Benchmarking Harp-DAAL: High Performance Hadoop on KNL Clusters

Langshi Chen[1] Bo Peng[1] Bingjing Zhang[1] Tony Liu[1] Yiming Zou[1] Lei Jiang[1]
Robert Henschel[2] Craig Stewart[2] Zhang Zhang[3] Emily Mccallum[3] Zahniser Tom[3] Omer Jon[3] Judy Qiu[1]
[1]School of Informatics and Computing, Indiana University
[2]UITS, Indiana University
[3]Intel Corporation
{lc37, pengb, zhangbj, xqiu, xl41}@indiana.edu
{yizou, jiang60, henschel, stewart}@iu.edu
{zhang.zhang, emily.l.mccallum, tom.zahniser, jon.omer}@intel.com

***Abstract*—Data analytics is undergoing a revolution in many scientific domains, and demands cost-effective parallel data analysis techniques. Traditional Java-based Big Data processing tools like Hadoop MapReduce are designed for commodity CPUs. In contrast, emerging manycore processors like the Xeon Phi have an order of magnitude greater computation power and memory bandwidth. To harness their computing capabilities, we propose the Harp-DAAL framework. We show that enhanced versions of MapReduce can be replaced by Harp, a Hadoop plug-in, that offers useful data abstractions for both high-performance iterative computation and MPI-quality communication, as well as drive Intel's native DAAL library. We select a subset of three machine learning algorithms and implement them within Harp-DAAL. Our scalability benchmarks ran on Knights Landing (KNL) clusters and achieved up to 2.5 times speedup of performance over the HPC solution in NOMAD and 15 to 40 times speedup over Java-based solutions in Spark. We further quantify the workloads on single node KNL with a performance breakdown at the micro-architecture level.**

***Keywords*-HPC, Xeon Phi, BigData**

I. Introduction

In recent years, the volume and variety of data have been collected at enormous rates. The data comes from sources ranging from massive physics experiments and instruments that read our DNA to a multitude of sensors that monitor our environment. It also comes from our digitized libraries, media streams, and personal health monitors. Many of the primary software tools that are used to do the large-scale data analysis were born in the Cloud. Big data processing frameworks like Apache Hadoop are designed to run on large-scale commodity CPU clusters connected by Ethernet. Although using large clusters of commodity servers is still the most cost-effective way to process petabytes or exabytes of data, the majority of data analytics jobs do not have tremendous workloads. Furthermore, machine learning algorithms become increasingly common and can easily fit into memory but require fine-grained parallelism for high performance. Modern scale-up servers using GPUs and Xeon Phi provide substantial processing, memory, and I/O capabilities. Therefore small or middle-sized clusters for Big Data analytics become an attractive approach.

In this paper, we investigate and re-design optimized software stacks to effectively utilize scale-up servers in the Cloud for machine learning and data analytics applications. We conduct extensive benchmarking on the emerging Intel's Many Integrated Core Architecture (MIC) based Xeon Phi Knights Landing (KNL). Our goal is to bridge the performance gap of Big Data tools and HPC systems. To our knowledge, we are the first to run high-performance Hadoop for machine learning applications on KNL manycore clusters.

Here, we propose a hybrid machine learning framework named *Harp-DAAL* (Figure 1), which interfaces Harp [1], a highly efficient Hadoop based communication library, and *DAAL* [2], a native Data Analytics Acceleration Library from Intel. Harp is an open source project, which is a plug-in to the Apache Hadoop framework, developed by Indiana University.

Harp has two distinctive functions: 1) Collective communication operations that are highly optimized for big data problems. 2) Efficient and innovative computation models for different machine learning problems. The original Harp project has its codebase written in Java, which is unable to leverage the capabilities of shared-memory HPC hardware. The central idea is to replace Java kernels by highly optimized native kernels or math libraries at the node level. We use Intel's *DAAL* as the low-level kernels for Harp on HPC platforms. *DAAL* is a library that aims to provide the users with highly optimized building blocks for data analytics and machine learning applications. For each of its kernels, *DAAL* has three modes: Batch Processing, Online Processing, and Distributed Processing. The open source code of *DAAL* only provides MKL/TBB based kernels for intra-node computation while leaving the inter-node communication of distributed processing to the users. This motivates us to design and build the *Harp-DAAL* framework and to perform further optimization as follows: Within intra-node communication we can co-design and implement kernels that fully exploit

[1]https://dsc-spidal.github.io/harp/
[2]https://github.com/01org/daal

the advantages of hardware architectures. Furthermore, for inter-node, we can minimize the overhead of data conversion and data transfer to improve the scalability of code. The rest of the paper is organized as follows: In Section II, we give an overview of existing hybrid frameworks in the domain of data analytics and machine learning. In Section III, we describe our *Harp-DAAL* interface, the techniques to reduce data conversion overhead; and three benchmark algorithms. In Section IV and Section V, we discuss the configuration of the experimentation and analyze the experimental results. Finally, we conclude our work in Section VI.

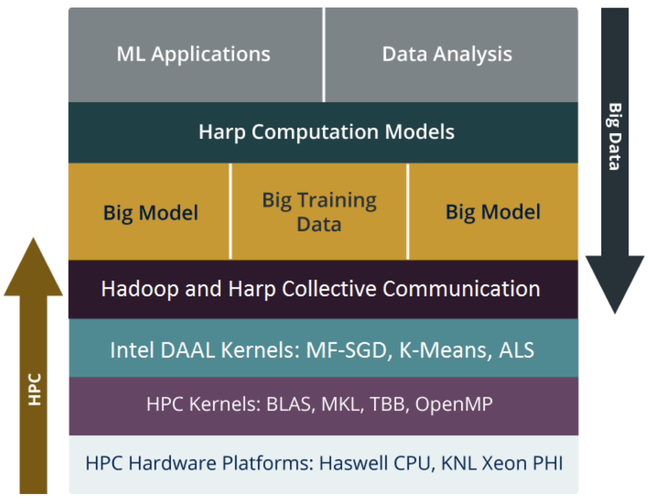


Figure 1. Harp-DAAL within HPC-BigData Stack

II. Related Work

For years, many scientific projects have tried to accelerate big data processing. Mars [1] combined the MapReduce framework with graphics processors and achieved 1.5 to 16 times performance improvements on a PC with G80 GPUs compared to a CPU-based counterpart on web data. *MLlib* [2] is a machine learning library built upon Apache Spark, which includes a variety of machine learning applications implemented using the Scala programming language. Although it is easy to use, the performance is not satisfying, especially on HPC clusters. Awan et al. [3] have characterized the performance of Spark MLlib on a 24-core processor, and they concluded that Spark MLlib suffers from imbalanced workload on threads and poor usage of the memory system. Therefore, another approach is to build machine learning libraries in C/C++, which are well suited for the HPC clusters. Such efforts include *Petuum*, developed to run machine learning algorithms efficiently on any hardware [4]. However, the C/C++ based solution demands relatively high programming skills, and in particular, the knowledge of multi-thread programming and architecture characteristics of multi-core/many-core processors. Other machine learning libraries build frameworks in a hybrid mode. In such libraries, users only need to know about high-level programming interfaces while letting the framework invoke proper HPC kernels written in C/C++ or FORTRAN. Currently Spark MLlib can utilize the highly optimized HPC library like BLAS and LAPACK interfaced by Breeze and netlib-java. Torch [3] is a scientific computing framework with broad support for machine learning algorithms from Facebook, which uses a script language called LuaJIT at the high level and the C/CUDA implementations on GPUs at the low level. Tensorflow [4] is a deep learning framework developed by Google, which has a Python interface to write dataflow graphs, and low-level implementations on different hardware devices like CPUs and GPUs.

III. Design and Implementation

A. Data Conversion within Harp-DAAL

Harp-DAAL has a two-level structure. At the top level, it runs a group of Harp mappers, which extend Hadoop's Java mapper. Unlike traditional MapReduce mapper, a Harp mapper holds data in main memory and invokes collective communication operations among different mappers.

At the bottom level, Harp-DAAL invokes native kernels written in C/C++, which use multi-threading programming paradigms such as OpenMP, TBB, and so forth. Since there are many highly optimized native kernels developed by the HPC community over the years, the invocation of HPC kernels can leverage the hardware resources better than the bottom level implementation of the original Harp applications, which, in contrast, use Java threads to execute jobs in parallel. However, the invocation of DAAL kernels from Harp is non-trivial work, hence, we must address two critical tasks. First, the undertaking of data conversion between Harp and DAAL, and second, the computation model of each application at both inter-node and intra-node levels. Data conversion between Harp and DAAL is critical to applications like MF-SGD, where a massive model needs to be synchronized in each iteration. For a case like this one, an inappropriate data conversion will significantly slow down the performance. Before discussing the data conversion, we first introduce the data structures of the Harp library and Intel's DAAL framework.

1) Data Structures of Harp and Intel's DAAL: Harp has a three-level hierarchy of data structures. The top level has the *Table* class, and a *Table* contains a certain number of *Partitions*. Each *Partition* consists of a *partition id* and a data container. Data containers could wrap up Java objects or primitive arrays, where the data is stored on heap memory managed by JVM. This design has two consequences. First, the data stored within a *Table* is scattered into different partitions, whose memory addresses are not contiguous,

[3]http://torch.ch
[4]https://www.tensorflow.org

while many native kernels with optimized memory access require contiguous memory allocation. Second, the memory addresses on the Java heap cannot be directly passed to native kernels because the JVM may change the objects' physical addresses during Garbage Collection.

On the contrary, the DAAL library consists of two modules.

- *Native Kernels*, which includes the implementation of algorithms and data structures in C/C++
- *Interface*, the API written in Java and Python to access *Native Kernels*

The native kernels are in charge of the computational work while the Java/Python APIs allow users to construct their applications without knowing the low-level implementations. In Harp-DAAL, data conversion happens between the Harp Java codes and the DAAL Java/Python APIs. Thus, the dataflow of the Harp-DAAL framework is from the Harp Java code to the DAAL Java APIs, and finally, to the DAAL native kernels. For instance, the *HomogenNumericTable* provides users two ways to store data.

- *ArrayImplm* stores the data at Java heap side
- *ByteBufferImplm* stores the data on the native side using DirectByteBuffer to read and write data between Java and native side.

Within the *ArrayImplm* storage, DAAL Java APIs hold data in the Java heap space, and it creates a data structure called *JavaNumericTable* for the native kernels to access such data. Whenever a native kernel requests the data in *JavaNumericTable*, it will callback to a member function of *HomogenNumericTable* at the Java side to trigger the data transfer via JNI functions and the DirectByteBuffer class. Most of the DAAL's data structures support this way of storing data. However, within the *ByteBufferImplm* storage, the Java APIs allocate an empty array in the native memory space, which makes it the user's responsibility to read and write the data between Java heap memory and allocated native memory. In the current DAAL release version as of 2017, only *HomogenNumericTable* supports this data storage method. Native kernels can only manipulate data that is stored in the native memory space, using an auxiliary data structure named *MicroTable* to retrieve specific rows or columns from a *NumericTable*.

We design two data conversion operations in Harp-DAAL. The first operation named *JavaBulkCopy* starts from the Harp side and copies data from a Harp *Table* to a *HomogenNumericTable* via the *ByteBufferImplm*. The second operation named *NativeDiscreteCopy* starts from a native kernel and transfers data from the DAAL Java APIs to native kernels via the *ArrayImplm*. Figure 2 (a) and (b) explains the work-flow of the two data conversion operations.

2) JavaBulkCopy: There are two steps in *JavaBulkCopy*

- Create a large DirectByteBuffer, and copy data from a Harp *Table* into the DirectByteBuffer
- Write the content of DirectByteBuffer into a *HomogenNumericTable*.

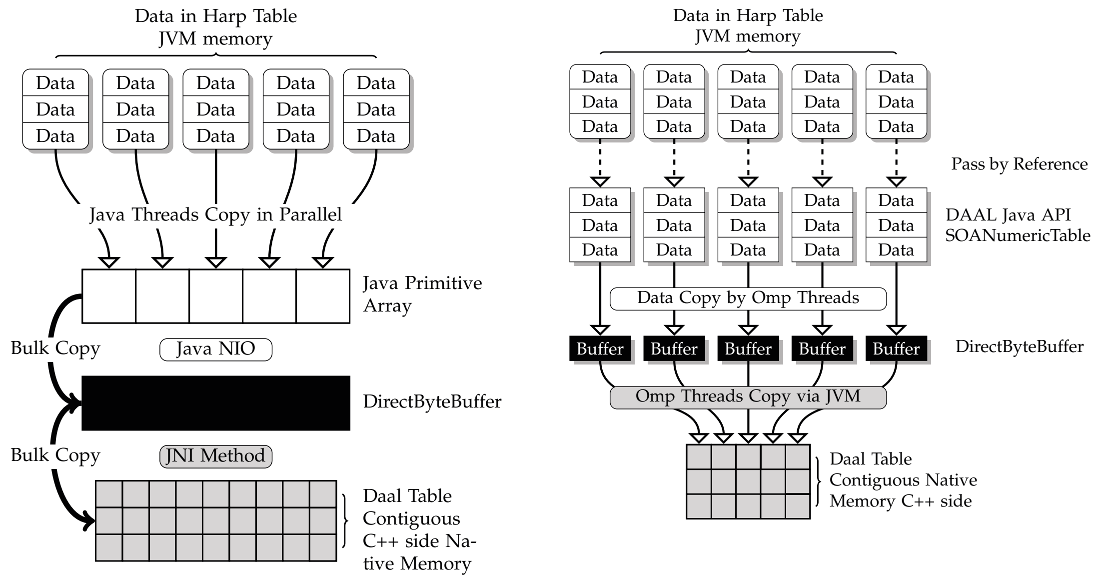
First, we use the *java.lang.Thread* class to do a parallel data copy from JVM heap memory of a Harp *Table* to a memory block within DirectByteBuffer. Second, a member function named *releaseBlockOfRows* from *HomogenNumericTable* does a bulk data copy from DirectByteBuffer to the native memory allocated in a *HomogenNumericTable*. The advantage of *JavaBulkCopy* is that the data within *HomogenNumericTable* is contiguous, which favors many of the DAAL algorithms such as K-means, and usually results in more efficient cache usage. However, the downsdies are two-fold. First, the maximal size of a single DirectByteBuffer is limited to 2 GB. Second, the scheduling of Java threads is not as efficient as the scheduling of OpenMP and TBB.

3) NativeDiscreteCopy: In *NativeDiscreteCopy*, the data is stored in Java heap memory and DAAL uses a C++ class named *JavaNumericTable* to expose the data to native kernels. In *NativeDiscreteCopy*, a thread from a native kernel will be attached to a C++ pointer, which is a member of *JavaNumericTable* and points to a JVM object via JNI. The thread then calls Java functions of *NumericTable* to copy data from Java heap memory back to native memory by using DirectByteBuffer. Since the JVM pointer allows concurrent access from different threads, we can use OpenMP or TBB threads to copy data in parallel as shown in Figure 2 (b). The advantage of using *NativeDiscreteCopy* is to reduce the size of DirectByteBuffer, because each thread can reuse the assigned buffer to copy different data. The downside is the exposure of the JNI interface and low-level implementations to users. Deciding which memory copy operation to use depends on the data structures and model synchronization operations.

B. Benchmark Algorithms

We select three typical learning algorithms to evaluate our framework: 1) K-means Clustering (K-means), a computation-bounded algorithm; 2) Matrix Factorization by Stochastic Gradient Descent (MF-SGD), a computation-bounded and communication-bounded algorithm; 3) Alternating least squares (ALS), a communication-bounded algorithm.

1) K-means Clustering: K-means is a widely used clustering algorithm in the machine learning community. It computes the distance between each training point to every centroid, re-assigns the training point to the new cluster and re-computes the new centroid of each cluster at each iteration. Harp-DAAL-Kmeans is built upon Harp's original K-means parallel implementation, with a computation complexity for each iteration of $\mathcal{O}(|\Omega|KM)$, where Ω is the set of training samples, K is the feature dimension of a training point, and M is the number of centroids. Harp-DAAL-Kmeans uses the regroup-allgather operation [5] to synchronize the model, i.e. centroids, among each mapper. Harp-DAAL-Kmeans uses



(a) JavaBulkCopy

(b) NativeDiscreteCopy

Figure 2. (a) JavaBulkCopy; (b) NativeDiscreteCopy

DAALs K-means kernel, where the computation of point-centroid distance is implemented by BLAS-level 3 matrix-matrix operations. This optimization significantly increases the computation intensity, resulting in highly vectorized codes against the original Harp-Kmeans.

2) MF-SGD: Matrix Factorization based on Stochastic Gradient Descent (MF-SGD) is commonly used in recommender systems [6], where it aims to factorize a sparse matrix into two dense model matrices W and H. The computation complexity for each iteration is $\mathcal{O}(|\Omega|K)$, where Ω is the set of training samples, and K is the feature dimension of a training point. Previous work such as [7] concentrates on the single-node shared memory optimization for MF-SGD. For distributed memory system, we have already implemented a pure Java version within the Harp framework [5], and we re-implement it by invoking native kernels in our hybrid Harp-DAAL framework.

The implementation of MF-SGD consists of two levels. At the inter-node level, we use a rotation operation to synchronize the model distributed among the nodes [8]. At the intra-node level, we choose an asynchronous operation to update the model manipulated by different threads. By eliminating the lock and wait problems at the thread level, we are able to relieve the load balancing problems caused by uneven distribution of training points in each row and column.

3) ALS: Alternating least squares (ALS) is another popular algorithm to decompose rating matrices in recommender systems. The algorithm has a computation complexity for each iteration of $\mathcal{O}(|\Omega|K^2 + (m+n)K^3)$. Here Ω is the set of training samples, K is the feature dimension, m is the row number of the rating matrix, and n is the column number of the rating matrix. Unlike MF-SGD, ALS alternatively computes models W and H independently of each other. The implementation of ALS in our Harp-DAAL framework chooses the regroup-allgather operation to interface the DAAL-ALS kernels based on the work of Zou et al. [9]

IV. Experimentation

In the experiments, we compared the performance of the following six implementations:

- Harp-DAAL-Kmeans
- Spark-Kmeans
- Harp-DAAL-SGD
- NOMAD-SGD
- Harp-DAAL-ALS
- Spark-ALS

In addition to the three applications from Harp-DAAL, we chose three other applications to compare with in the test. Spark-Kmeans and Spark-ALS are pure Java applications from Apache Spark (Section II). NOMAD-SGD is a distributed MF-SGD application developed by Yun et al. [10] in C/C++ and MPI.

A. Hardware Platform

We conducted experiments on a cluster of Intel's Xeon Phi processor 7250 codenamed Knights Landing (KNL). Table I gives the specification of one KNL node. Compared to Intel's Xeon processor family, KNL has three advantages: 1) A high number of physical cores, 2) Up to 136 AVX-512 Vector

Table I
SPECIFICATION OF XEON PHI 7250 KNL

Cores		Memory		Node Specs		Misc Specs	
Cores	68	DDR4	190 GB	Network	Omni-path	Instruction Set	64 bit
Base Freq	1.4GHz	MCDRAM	16 GB	Peak Port Band	100 Gbps	IS Extension	AVX512
L1 Cache	2 MB	DDR4-Band	90 Gbps	Socket	1	Max Threads	271
L2 Cache	34 MB	MCDRAM-Band	400 Gbps	Disk	1 TB	VPUs	136

Table II
DATASETS USED IN K-MEANS, MF-SGD, AND ALS

Dataset	Kmeans-Single	Kmeans-Multi	Movielens	Netflix	Yahoomusic	Enwiki	Hugewiki
#Training	5 million	20 million	9 million	99 million	252 million	609 million	3 billion
#Test	none	none	698 thousand	1 million	4 million	12 million	365 million
#centroid	10 thousand	100 thousand	none	none	none	none	none
Dim	100	100	40	40	100	100	1000
λ	none	none	0.05	0.05	1	0.01	0.01
γ	none	none	0.003	0.002	0.0001	0.001	0.004

Processing Units (VPU). Each VPU could simultaneously compute 8 double or 16 float operations in parallel by enabling Intel's AVX-512 instruction set extension. 3) On-chip high bandwidth memory named MCDRAM, whose bandwidth reaches 400 Gbps. Therefore, KNL favors applications that maximally leverage the intra-node threads parallelism, codes vectorization, and memory bandwidth usage. To compare the utilization of KNL's features, we employ Intel VTune Amplifier to do the micro-benchmark profiling.

B. Dataset

A variety of datasets are used to examine the performance of implementations. Table II describes the details including sizes and parameters. For K-means, we have used synthetic datasets, a small one for single node tests and a large one for multi-node scaling tests. For MF-SGD and ALS, we used the same datasets as in related work.

V. RESULTS AND ANALYSIS

A. Single Node Performance

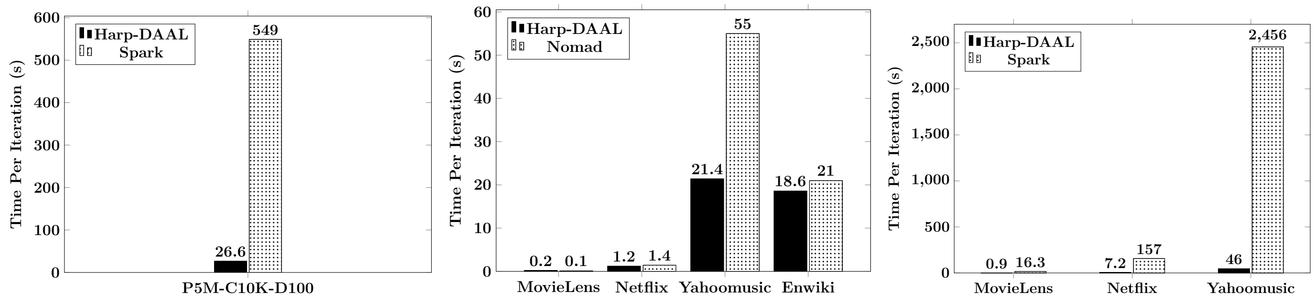
We first evaluated the performance of Harp-DAAL framework on a single KNL node. We used 64 cores out of the total 68 cores because the self-booted KNL node requires several cores to run the OS system. Each physical core runs one thread, which implies a total of 64 threads for the applications. The metric includes the execution time per training iteration for K-means, MF-SGD, and ALS respectively. The time is averaged over a certain number of iterations to avoid potential cold start anomalies.

In Figure 3, we find that Harp-DAAL achieves the best performance among the three frameworks. For K-means and ALS, it runs 20x to 50x faster than Spark. This result is not surprising because both K-means and ALS contain matrix-matrix computations that benefit from the optimized MKL native kernels invoked by Harp-DAAL. For MF-SGD, we still achieve comparable performance to NOMAD on datasets Movielens, Netflix, and Enwiki. On Yahoomusic, we even have a 2x speedup, which exemplifies the intra-node optimization brought by DAAL kernels.

B. Multi-node Performance

To compare multi-node performance, we used 60 instead of 64 threads per node in computation and reserved the rest of the threads to communication tasks that are overlapped in a pipeline design of applications like MF-SGD. The scalability for K-means and MF-SGD is not measured against one node because the datasets are too large to fit into the memory of single node. Instead, we assume that the scalability from one node to 10 nodes are linear, and we measure the strong scalability over 10 nodes. Figure 4 decomposes the execution time of Harp-DAAL applications on multiple nodes into three components: 1) Computation time on local nodes, 2) Data conversion time on local nodes, and 3) Data communication time among remote nodes. For K-means, the computation time is dominant, taking more than 80% of the total execution time. When it scales from 10 nodes to 20 nodes, the communication ratio decreases because the model volume on each node decreases, and Harp regroup-allgather operations favor small communication data. The communication ratio slightly increases from 20 nodes to 30 nodes, which is due to the insufficient computation work on each node that causes a reduced computation ratio.

In Figure 5 (a), Harp-DAAL-Kmeans runs 15x to 40x faster than Spark Kmeans, and shows better scalability from 10 nodes to 20. Beyond 20 nodes, due to low computation workload on local nodes, the scalability of Harp-DAAL-Kmeans drops while Spark-Kmeans still has substantial computation workload. It suggests that Spark-Kmeans is more computation-bounded than Harp-DAAL-Kmeans be-



(a) K-means (b) MF-SGD (c) ALS

Figure 3. The execution time (per iteration) on a single KNL node. Each subgraph compares two implementations on the same algorithm with different datasets. (a) K-means dataset: 5 million points, 10 thousand centroids, 1000 feature dimension; (b) MF-SGD dataset: 1) Movielens 9 million points, 40 feature dimension, 2) Netflix 100 million points, 40 feature dimension, 3) Yahoomusic 250 million points, 1000 feature dimension, 4) Enwiki 600 million points, 100 feature dimension; (c) ALS dataset: 1) Movielens 9 million points, 40 feature dimension, 2) Netflix 100 million points, 40 feature dimension, 3) Yahoomusic 250 million points, 100 feature dimension

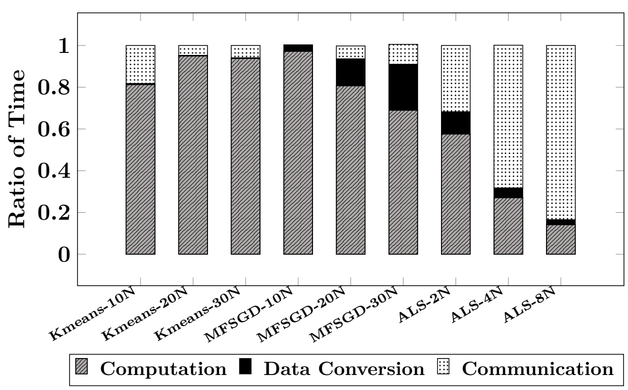


Figure 4. Breakdown of execution time of Harp-DAAL Applications on multiple KNL nodes 1) K-means dataset:20 million points, 100 thousand centroids, 100 feature dimension; 2) MF-SGD dataset, Hugewiki, with 3 billion training points, 1000 feature dimension; 3) ALS dataset, Yahoomusic, with 250 million training points, 100 feature dimension

cause strong scalability reflects how an implementation is bounded by local computations. Since Harp-DAAL-Kmeans invokes fast MKL kernels at the low level, it is much less bounded by computation time.

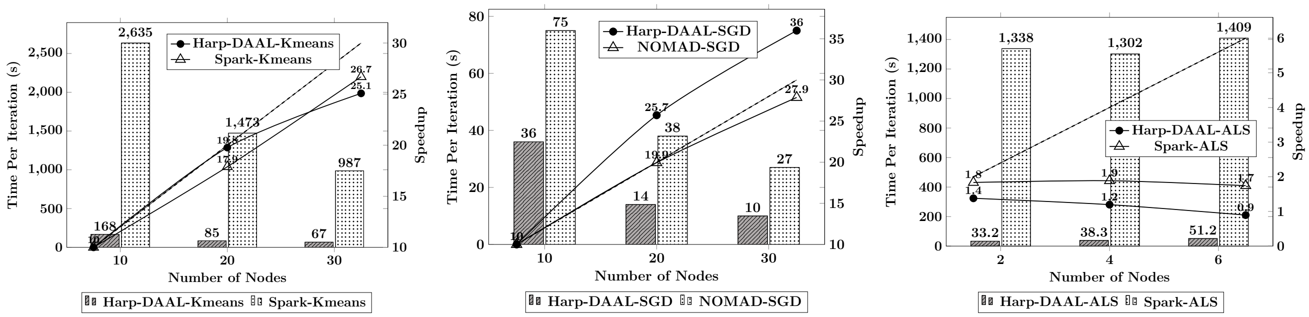
For MF-SGD, Figure 5 (b) shows that Harp-DAAL-SGD runs 2.5x faster than NOMAD-SGD, and it even achieves super-linear scalability on 20 nodes and 30 nodes, which is also better than the scalability of NOMAD-SGD. There are two reasons why Harp-DAAL-SGD has a super-linear speedup: First, Harp-DAAL-SGD has its native kernels implemented by OpenMP, and a small number of training data may significantly reduce the scheduling overheads. Second, the native kernel for intra-node computation uses asynchronous shared-memory data access operations, which favors sparse training points that has less conflicts in accessing the same data.

For ALS, Figure 4 shows that the communication time already takes up more than 50% of the execution time on four nodes. This means that ALS is not bounded by local computation, and therefore both Harp-DAAL-ALS and Spark-ALS have poor strong scalability in Figure 5 (c). However, Harp-DAAL-ALS still has around 25x to 40x speedup when compared to Spark-ALS because it invokes highly efficient MKL kernels.

C. Micro-Benchmark

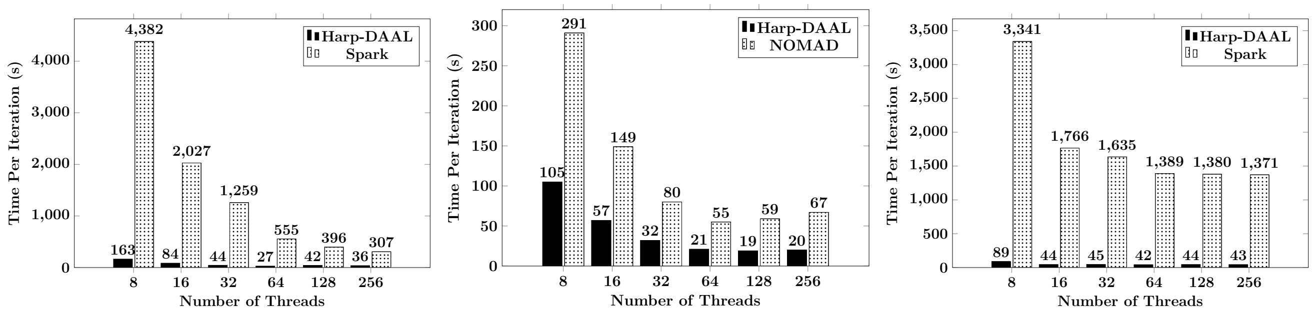
1) Performance Breakdown on a KNL Single Node: The execution time breakdown of all benchmarks in both frameworks is shown in Figure 7. Compared to another C++ based SGD implementation (Nomad-SGD), our SGD with Harp-DAAL significantly reduces execution time. This is because Harp-DAAL-SGD is able to fully take advantage of AVX-512 to reduce the retiring instruction number by packaging multiple floating point operations into one SIMD instruction. One AVX-512 instruction may cause multiple simultaneous L1 cache accesses, so that Harp-DAAL-SGD also improves the L1 cache bandwidth utilization. Compared to Harp-DAAL, our C++ and Java hybrid framework, the pure Java framework, Spark, inflates the executed instruction number by at least 10 times on K-means and ALS. Therefore, the retiring of benchmarks with Harp-DAAL only takes at most 10% of that with Spark. Spark is only developed upon the Java virtual machine which can barely benefit from AVX-512. Therefore, its non-vectorized code only generates memory requests with poor temporal locality, hardly saturating the large bandwidth supplied by the MCDRAM. The serialized memory accesses of Spark framework substantially prolong the time stall along the memory hierarchy.

2) Thread Scaling: Figure 6 shows the performance of K-means, MF-SGD and ALS with varying numbers of threads on one single node. We noticed that by increasing the number of threads, the total computing power of KNL increases.



(a) K-means (b) MF-SGD (c) ALS

Figure 5. Strong Scaling on multiple KNL nodes. The bars refer to the execution time of each application. Dashed line is the linear speedup of multiple nodes to a single node. Solid lines are the measured speedup of applications on multiple nodes against a single node. E.g., the value 25.7 of the marker for lines Harp-DAAL-SGD in (b) illustrates a 25.7 speedup computed as T(1)/T(20). (a) K-means dataset:20 million points, 100 thousand centroids, 100 feature dimension; (b) MF-SGD dataset, Hugewiki, with 3 billion training points, 1000 feature dimension; (c) ALS dataset, Yahoomusic, with 250 million training points, 100 feature dimension



(a) K-means (b) MF-SGD (c) ALS

Figure 6. Strong Scaling on Threads of a single KNL (a) K-means dataset: 5 million points, 10 thousand centroids, 100 feature dimension; (b) MF-SGD dataset: Yahoomusic with 250 million training points, 1000 feature dimension; (c) ALS dataset, Yahoomusic, with 250 million training points, 100 feature dimension

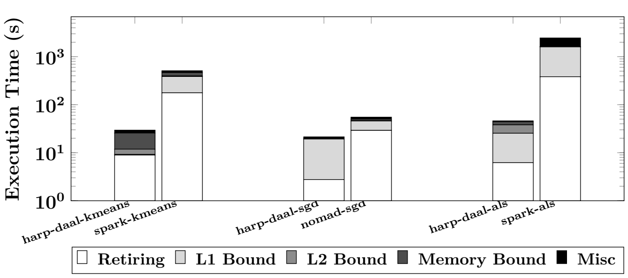


Figure 7. Breakdown of execution time on a single KNL node. X Bound indicates the time stall caused by X-event. Retiring means the execution time of instructions. Misc represents the time stall triggered by instruction cache miss, branch prediction miss, TLB miss and all other micro-architecture events.

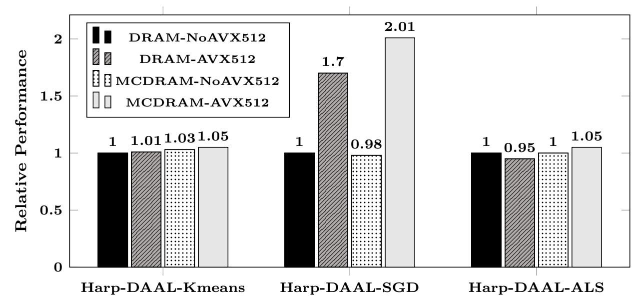


Figure 8. Relative Performance by enabling AVX-512 and MCDRAM. The baseline performance is the configuration without AVX-512 and MCDRAM, which is set to 1. The relative performance of other configurations is their acceleration of execution time compared with the baseline performance

However, communication between cores intensify and cache capacity per thread also drops significantly. Therefore, more threads do not necessarily indicate shorter execution time on a KNL node. K-means, MF-SGD, and ALS encapsulated by Harp-DAAL achieve the best performance with 64, 128, and 64 threads respectively. Since Spark cannot fully utilize AVX-512, both K-means and ALS with Spark have to execute more instructions to do the same job. Both prefer 256-thread configuration to relieve their bottleneck on instruction retiring.

3) AVX-512 and MCDRAM: Figure 8 exhibits the performance improvement achieved by AVX-512 and MCDRAM in our Harp-DAAL framework. Instead of configuring MCDRAM as a hardware managed cache, we have deployed it as a parallel component to DDR4 in the main memory system. Through the *numactl* command, we can use either DDR4-based main memory or MCDRAM-based main memory. All bars are normalized to the scheme compiled with disabling vectorization and with DDR4-based main memory. By enabling AVX-512, Kmeans and ALS do not reduce execution time significantly. This is because, although we compiled them without vectorization, kernels in Kmeans and ALS invoke functions from Intel Math Kernel Library which are fully optimized with AVX-512. AVX-512 improves the performance of SGD by 70%, since VPUs on KNL compute matrix multiplications with much higher throughput. Also, only enabling MCDRAM does not obviously boost the performance of all three benchmarks. K-means, non-vectorized SGD and ALS have relatively high L2 cache hit rate; therefore, they cannot benefit from large memory bandwidth provided by MCDRAM. By enabling both AVX-512 and MCDRAM, SGD improves the performance by 101%, since AVX-512 instructions may generate multiple memory accesses in one cycle and the MCDRAM memory bandwidth can be fully utilized.

VI. Conclusion

We designed and implemented Harp-DAAL that enables Hadoop on cloud servers with manycore KNL processors. Many machine learning applications can be implemented with MapReduce-like interfaces with significantly boosted performance caused by scaling up. Through evaluating computation and communication-bounded applications, we show that Harp-DAAL combines advanced communication operations from Harp and high performance computation kernels from DAAL. Our framework achieves 15x to 40x speedups over Spark-Kmeans and 25x to 40x speedups to Spark-ALS. Compared to NOMAD-SGD, a state-of-the-art C/C++ implementation of the MF-SGD application, we still get a factor of 2.5 improvement in performance. An interesting future direction will be to compare our current work with other hardware, including Haswell and GPUs, and develop high performance machine learning libraries. The code and documentation of Harp-DAAL framework can be found at https://dsc-spidal.github.io/harp/.

Acknowledgments

We gratefully acknowledge generous support from the Intel Parallel Computing Center (IPCC) grant, NSF OCI-114932 (Career: Programming Environments and Runtime for Data Enabled Science), CIF-DIBBS 143054: Middleware and High Performance Analytics Libraries for Scalable Data Science. We appreciate the support from IU PHI, FutureSystems team and ISE Modelling and Simulation Lab.

References

[1] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a MapReduce framework on graphics processors," *International Conference on Parallel Architectures and Compilation Techniques*, pp. 260–269, 2008. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1454115.1454152{%}5Cnhttp://dl.acm.org/citation.cfm?id=1454152

[2] X. Meng, J. Bradley, S. Street, S. Francisco, E. Sparks, U. C. Berkeley, S. Hall, S. Street, S. Francisco, D. Xin, R. Xin, M. J. Franklin, U. C. Berkeley, and S. Hall, "MLlib : Machine Learning in Apache Spark," *The Journal of Machine Learning Research*, vol. 17, pp. 1235–1241, 2016.

[3] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade, "Performance Characterization of In-Memory Data Analytics on a Modern Cloud Server," *Proceedings - 2015 IEEE 5th International Conference on Big Data and Cloud Computing, BDCloud 2015*, pp. 1–8, 2015.

[4] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: A New Platform for Distributed Machine Learning on Big Data," *Kdd*, vol. XX, no. X, p. 15, 2015. [Online]. Available: http://www.cs.cmu.edu/{~}./seunghak/petuum{_}kdd15.pdf

[5] B. Zhang, Y. Ruan, and J. Qiu, "Harp: Collective communication on Hadoop," *Proceedings - 2015 IEEE International Conference on Cloud Engineering, IC2E 2015*, pp. 228–233, 2015.

[6] Y. Koren, R. Bell, and C. Volinsky, "Matrix Factorization Techniques for Recommender Systems," *Computer*, vol. 42, no. 8, pp. 42–49, 2009.

[7] W.-S. Chin, Y. Zhuang, Y.-C. Juan, and C.-J. Lin, "A Fast Parallel Stochastic Gradient Method for Matrix Factorization in Shared Memory Systems," *ACM Transactions on Intelligent Systems and Technology*, vol. 6, no. 1, pp. 1–24, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2745393.2668133

[8] B. Zhang, B. Peng, and J. Qiu, "Model-centric computation abstractions in machine learning applications," *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond - BeyondMR '16*, vol. 09, pp. 1–4, 2016. [Online]. Available: http://doi.acm.org/10.1145/2926534.2926539{%}5Cnhttp://dl.acm.org/citation.cfm?doid=2926534.2926539

[9] R. P. Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, "Large-Scale Parallel Collaborative Filtering for the Netix Prize," *Lecture Notes in Computer Science*, vol. 5034, pp. 337–347, 2010. [Online]. Available: http://link.springer.com/content/pdf/10.1007/978-3-642-14355-7.pdf

[10] H. Yun, H.-F. Yu, C.-J. Hsieh, S. V. N. Vishwanathan, and I. Dhillon, "NOMAD: Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion," *Pvldb*, vol. 7, no. 11, pp. 975—-986, 2014. [Online]. Available: http://www.vldb.org/pvldb/vol7/p975-yun.pdf