Optimizing the MapReduce Framework on Intel Xeon Phi Coprocessor

[1]Mian Lu [2]Lei Zhang [3]Huynh Phung Huynh [4]Zhongliang Ong [5]Yun Liang [6]Bingsheng He [7]Rick Siow Mong Goh [8]Richard Huynh
[1,3,4,7]Institute of High Performance Computing, A*STAR [2,5]Peking University [6,8]Nanyang Technological University
{[1]lum,[3]huynhph,[4]ongzl,[7]gohsm}@ihpc.a-star.edu.sg {[2]1000012927,[5]ericlyun}@pku.edu.cn [6]bshe@ntu.edu.sg

***Abstract*—MapReduce has become one of the most popular framework for building big-data applications. It was originally designed for distributed-computing, and has been extended to various hardware architectures, e.g., multi-core CPUs, GPUs and FPGAs. In this work, we develop the first MapReduce framework on the recently released Intel Xeon Phi coprocessor. We utilize advanced features of the Xeon Phi to achieve high performance. In order to take advantage of the SIMD vector processing units, we propose a vectorization friendly technique to assist the auto-vectorization as well as develop SIMD hash computation algorithms. Furthermore, we utilize MIMD hyper-threading to pipeline the map and reduce phases to improve the resource utilization. We also eliminate multiple local arrays but use low cost atomic operations on the global array for some applications, which can improve the thread scalability and data locality. We conduct comprehensive experiments to compare our optimized MapReduce framework with a state-of-the-art multi-core based MapReduce framework (Phoenix++). By evaluating six real-world applications, the experimental results show that our optimized framework is 1.2X to 38X faster than Phoenix++ for various applications on the Xeon Phi.**

I. Introduction

Big data analytics has been identified as an exciting area for both academic and industry. We have witnessed the success of applying various high performance computing techniques using co-processors, such as graphics processors (GPUs) [1], [2], [3], FPGAs [4] and the coupled CPU-GPU architecture [5], to solve big data analytical problems. In order to fully utilize the capability of those architectures, developers need to write co-processor specific programming languages, such as CUDA [6]. This may affect developer productivity, maintenance costs as well as the code portability. Therefore it is desirable to have high-performance accelerator systems with compatible software development and maintenance techniques that are compatible with existing CPU-baed systems.

Recently, Intel released a new product family named Xeon Phi for high performance coprocessors. It offers a much larger number of cores than conventional CPUs. Furthermore, it highlights the 512-bit width vector processing units (VPUs) and fully coherent L2 caches. The Xeon Phi has already demonstrated its promising through adoptions in either specific applications [7], [8] or supercomputer systems [9], [10]. Instead of optimizing individual applications like previous studies [7], [8], we investigate a productivity programming framework to facilitate the implementation of big data analytics tasks correctly, efficiently, and easily on Xeon Phi.

MapReduce [11] has become a popular programming framework for big data analytics. It was originally proposed by Google for simplified parallel programming on a large number of machines. Recently, it has been extended to different architectures to facilitate parallel programming, such as multi-core CPUs [12], [13], [14], [15], GPUs [16], [17], [18], [19], the coupled CPU-GPU architecture [20], FPGA [21] and Cell processors [22]. In this work, we investigate the implementation and optimization of the MapReduce framework on Xeon Phi. To the best of our knowledge, this is the first MapReduce framework specifically optimized for Xeon Phi.

Directly porting a multi-core CPU based MapReduce framework (such as Phoenix++ [13]) to the Xeon Phi cannot fully utilize the hardware capability. Because it is not aware of the advanced features of Xeon Phi, such as the VPUs, small L2 cache per core and small memory capacity. To address these issues, we develop **MRPhi**, the first MapReduce framework on Xeon Phi with following features.

- We implement the map phase in a vectorization friendly way to take advantage of the SIMD VPUs.
- We develop SIMD hash algorithms to utilize the SIMD VPUs.
- We pipeline the map and reduce phases to improve the resource utilization.
- We eliminate local containers by using atomic operations on the global container in certain cases. This mainly addresses the thread scalability issue (due to the limited memory size) for some applications.
- Based on the above four optimization techniques, we develop a framework that either automatically applies optimizing techniques or provides useful suggestions to the users.

The rest of the paper is organized as follows. We introduce the background in Section II. Section III gives detailed implementations. The experimental results are presented in Section IV. We conclude this paper in Section V.

II. Background

A. MapReduce Framework

The MapReduce framework is originally designed for distributed computing [11]. Later, it is extended to other architectures such as multi-core CPUs [12], [13], [14], [15], GPUs [16], [17], [18], [19], the coupled CPU-GPU architecture [20], FPGA [21] and Cell processors [22]. These different

MapReduce frameworks share the common basic workflow, but differ in detailed implementation and optimization.

We introduce the basic workflow of a MapReduce framework. At the beginning, a *split* function divides the input data across *workers*. On multi-core CPUs, a worker is handled by one thread. A worker usually needs to process multiple input elements. Thus the *map* function is applied to the input elements one by one within a worker. Such a call of the map function for an input element is called a *map operation*. Each map operation produces a set of intermediate key-value pairs. Then a *partition* function is applied to these key-value pairs according to the keys. After that, in the reduce phase, each *reduce operation* applies the reduce function to a set of intermediate pairs with the same key. Finally the results from multiple reduce workers are merged and output.

B. Intel Xeon Phi Coprocessor

The Intel Xeon Phi coprocessor was released in November 2012. It is based on Intel Many Integrated Core (MIC) Architecture. The current released product is 5110P. Overall, the Xeon Phi 5110P integrates 60 x86 cores on the same chip. Each core has a frequency of 1.05 GHz and supports 4 hardware threads. The memory hierarchy of Xeon Phi is similar to a conventional multi-core CPU. The memory refers to the main memory on the Xeon Phi, which is shared and accessible for all cores. The main memory size is 8 GB. Each core has local L1 and L2 caches. The L2 cache size is 512 KB per core. Additionally, on each core, there are 32 512-bit vector registers.

Xeon Phi has been used to accelerate linear algebra [8] and molecular dynamics [7]. Furthermore, Xeon Phi coprocessors also start to play an important role for supercomputers, such as STAMPEDE [9] and Tianhe-2 [10]. In the following, we briefly introduce the major features of the Xeon Phi.

512-bit vector processing units (VPUs). Xeon Phi features 512-bit VPUs on each core. Utilizing VPUs effectively is the key to deliver high performance. The VPUs can be either exploited by manual implementations with SIMD instructions or *auto-vectorization* by the Intel compiler.

MIMD Massive thread parallelism. Each core of the Xeon Phi supports up to 4 hardware hyper-threads. Thus, there are 240 threads in total. The MIMD (Multiple Instruction, Multiple Data) thread execution allows different threads to execute different instructions at any time. Thus, we can assign different workloads to different threads to improve the hardware resource utilization.

Coherent L2 caches with ring interconnection. The interconnection on the Xeon Phi employs a ring architecture. All L2 caches are coherent through the ring interconnection. This design is able to improve the cache efficiency by trying to avoid expensive memory accesses when cache misses occur.

Atomic operations. Atomic data types are well supported on the Xeon Phi. For random memory accesses, we find that the overhead of atomic operations is almost hidden by the access latency. Therefore, it is reasonable to exploit the use of atomic operations.

However, the memory size of the Xeon Phi, which is only 8 GB, is fixed and small compared with the traditional main memory. This may become a bottleneck when designing efficient algorithms. We demonstrate such an issue for particular applications and use atomic operations to address this issue in Section III-E.

C. Challenges of a Shared Memory MapReduce Framework on Xeon Phi

State-of-the-art shared memory MapReduce frameworks on multi-core architectures, such as Phoenix++ [13], can be executed directly on the Xeon Phi without code modification. However, we have identified three major performance issues of Phoenix++ when porting it onto the Xeon Phi. First, Phoenix++ takes little advantage of the VPUs on the Xeon Phi. The compiler is unable to vectorize the code effectively. Second, there are a large number of random memory accesses when building containers. Third, due to the limited memory (8 GB) on the Xeon Phi, Phoenix++ cannot scale to hundreds of threads when the intermediate result is large. As a result, running Phoenix++ directly on the Xeon Phi does not give good performance. In our framework, we propose various techniques to address these performance issues.

III. Optimized MapReduce Framework on Xeon Phi

In this section, we present our proposed MapReduce framework *MRPhi*, which is specially optimized for the Xeon Phi.

A. Overview

MRPhi adopts state-of-the-art techniques from the shared memory MapReduce framework and applies specific optimizations for the Xeon Phi coprocessor. Overall, we use a similar design as Phoenix++ [13] to implement the basic MapReduce workflow. There are two major techniques from Phoenix++ adopted in our framework, which are efficient combiners and different container structures.

Efficient combiners. Each map worker maintains a local container. When an intermediate key-value pair is generated by a map function, the reduce operator is immediately applied to that pair based on the local container. This process is performed using a *combiner*. After that, the partition is applied to each local container and multiple local containers are merged to a global container in the reduce phase.

Hash table and array containers. MRPhi supports two data structures for containers, which are hash tables and arrays. The array container is efficient when the keys are integers and in a fixed range.

More importantly, we propose four optimization techniques specific to the Xeon Phi as shown in Figure 1. The left part of the figure summarizes the flow of MapReduce framework while the right part shows the optimization techniques applied correspondingly. On the right part, the white boxes are the adopted combiners and containers from Phoenix++ while the dark boxes are our proposed optimization techniques.

- **Vectorization friendly map phase.** MRPhi implements the map phase in a vectorization friendly way, which

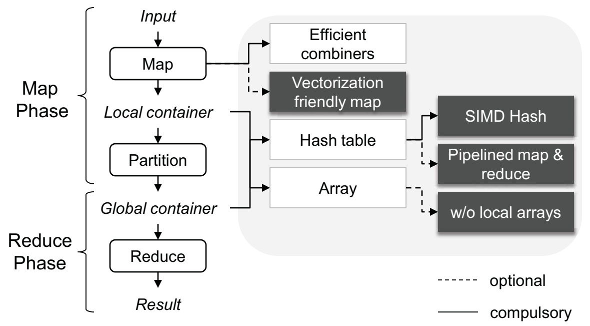

Fig. 1. Proposed techniques (in dark box) and their applicability in MRPhi.

clears the dependency between different map operations. By doing this, the Intel compiler is able to automatically vertorize multiple map operations to utilize VPUs.

- **SIMD parallelism for hash computation.** Hash computation is implemented employing SIMD parallelism by using SIMD instructions.
- **Pipelined execution for map and reduce phases.** In general, the user-defined map function contains heavy computation workload, while the reduce function has many memory accesses [13]. In order to better utilize the hardware resource with hyper-threading, the map and reduce phases are pipelined.
- **Eliminating local arrays.** For the array container, when the array is large, it introduces a number of performance issues due to local containers. We address these issues by using atomic operations on the global array instead of using local arrays.

Note that these techniques are not applicable for all MapReduce applications. Our framework is able to either automatically detect whether a specific technique is applicable or provide helpful suggestions to users at compilation time.

B. Vectorization Friendly Map Phase

Utilizing VPUs is critical to high performance on the Xeon Phi [7], [8]. For the MapReduce framework itself, except the hash computation (Section III-C), there is little chance to employ SIMD instructions. If the user defined map function contains loops, we then leave it to the compiler for auto-vectorization; otherwise we propose to implement the map phase in a vectorization friendly manner.

```
 1  //N: the number of map operations
 2  //elems: the input array
 3  #pragma ivdep
 4  for(i = 0; i < N; i++) {
 5      //the inlined map function
 6      map(data_t elems[i]) {
 7          ... //some computation
 8          emit_intermediate(key, value);
 9          ...
10      }
11  }
```

Listing 1. Vectorization for multiple map operations within a worker

Recall that in the map phase, each thread (or map worker) processes multiple map operations. We use directives to guide the compiler to vectorize multiple map operations. Listing 1 shows the basic idea. *emit_intermediate* is a system-defined function for combiners. The *#pragma ivdep* (line 3) tells the compiler to try to vectorize this for-loop.

This auto-vectorization can be effective if there is no dependencies between map operations (from line 6 to 10 in Listing 1). However, if multiple *emit_intermediate* operations are performed concurrently, the execution will cause the write conflict on a local container. This conflict exists for both the array and hash table containers.

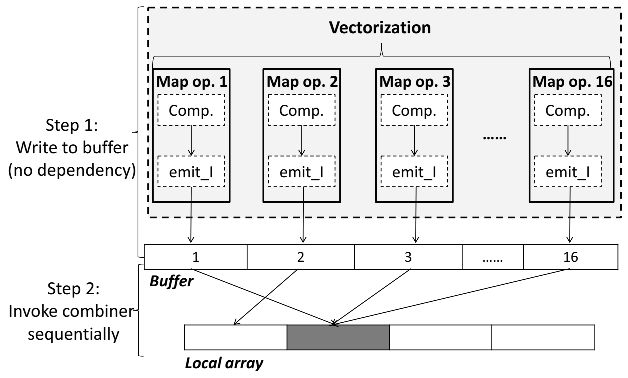

Fig. 2. A vectroization friendly algorithm without write conflict by using a buffer for the map phase.

We develop a new vectorization friendly algorithm to address the write conflict issue. Instead of performing the combiner for each intermediate pair generated by *emit_intermediate* immediately, we buffer a number of pairs. Writing to the buffer is independent for each map operation. When the buffer is full, we call the combiner for those pairs sequentially. Figure 2 demonstrates this vectorization friendly map. If there is no buffer, there will be write conflicts on the local array, which occurs in the dark element of local array. Our proposed approach avoids the write conflicts since each map operation writes to the buffer independently and the combiners are invoked sequentially. This way, auto-vectorization by the compiler is possible.

The decision on whether to use the vectorization friendly map depends on the ability of the map operation to be vectorized by the compiler. Users are suggested to adopt this technique if the map operations can be successfully vectorized based on the printout message from the Intel compiler about vectorization eligibility.

C. SIMD Parallelism for Hash Computation

Hash computation is a key component in the MapReduce framework. We observe that the auto-vectorization often fails due to the complex logic for hash computation. Thus, we choose to manually implement the hash computation using SIMD instructions.

SIMD hash computation for native data types is straightforward. The same procedure is applied to different input elements, which fully employs the SIMD feature. However, it is challenging to process variable-sized data types, such as text strings. Overall, various hash functions for strings, such as *FNV* [23] and *djb2* [24], have the similar workflow, which processes characters one by one. As a result, the workload of the hash computation for a given string depends on its length.

The challenge is how to efficiently handle strings with variable lengths. Furthermore, SIMD instructions only can be applied to VPU vector registers. How to pack data from memory to the vectors efficiently is another challenging problem.

We propose two SIMD hash computation algorithms, which are named as *SIMDH-Padding* and *SIMDH-Stream*.

1) SIMDH-Padding: It contains multiple rounds and each round processes characters from 16 consecutive strings in parallel. The intuition is within each round, we treat 16 strings as equal-length strings with the length L_r. L_r is equal to the number of characters in the longest string among these 16 strings. Then if a string is shorter than L_r, we pad this string with empty characters.

SIMDH-Padding has low control overhead due to its simplicity. It takes full advantage of SIMD instructions for computation and data packing. However, it underutilizes the computation resource due to the padding of empty characters.

2) SIMDH-Stream: This algorithm continuously feeds the SIMD units with strings. Input strings are treated as a stream. As long as a SIMD unit becomes available, the next entire string is fed to that unit for processing.

Compared with SIMDH-Padding, SIMDH-Stream introduces more complex control flow for data packing. It checks characters in the vector one by one for each iteration. If a character is zero, then it loads the first character of the next unprocessed string to the vector. There is no direct SIMD instructions for this process. However, SIMDH-Stream does not waste computing resources on empty characters. We evaluate both SIMDH-Padding and SIMDH-Stream in Section IV-A.

D. Pipelined Execution for Map and Reduce Phases

We propose to pipeline map and reduce phases on the Xeon Phi based on the MIMD hyper-threading execution. The motivation is that the map function defined by users usually performs heavy computation. But the reduce phase contains many memory accesses in which the major work is to construct the global container. We pipeline the computation-intensive map and memory-intensive reduce to improve the overall hardware resource utilization. For the array container implementation, the pipeline stage is not balanced as the reduce phase is too short. Thus, this technique is more effective for hash table container implementation than array container implementation.

The pipelined map and reduce are implemented using a producer-consumer model. The map and reduce workers are the producers and consumers, respectively. A key improvement is to make the local hash table work on a pre-allocated small buffer with a smaller size than the L2 cache. This is to improve the data locality of local hash table building.

If the final global hash table is very small, e.g., smaller than the L2 cache, the non-pipelined model will be more efficient. This is because the reduce phase will be too short to take advantage of pipelining because of the small hash table. On the other hand, the pipelined model introduces storage overhead. Our producer-consumer model is adaptive to the pipelined and non-pipelined models. Recall that we allocate a fix-sized buffer (smaller than the L2 cache) for the local hash table. If the final hash table is smaller than this buffer, no data will be fed to the reduce worker (the consumer) until the map phase is finished. This way, our pipelined model essentially degrades to a non-pipelined model as we expected.

E. Eliminating Local Arrays

In Phoenix++, each worker maintains a local container in the map phase. This design is efficient when the container size is small. However, it will introduce performance issues when the container becomes large. An alternative is to eliminate local containers and directly update the global container when the container size is large. This technique is applied to the array container only since it can be implemented using supported atomic data types. There are two major advantages.

Thread scalability. Due to the relatively small memory size on the Xeon Phi (8 GB), the thread scalability can be limited when using local arrays. As an extreme example of using Bloom filter in bioinformatics [25] (evaluated in Section IV), if the whole human genome is used, the local array size is around 3.7 GB. In such a case, only two threads can be used on the Xeon Phi employing local arrays.

Cache efficiency. When the array becomes large, random memory accesses on arrays cause poor data locality. Using the global array is able to improve the cache efficiency because of the ring interconnection. Specifically, when the local arrays are eliminated, the large global array is likely distributed across L2 caches of different cores. When a cache miss occurs on one core, the data may be copied from another core's L2 cache to avoid the expensive memory access. On the contrary, using local arrays cannot benefit from this hardware feature. Because a local array is only stored in a specific core's L2 cache but not shared across different L2 caches.

The decision on whether to eliminate local arrays is automatically decided by our framework to maximize cache efficiency. If the size of each local array is smaller than the L2 cache, then we keep these small local arrays as Phoenix++ does. Otherwise our framework will eliminate the local arrays.

IV. EXPERIMENTAL EVALUATION

Hardware setup. We conduct our experiments on an Intel Xeon Phi coprocessor 5110P. The hardware specification has been summarized in Section II-B.

TABLE I
BENCHMARK APPLICATIONS.

Application	Container	Applied optimization
Monte Carlo	Array (small)	Vectorization friendly map
Black Scholes	N/A	Vectorization friendly map
Word Count	Hash table	SIMD hash, pipelining
Reverse Index	Hash table	SIMD hash, pipelinine
Histogram	Array (large)	Eliminating local arrays
Bloom Filter	Array (large)	Eliminating local arrays

Benchmark applications. We choose six MapReduce applications as shown in Table I.

Implementation detail. MRPhi is developed using C++ and pthread. It natively runs on the Xeon Phi coprocessor. We organize the thread affinity in the scatter way such as thread i belong to core $(i \mod 60)$, where 60 is the number of cores. For evaluations, we only report the in-memory processing time of the MapReduce framework.

A. Performance Evaluation of Optimization Techniques

In this section, we evaluate the performance impact of our proposed techniques in details. By default, the number of threads per core is tuned to the number that can generate the best performance, unless specified otherwise.

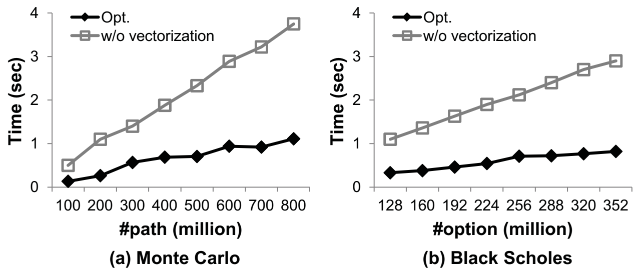

Fig. 3. Performance impact of vectorization friendly map for Monte Carlo (with the number of paths varied) and Black Scholes (with the number of options varied).

Vectorization friendly map. Figure 3 shows the performance impact of vectorization friendly map for Monte Carlo and Black Scholes with data size varied. It shows that the vectorization friendly map can improve the performance by 2.5-4.2X and 3.0-3.6X for Monte Carlo and Black Scholes, respectively. For these two applications, the map phase dominates the overall performance (>99%). Therefore the vectorization for the map phase can greatly improve the overall performance.

SIMD parallelism for hash computation. We first evaluate the performance of hash computation separately using a single thread. Figure 4(a) shows the performance result of pure hash computation with the input data size varied. We use the same data set as that used in the Word Count application. This shows that the SIMDH-Padding and SIMDH-Stream achieve a speedup of up to 2.8X and 2.2X over the scalar hash, respectively. Though SIMDH-Padding wastes computation resource due to the padding, it achieves better performance. The major reason is the control logic of SIMH-Padding is simpler, thus the overhead of data padding to vectors is also lower.

We further show the performance impact of using the SIMD hash for Word Count in Figure 4(b). It shows the overall performance is slightly improved (around 6%). The small improvement is because the overall performance is dominated by the memory latency rather than the hash computation [26].

Pipelined map and reduce. Word Count and Reverse Index are able to take advantage of pipelined map and reduce. We report the results of Reverse Index only and the Word Count has a similar conclusion. We use the data set in Phoenix++ for evaluations, which contains 78,355 files and 307,921 links in total. Figure 5(a) shows the elapsed time with the number of threads varied. It shows that the overall performance is improved by around 8.5%. We further decompose the time as shown in Figure 5(b). This shows that for the map and reduce phases only, the pipelining technique improves the performance by around 14%. However, due to the storage overhead, the memory cleanup phase of the pipelined map and reduce is more expensive and offsets the overall performance improvement.

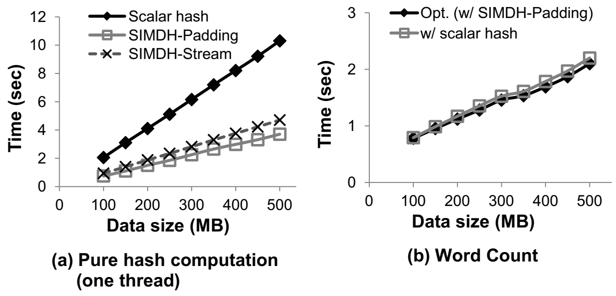

Fig. 4. The SIMD hash computation performance with the input data size varied. (a) The pure hash computation time (one thread). (b) The performance of Word Count with and without the SIMD hash computation.

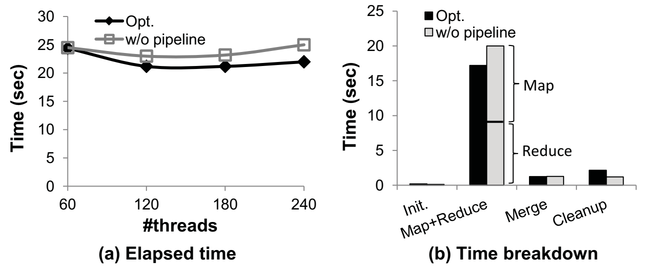

Fig. 5. Performance impact of pipelined map and reduce for Reverse Index. (a) Elapsed time (b) Time breakdown.

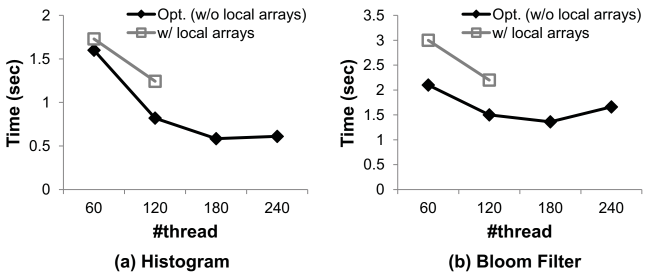

Fig. 6. Elapsed time of Histogram and Bloom Filter with the number of map threads varied. Histogram: 16 million unique keys and 256 million input elements; Bloom Filter: 30 million input elements, 300 million entries.

Eliminating local arrays. By eliminating local arrays, the thread scalability for the map phase can be improved. Figure 6 demonstrates such a scenario. The sizes of each local array are 64 MB and 40 MB for the Histogram and Bloom Filter, respectively. Figure 6 shows that the largest numbers of threads when using local arrays are 120 for Histogram and Bloom Filter, limited by the 8 GB memory. On the contrary, if local arrays are eliminated, more threads can be used. As a result, by eliminating local arrays, it achieves the speedup of up to 2.1X and 1.6X for Histogram and Bloom Filter, respectively.

B. MRPhi vs. Phoenix++ on the Xeon Phi

Now we show the end-to-end performance comparison between our MRPhi and Phoenix++ [13], which is state-of-the-art MapReduce framework on multicore. On the Xeon

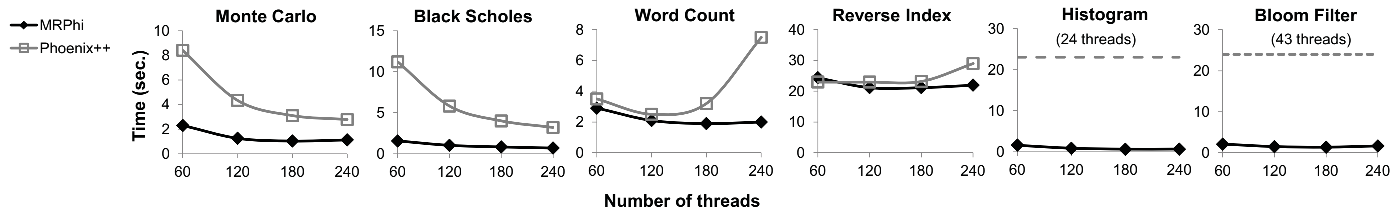

Fig. 7. Performance comparison between MRPhi and Phoenix++ on Xeon Phi. The horizontal axis is number of threads. The vertical axis is time (second).

TABLE II
DATA SETS FOR END-TO-END PERFORMANCE COMPARISON.

Application	Data set
Word Count	Input data size: 500 MB
Reverse Index	#files; 78,355 ; #links: 307,921 ; size: 1 GB
Monte Carlo	#paths: 800 million
Black Scholes	#options: 352 million
Histogram	#unique keys: 16 million; #elements: 256 million
Bloom Filter	#elements: 30 million; #entries: 300 million (Arabidopsis chromosome 1)

Phi, Phoenix++ also runs natively. We use large data sets for evaluations, which are summarized in Table II.

Figure 7 shows the overall performance comparison. The corresponding techniques used for each application have been summarized in Table I. Note that for a specific benchmark application, our framework is able to analyse the MapReduce code and apply accordingly the combination of optimization techniques. This is straightforward to applying for other big data applications. We used dash lines for Histogram and Bloom Filter as the largest numbers of threads for them are less than 60. For Monte Carlo and Black Scholes, which take advantage of vectorization in MRPhi, they are up to 2.7X and 4.6X faster than their counterparts in Phoenix++. For Word Count and Reverse Index that are based on the hash table, MRPhi can achieve a speedup of up to 1.2X. Furthermore, by eliminating local arrays, MRPhi is able to achieve a speedup of up to 38X and 18X for Histogram and Bloom Filter, respectively. In summary, MRPhi can achieve a speedup of 1.2X to 38X over Phoenix++ for various applications on the Xeon Phi.

V. CONCLUSION

In this paper, we present MRPhi, which is the first MapReduce framework optimized for the Intel Xeon Phi coprocessor. In MRPhi, in order to take advantage of VPUs, we develop a vectorization friendly technique for the map phase and SIMD hash computation. We also pipeline the map and reduce phases to better utilize the hardware resource. Furthermore, we eliminate local arrays to improve the thread scalability and data locality. Our framework is able to identify suitable techniques for a given application automatically. Our experimental results show that MRPhi can achieve a speedup of 1.2X to 38X over Phoenix++ for different applications by using our proposed optimization techniques.

REFERENCES

[1] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational joins on graphics processors," in *SIGMOD*, 2008.
[2] H. P. Huynh, A. Hagiescu, W.-F. Wong, and R. S. M. Goh, "Scalable framework for mapping streaming applications onto multi-gpu systems," in *PPoPP*, 2012.
[3] M. Lu, J. Zhao, Q. Luo, B. Wang, S. Fu, and Z. Lin, "Gsnp: A dna single-nucleotide polymorphism detection system with gpu acceleration," in *ICPP*, 2011.
[4] J. Castillo, J. Bosque, E. Castillo, P. Huerta, and J. Martinez, "Hardware accelerated montecarlo financial simulation over low cost fpga cluster," in *IPDPS*, 2009.
[5] J. He, M. Lu, and B. He, "Revisiting co-processing for hash joins on the coupled cpu-gpu architecture," in *VLDB*, 2013.
[6] "NVIDIA Cuompute Unified Device Architecture (CUDA)," http://www.nvidia.com/object/cuda_home_new.html.
[7] S. Pennycook, C. Hughes, M. Smelyanskiy, and S. Jarvis, "Exploring simd for molecular dynamics, using intel xeon processors and intel xeon phi coprocessors," in *IPDPS*, 2013.
[8] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, G. Chrysos, and P. Dubey, "Design and implementation of the linpack benchmark for single and multi-node systems based on intel xeon phi coprocessor," in *IPDPS*, 2013.
[9] "STAMPEDE: Dell PowerEdge C8220 culster with Intel Xeon Phi Coprocessors," http://www.tacc.utexas.edu/resources/hpc/stampede.
[10] "China's Tianhe-2 Supercomputer Takes No. 1 Ranking on 41st TOP500 List," http://www.top500.org/blog/lists/2013/06/press-release/.
[11] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004.
[12] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *HPCA*, 2007.
[13] J. Talbot, R. M. Yoo, and C. Kozyrakis, "Phoenix++: modular mapreduce for shared-memory systems," in *Proceedings of the second international workshop on MapReduce and its applications*, 2011.
[14] R. Chen, H. Chen, and B. Zang, "Tiled-mapreduce: optimizing resource usages of data-parallel applications on multicore with tiling," in *PACT*, 2010.
[15] Y. Mao, R. Morris, and M. F. Kaashoek, "Optimizing mapreduce for multicore architectures," *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep*, 2010.
[16] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *PACT*, 2008.
[17] J. A. Stuart and J. D. Owens, "Multi-gpu mapreduce on gpu clusters," in *IPDPS*, 2011.
[18] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin, "Mapcg: writing parallel program portable between cpu and gpu," in *PACT*, 2010.
[19] W. Fang, B. He, Q. Luo, and N. Govindaraju, "Mars: Accelerating mapreduce with graphics processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 4, pp. 608–620, 2011.
[20] L. Chen, X. Huo, and G. Agrawal, "Accelerating mapreduce on a coupled cpu-gpu architecture," in *Supercomputing*, 2012.
[21] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, "Fpmr: Mapreduce framework on fpga," in *FPGA*, 2010.
[22] M. de Kruijf and K. Sankaralingam, "Mapreduce for the cell broadband engine architecture," *IBM J. Res. Dev.*, vol. 53, no. 5, Sep. 2009.
[23] "FNV Hash," http://www.isthe.com/chongo/tech/comp/fnv/index.html.
[24] "Hash Functions," http://www.cse.yorku.ca/~oz/hash.html.
[25] L. Ma, R. D. Chamberlain, J. D. Buhler, and M. A. Franklin, "Bloom filter performance on graphics engines," in *ICPP*, 2011.
[26] S. Chen, A. Ailamaki, P. Gibbons, and T. Mowry, "Improving hash join performance through prefetching," in *ICDE 2004*, 2004.