LITERATURE REVIEW: XRT 0.1.0 - A Language-Agnostic MapReduce Runtime for Shared-Memory Systems

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

October 8, 2017

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. The 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 multiple runtimes for alternative environments like GPUs [12], FGPA's [20], Coprocessors [16] and shared-memory systems [18] [19] [7] [17] [8].

The increasing availability of cost-effective shared-memory systems with high core counts [1] [5] means that shared-memory systems are becoming a serious alternative to networked commodity hardware for parallel data processing. From a previous survey of the literature it was observed that MapReduce runtimes for shared-memory systems are all built with the sole purpose of beating benchmarks. They tend to require the programmer to write mapper and reducer code in C or C++, are not cross platform compatible, occasionally break the MapReduce abstraction and in some cases are not even available for use.

The purpose of this literature review is to confirm these observations as well as dig deeper into the literature to paint a clear picture of the current state of MapReduce runtimes for shared-memory systems. Ultimately, the goal is to justify the creation of XRT, a new cross-platform language-agnostic MapReduce runtime for shared-memory systems that is inspired by Hadoop Streaming [4] and based on techniques developed in modern MapReduce runtimes for shared-memory systems [18] [19] [7] [8] [17].

2 Literature Review

2.1 The MapReduce 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

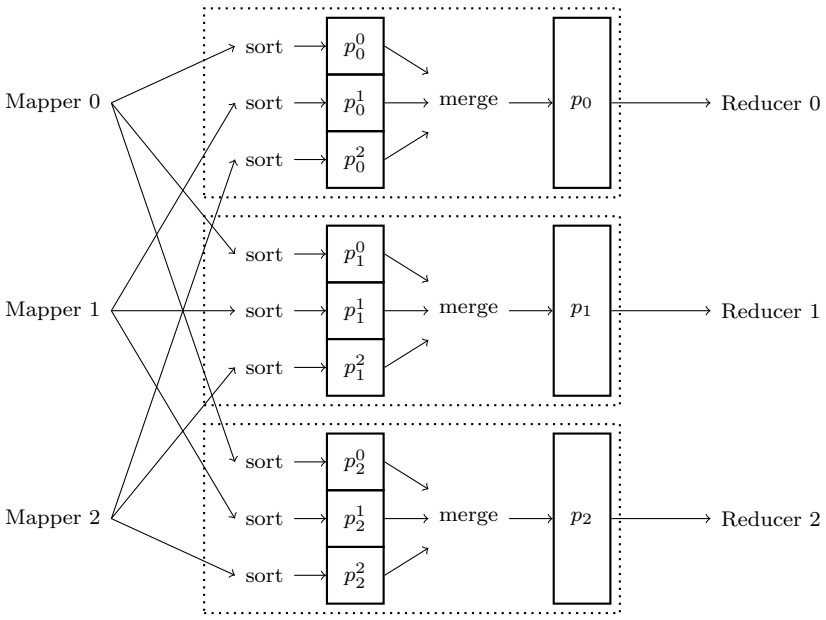

Figure 1: Classical MapReduce programming model.

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 produced by the mappers 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 MapReduce 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.

2.3 MapReduce on shared-memory systems

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 3.9 terabytes of memory for $26/hour [1]. Based on recent releases from popular cloud providers the trend of affordable high core-count systems is likely to continue and cost per core will likely keep decreasing [1] [5].

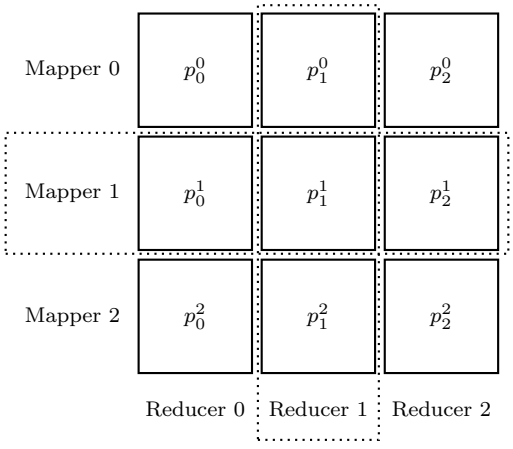

Figure 2: Memory layout in shared-memory MapReduce runtimes.

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 partitions 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 runs 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 heavy tasks since mappers and reducers can operate on complex data structures. However, Phoenix++ performs better than CilkMR on classical data processing jobs like wc and strmatch [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 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 Conclusion

The field of shared-memory MapReduce runtimes is still extremely young and almost all efforts are focused on beating benchmarks. Programming mapper and reducer code for shared-memory MapReduce runtimes requires knowledge of C or C++ and sometimes a deep understanding of the runtime internals. A surprising discovery from this review is that the shared-memory MapReduce community is not really considering handling data that is too large to fit in memory.

Following the literature review the case for building XRT has been strengthen. Shared-memory MapReduce runtimes lack a cross platform, disk aware and simple to use runtime and the broader MapReduce community lack exploration of language agnostic runtimes. That being said a lot of great ideas have already been developed by the shared-memory MapReduce runtimes. Concepts like the matrix memory layout, intermediate data structure reuse and language-agnostic support through communication over standard streams will all be brough into XRT.

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 multi-core 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.