Harp-DAAL: A High Performance Data-Intensive Machine Learning Framework

Langshi Chen and Judy Qiu
School of Informatics and Computing
Indiana University

1.Introduction

Intel® Data Analytics Acceleration Library (DAAL) is a library from Intel that aims to provide the users of some highly optimized building blocks for data analytics and machine learning applications. The latest version is the 2017 beta version that is already made open source on the GitHub homepage[1]. For each of its kernel, DAAL has three modes:

- A *Batch Processing* mode is the default mode that works on an entire dataset that fits into the memory space of a single node.
- An *Online Processing* mode works on the blocked dataset that is streamed into the memory space of a single node.
- A *Distributed Processing* mode works on datasets that are stored in distributed systems like multiple nodes of a cluster.

Nowadays, many data analytics and machine learning problems contain millions or billions of training data and parameter data, it is obvious that the *Distributed Processing* mode is the only choice for many applications. Within DAAL's framework, the communication layer of the *Distributed Processing* mode is left to the users, which could be Hadoop, Spark, MPI, or any of the user-defined middleware. The goal of our project is thus to integrate Harp (a Hadoop plugin) into the *Distributed Processing* mode of DAAL. Harp has the following advantages:

- Harp has MPI-like collective communication operations that are highly optimized for big data problems.
- Harp has efficient and innovative computation models for different iterative machine learning problems.

The original Harp project has all of its codes written in Java, which is a common choice in the Hadoop ecosystem. The downside of the pure Java implementation is its inability of benefiting from HPC hardware due to limitations brought by JVM. Since many-core architecture devices, such as GPU and Xeon Phi, are increasingly equipped by data centers, it is desirable to run Hadoop applications within high performance native kernels at hardware level. The challenges of such an interface between Harp and DAAL will be discussed in the following sections.

2. Data Structure of Harp and DAAL

The major roadblock of the interface between Harp and DAAL is their data structures. Harp is written in Java while DAAL has most of its codes written in C++. Fortunately, DAAL already provides users of a Java API, which invokes the C++ codes at low level. Nevertheless, Harp and DAAL have different data structures shown in Figure 1.

---

[1] https://github.com/01org/daal

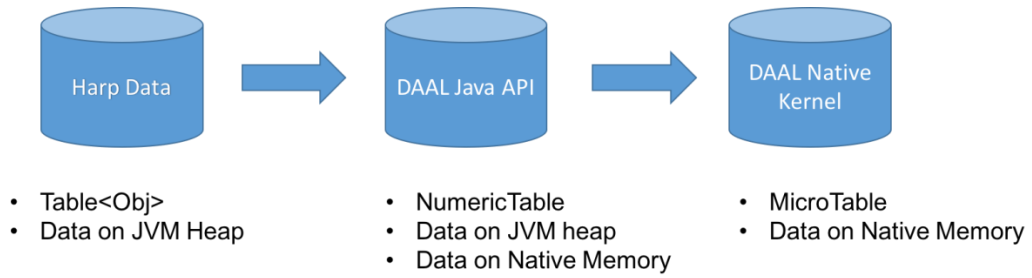

Figure 1. Data structure of Harp and DAAL

Harp defines its own three-level data structure. At the top level is its *ArrTbale*, which extends the Table class of Java. At the middle level is the *ArrPartition*, which owns an *identifier* and an *Array*. At the low level is the *Array* that derives from the primitive Array class of Java. Therefore, an *ArrTable* of Harp could contains a substantial number of *ArrPartition*, and the data resides on non-consecutive memory space.

DAAL Java API also has a variety of data structures. In general, it has *data source*, *data dictionary*, *NumericTable*, *matrix*, and so forth. E.g., *NumericTable* is a major data structure used in DAAL's algorithms. *NumericTable* has many sub-classes such as *HomogenNumericTable*, *AOSNumericTable*, *SOANumericTable*. In *HomogenNumericTable*, the data is stored in consecutive chunks of memory space, while the *AOSNumericTable* has its data stored in non-contiguous memory space. For some kinds of *NumericTable*, users could also decide whether the data's memory space is allocated on the Java side or on the C++ side. There are pros and cons for both of the two ways, and we will investigate them in the following sections.

DAAL native kernels uses a MicroTable structure to retrieve data stored in NumericTable, either the entire data, or several rows or columns of data. If the data in NumericTable is already stored at C++ side, the MicroTable gets a reference to that memory space, otherwise, MicroTable will create a memory workspace and use DirectByteBuffer to copy data from JVM heap memory to native memory.

Because of the different data structures, the interface needs to address the data conversion problem, and consequently, will cause additional time overhead. Unless we modify the data structures of Harp and DAAL, we can only use some multithreading copy to reduce the conversion time overhead. The JNI interface used in DAAL's Java APIs will also give us some challenges that will be discussed in the following sections.

3. Two Case Studies

We select two algorithms, K-means and Matrix Factorization by Stochastic Gradient Descent (MF-SGD), to test our interface implementation. K-means represents the category of computation-intensive problems while MF-SGD represents the category of memory-intensive applications. For each of them, we will compare their performance at different HPC platforms.

3.1 Harp-DAAL-Kmeans

The interface between Harp's K-means and DAAL's K-means is quite straightforward. At the inter-node level, it chooses an allreduce computation model shown in Figure 2, where each node (mapper) keeps a local copy of the model data (centroids), and it updates the model data by synchronizing it with all the other nodes (mappers). The benefits of using allreduce lies in its simplicity of implementation. When the

centroids data size is not large, the allreduce will not cause significant inter-node synchronization overhead.

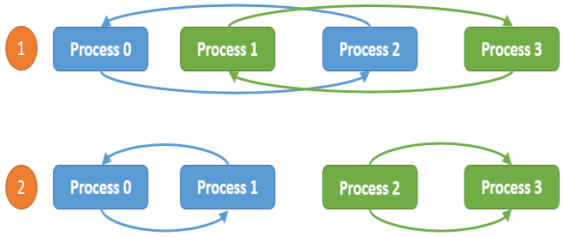

Figure 2. Allreduce computation model used in Harp-DAAL-Kmeans

At the intra-node level, Harp-DAAL-Kmeans invokes the computation kernel written in DAAL's K-means module, which is in the form of matrix-matrix multiplication. By using matrix-matrix operations, the computation intensity is improved (BLAS-3 level) and it is well suited for platforms with substantial number of VPUs. The implementation of Harp-DAAL-Kmeans is relatively straight forward

```
//----- Harp codes to get pointPartitions ----------------//
//------ start convert data from Harp to DAAL -----------

HarpDaalParallelConvert converter = new HarpDaalParallelConvert(pointPartitions, pointData, pointPartitions.size(),
pointsPerFile, vectorSize);
converter.HarpToDaal();

HarpDaalParallelConvert converter_cen_to_daal = new HarpDaalParallelConvert(cenTable, daalCenData, numCenPartitions, 0,
vectorSize);
converter_cen_to_daal.HarpCenToDaal();

//------- End of converting data from Harp to DAAL ----------//

//create the DAAL's data structure
HomogenNumericTable ntData = new HomogenNumericTable(daalContext, pointData, vectorSize, totalVectors);
HomogenNumericTable ntCen = new HomogenNumericTable(daalContext, daalCenData, vectorSize, numCentroids);
// compute K-means by DAAL
DistributedStep1Local kmeansLocal = new DistributedStep1Local(daalContext, Double.class, Method.defaultDense, numCentroids);

kmeansLocal.input.set(InputId.data, ntData);
kmeansLocal.input.set(InputId.inputCentroids, ntCen);
PartialResult pres = kmeansLocal.compute();
```

Users first create a class *HarpDaalParallelConvert* to transfer data from Harp side to DAAL side in parallel. They then directly invoke the class *DistributedStep1Local* from DAAL Java API.

We test Harp-DAAL-Kmeans on both of Haswell Xeon E5-2699 v3 and Xeon Phi Knights Landing 7250. The experimentation uses a dataset that includes 20 million training data points with feature dimension of 100, and 100 thousand centroids. Each KNL node has 1 KNL 7250 socket, and each Haswell node has two Haswell E5-2600 v3. We use 60 threads on each KNL node and 30 threads on each Haswell node. Since the thread number does not exceed the number of cores, we are sure that each thread represents a physical core. The core on KNL 7250 has a lower frequency than that of Haswell E5-2699 (1.40Ghz vs 2.3Ghz). Therefore, we can estimate a Haswell node with 30 threads to have a peak performance of 576 Gflops in double precision, and a KNL node with 60 threads to have a peak performance of 2688 Gflops in double precision. Theoretically, each KNL node should deliverer 4.6x speedups than a Haswell node.

In Figures 3 and 4, we compare the execution time of each iteration, from 10 nodes scaling up to 30 nodes. The execution time is decomposed into four parts: 1) Computation time by core, 2) Shared memory access time within a node, 3) Data conversion time between Harp and DAAL interface, and 4) Synchronization overhead among nodes (communication and load balance overhead).

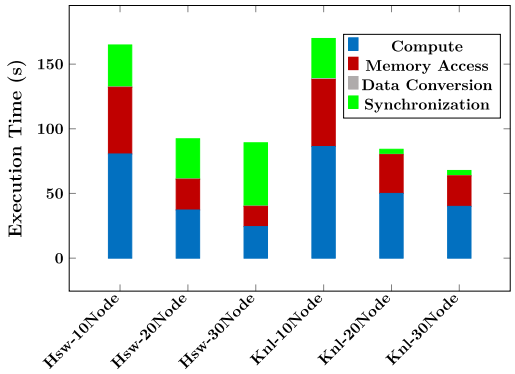

Figure 3. Execution Time of Harp-DAAL-Kmeans

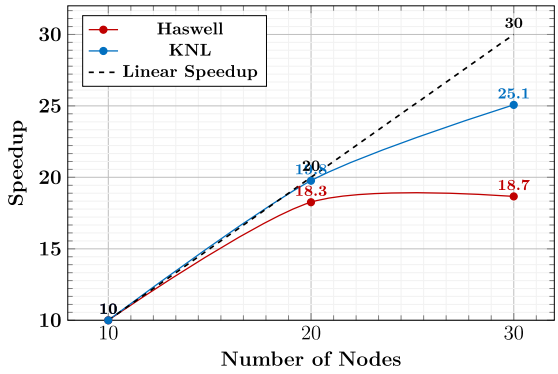

Figure 4. Strong Scalability of Harp-DAAL-Kmeans

On inter-node level, Harp-DAAL-Kmeans has much smaller synchronization overhead at KNL than at Haswell because the Omni-path interconnect at KNL nodes are much faster than the InfiniBand interconnect at Haswell nodes. That is why Harp-DAAL-Kmeans has a better strong scalability at KNL than at Haswell. However, when we consider the intra-node level performance, Harp-DAAL-Kmeans does not gain extra performance by using KNL because it spends more time on computation and memory access compared to a Haswell node. We need to look further the intra-node performance profiling with VTune.

We first investigate the VPU utilization on the two platforms. Figure 5 describes the percentage of the vectorized codes and non-vectorized codes.

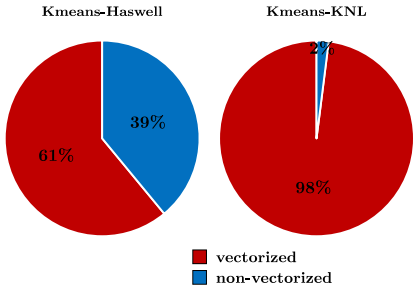

Figure 5. Ratio of vectorized codes for K-means at Haswell and KNL

K-means running on KNL achieves a higher VPU utilization than on Haswell. The reason has two folds. 1) Each KNL core has two VPUs, which doubles the number on Haswell. 2) Each KNL VPU has AVX-512 instruction support, which could deal with 8 double precision operations simultaneously that also doubles that of Haswell VPU (only support AVX2 instruction on 256-bit register). Therefore, K-means on KNL can transfer more instructions into vectorized instructions used by VPUs. Although K-means on KNL has a higher VPU utilization than on Haswell, it still does not gain much performance on KNL than on Haswell. We then investigate the back-end bound metric provided by VTune.

Table 1. The Core and Memory bound at back-end of Processor

Platform	Core Bound	Memory Bound
Haswell	48%	39%
KNL	49.7%	26.1%

Table 1 shows that both of KNL and Haswell have near half of their cycles blocked by CPU cores, however, KNL has a lower core frequency than Haswell and thus has a higher time penalty than Haswell. When it comes to the memory bound, we must decompose the memory bound into different levels of

caches. In Figure 6, we find that Harp-DAAL-Kmeans at KNL has most of the memory access overhead from DRAM. This may be caused by a high L2 cache miss (L2 is the last cache level of KNL). VTune shows that Harp-DAAL-Kmeans has a L2 miss rate of 52%, while at Haswell it also has a similar L3 miss rate around 46%. The additional penalty of last level cache miss for KNL is probably due to its distributed L2 cache architecture, where a local L2 cache miss of a core will first send request to other core's local L2 cache before search the main memory.

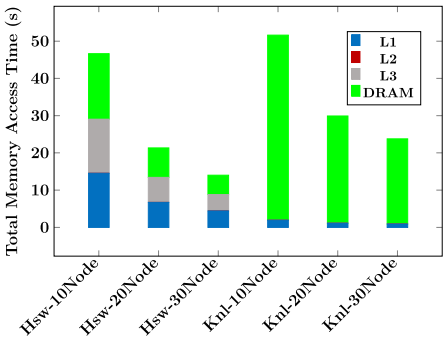

Figure 6. Ratio of memory access overhead for Harp-DAAL-Kmeans

In conclusion, Harp-DAAL-Kmeans have a comparable performance on Haswell and KNL platforms. The KNL version has a higher scalability than the Haswell version, while its computation kernel requires further optimizations in cache usage. We expect that an improved DAAL K-means kernel can exploit the high peak performance of KNL processor.

3.2 Matrix Factorization based on Stochastic Gradient Descent

Matrix Factorization based on Stochastic Gradient Descent (SGD-MF for short) is an algorithm widely used in recommender systems. MF-SGD is one of the implemented algorithm within Harp, however, DAAL current does not have a MF-SGD kernel. We first implement a MF-SGD kernel inside DAAL's framework and then interface it with that of Harp. MF-SGD aims to factorize a sparse matrix into two dense matrices named mode W and model H as follows.

$$V = WH$$

Matrix V includes both training data and test data, a machine learning inspired kernel with use the training data to approximate the model matrices W and H. A standard SGD procedure will update the model W and H when it trains each training data, i.e., an entry in matrix V in the following formula.

$$E_{ij} = V_{ij} - \sum_{k=0}^{r} W_{ik} H_{kj}$$

$$W_{i*}^{t} = W_{i*}^{t-1} - \eta(E_{ij}^{t-1} \cdot H_{*j}^{t-1} - \lambda \cdot W_{i*}^{t-1})$$
$$H_{*j}^{t} = H_{*j}^{t-1} - \eta(E_{ij}^{t-1} \cdot W_{i*}^{t-1} - \lambda \cdot H_{*j}^{t-1})$$

We divide the implementation of MF-SGD into two levels. At the inter-node level, we use a model rotation synchronization pattern shown in Figure 7. The model matrix W is partitioned across all the nodes (mappers), and it becomes local model data of each mapper. In contrast, the model matrix H is sliced and each mapper obtains and updates one slice in a rotated way. As model data has a large size in MF-SGD, the rotation pattern guarantees that each mapper in one iteration will have a total communication data volume of O(Nw), which is irrelevant of the number of nodes N. Theoretically, this rotation pattern will

get a good strong scalability since the communication overhead will not increase when the nodes scale up. At the intra-node level (Figure 8), we choose an asynchronous computation model, where each training point updates its associated row and column simultaneously without lock and waits. By doing so, we can avoid the unbalancing workload caused by the distribution of training points on rows and columns.

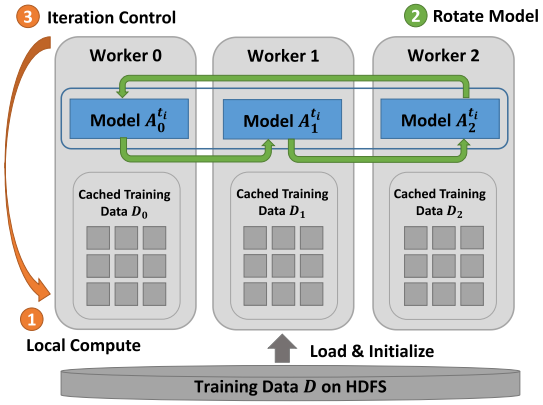

Figure 7. Model data rotated among all the mappers for MF-SGD

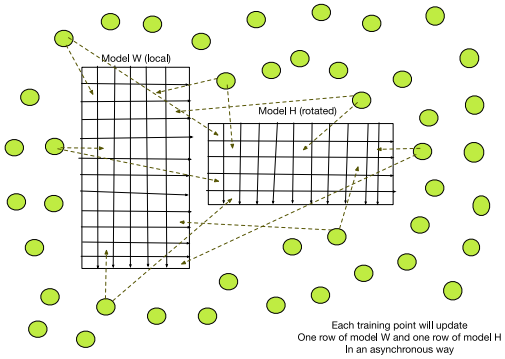

Figure 8. Asynchronous model data updating inside a node

The implementation of Harp-DAAL-SGD has several steps. It first load and copy training and test data from Harp side to DAAL side.

```
//rowMap contains all the row ids on this local worker
Int2ObjectOpenHashMap<int[]> rowMap = new Int2ObjectOpenHashMap<>();

//regrouping the training data points indexed by row ids
//create daal table for training set
//create daal table for row ids
long totalNumTrainV = regroupLoadTrainData(vRowMap, rowMap, colMaps, numThreads);
LOG.info("Total Train Points on this worker: "+totalNumTrainV);
```

It then creates the model data H and the rotator.

```
// ----------------- creating H model within Harp  -----------------
Table<DoubleArray>[] hTableMap = new Table[numModelSlices];
createHModel(hTableMap, colMaps, numModelSlices, oneOverSqrtR,random);

// Use harp's rotator, order of columns is randomized in each rotation
int numSplits = ((int) Math.round(numThreads / 20.0) + 1) * 20;
final int numWorkers = this.getNumWorkers();
int[] order = RotationUtil.getRotationSequences(random, numWorkers, (numIterations + 1) * 2, this);
Rotator<DoubleArray> rotator = new Rotator<>(hTableMap, numSplits, true, this, order, "sgd");
rotator.start();
```

Thirdly, it invokes the DAAL kernel MF-SGD and setup parameters.

```
//create DAAL algorithm object, using distributed version of DAAL-MF-SGD
Distri sgdAlgorithm = new Distri(daal_Context, Double.class,
Method.defaultSGD);

// loading training and test datasets into DAAL
sgdAlgorithm.input.set(InputId.dataWPos, train_wPos_daal);
sgdAlgorithm.input.set(InputId.dataHPos, train_hPos_daal);
sgdAlgorithm.input.set(InputId.dataVal, train_val_daal);

sgdAlgorithm.input.set(InputId.testWPos, test_wPos_daal);
sgdAlgorithm.input.set(InputId.testHPos, test_hPos_daal);
sgdAlgorithm.input.set(InputId.testVal, test_val_daal);

PartialResult model_data = new PartialResult(daal_Context);
sgdAlgorithm.setPartialResult(model_data);
```

Finally, it runs the computation function of DAAL MF-SGD kernel iteratively.

```
//-----load H data from Harp into DAAL's data structure
int hPartitionMapSize = hTableMap[k].getNumPartitions();
LOG.info(" hPartition Size: "+ hPartitionMapSize);
//setup new columns into hTableMap_daal ----------
int table_entry = 0;
for(Partition<DoubleArray> p : hTableMap[k].getPartitions())
{double[] data = (double[])p.get().get();
((SOANumericTable)hTableMap_daal).setArrayOnly(data, table_entry);
table_entry++;
}

model_data.set(PartialResultId.presHMat, hTableMap_daal);

//--------------- start of computation by DAAL ---------------
ComputeStart = System.currentTimeMillis();

sgdAlgorithm.parameter.set(epsilon,lambda, r, wMat_size, hPartitionMapSize, 1, numThreads, 0, 1);
sgdAlgorithm.parameter.setTimer(time);
sgdAlgorithm.compute();
```

In this experiment, we compare the performance of Harp-DAAL-SGD on Haswell nodes and KNL nodes. The dataset shown in Figure 9 and 10 is Hugewiki, which has about1 billion training points and 16 GB model data.

In Figure 9, Harp-DAAL-SGD achieves better execution time performance at KNL than at Haswell. For example, KNL has accelerated Harp-DAAL-SGD by 3.3x in comparison with Haswell when using 30 nodes. Unsurprisingly, Harp-DAAL-SGD at KNL also has a better scalability than the version running on Haswell.

In Figure 10, we see that Harp-DAAL-SGD at KNL even achieves super-liner speedups from 10 nodes to 30 nodes. On one hand, as we discussed that the model rotation synchronization pattern among mappers relieves the communication overhead; On the other hand, KNL's Omni-path interconnect outperforms the InfiniBand interconnect at Haswell nodes just like the case of Harp-DAAL-Kmeans. Besides inter-node synchronization time, Harp-DAAL-SGD also has faster computation and intra-node memory access than at Haswell in Figure 9.

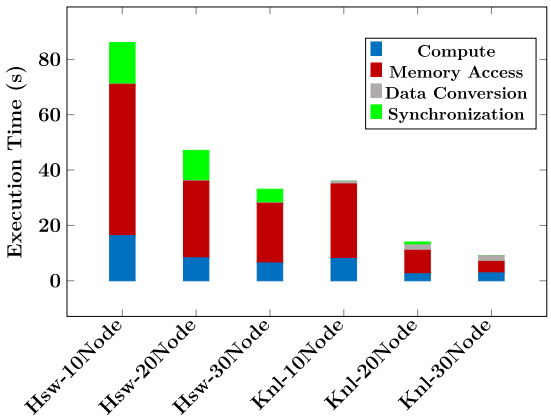

Figure 9. Execution time of Harp-DAAL-SGD on dataset Hugewiki, from 10 nodes to 30 nodes

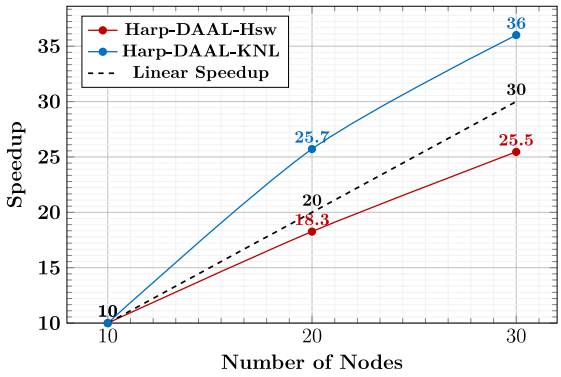

Figure 10. Strong scalability of Harp-DAAL-SGD on Hugewiki from 10 nodes to 30 nodes

We first investigate the VPU utilization of MF-SGD on Haswell and KNL in Figure 11. MF-SGD on KNL has a much higher vectorization than on Haswell, which comes from higher total number of VPUs and the wider AVX registers. However, from the results of K-means, we know that a higher VPU utilization could not guarantee a faster computation time because the KNL core has a lower frequency than Haswell. Therefore, we continue to profile the back-end bound metrics shown in Table 2.

Table 2. The Core and Memory bound at back-end of Processor for MF-SGD

Platform Name	Core Bound	Memory Bound
Haswell	16%	78%
KNL	66%	11%

We observe that Harp-DAAL-SGD running on Haswell is bounded by memory, while it is bounded by core at KNL. This explains why it saves time at KNL because memory access latency usually takes 4 cycles (L1 cache) to 100 cycles (DRAM). A further decomposition of memory access into cache levels is shown in Figure 12.

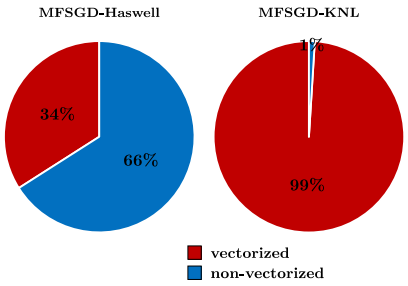

Figure 11. Ratio of vectorized codes for MF-SGD at Haswell and KNL

Harp-DAAL-SGD at KNL has a smaller ratio of DRAM overhead than at Haswell. The last level cache miss rate at KNL is around 55%, which is around 43%. However, KNL has 97% of L1 hit rate while Haswell only has 88% of L1 hit rate. If we suppose the total memory request is N, then KNL generates around 0.0135N DRAM access requests, and Haswell generates around 0.0684N DRAM access requests. Since DRAM has a dominant access latency, Harp-DAAL-SGD on KNL has at least 6 times less memory access overhead than on Haswell.

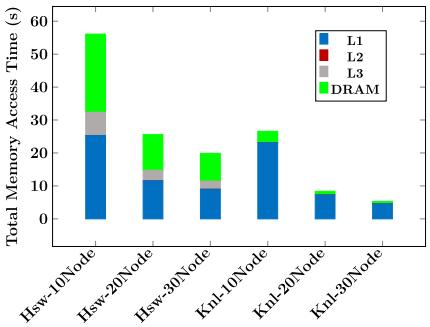

Figure 12. Ratio of memory access overhead for Harp-DAAL-SGD

In conclusion, Harp-DAAL-SGD has shown significant advantages on KNL platform, due to a combination of inter-node communication optimizations (Omni-path and Harp collectives) and intra-node performance tuning.