard M Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. *IEEE International Symposium on Workload Characterization*, pages 198–207, 2009.