Java Thread and Process Performance for Parallel Machine Learning on Multicore HPC Clusters

Saliya Ekanayake, Supun Kamburugamuve, Pulasthi Wickramasinghe, Geoffrey C. Fox
School of Informatics and Computing
Indiana University, Bloomington
sekanaya@indiana.edu, skamburu@indiana.edu, pswickra@indiana.edu, gcf@indiana.edu

***Abstract*—The growing use of Big Data frameworks on large machines highlights the importance of performance issues and the value of High Performance Computing (HPC) technology. This paper looks carefully at three major frameworks Spark, Flink and Message Passing Interface (MPI) both in scaling across nodes and internally over the many cores inside modern nodes. We focus on the special challenges of the Java Virtual Machine (JVM) using an Intel Haswell HPC cluster with 24 cores per node. Two parallel machine learning algorithms, K-Means clustering and Multidimensional Scaling (MDS) are used in our performance studies. We identify three major issues – thread models, affinity patterns, and communication mechanisms – as factors affecting performance by large factors and show how to optimize them so that Java can match the performance of traditional HPC languages like C. Further we suggest approaches that preserve the user interface and elegant dataflow approach of Flink and Spark but modify the runtime so that these Big Data frameworks can achieve excellent performance and realize the goals of HPC-Big Data convergence.**

***Index Terms*—Big Data; Machine Learning; Java; Multicore; HPC;**

I. Introduction

Parallel machine learning is a blooming area in Big Data with a high demand for performance. A primary challenge with parallel machine learning is its sensitivity to performance variations in individual tasks. To elaborate, these algorithms are typically iterative in nature and require collective communications that are not easily overlapped with computations; hence the performance is susceptible to communication overheads and noise caused by slow performing tasks. Beyond the nature of these applications, The Java runtime on multicore Non-Uniform Memory Access (NUMA) nodes brings out additional challenges in keeping constant performance when scaling over the many cores within a node as well as across nodes. In this paper, we focus on such special challenges of the Java Virtual Machine (JVM) for parallel machine learning. In particular, we identify three major factors – thread models, affinity patterns, and communication mechanisms – that affect performance by large factors and show optimization techniques to bring Java performance closer to traditional High Performance Computing (HPC) applications in languages like C.

In studying performance, we carefully look at three major frameworks – Message Passing Interface (MPI), Spark, and Flink. Two parallel machine learning algorithms – K-Means clustering and Multidimensional Scaling (MDS) – are used to evaluate these frameworks using an Intel Haswell HPC cluster consisting of 24-core nodes. Most of the parallel machine learning algorithms employ global collective communications, hence the choice of MDS and K-Means to cover them broadly. Based on the results, we further suggest approaches to improve the runtime of Flink and Spark, while preserving their elegant dataflow programming model.

The remaining sections are organized as follows. Section II presents a comparison of execution models of MPI, Spark, and Flink, which paves the way to explain performance differences observed in later sections. Section III elaborates the three major factors affecting performance of Big Data applications. It describes two thread models, six affinity patterns, and two communication mechanisms used to evaluate performance. Section IV outlines the two machine learning applications and their various implementations in MPI, Spark, and Flink. Section V describes the testing of these applications using an Intel Haswell HPC cluster followed by a discussion on Big Data frameworks and MPI in Section VI. Section VII and Section VIII present our conclusion based on the experiments and future plans to improve Spark and Flink.

II. Comparison of Execution Models

The MPI and Big Data platform implementations that we study, follow two different execution models, message passing and dataflow [1]. The key differences are with the task distribution and communication. MPI is a rich execution model that can support different styles of programming including Bulk Synchronous Parallel (BSP) and many-task models. On the other hand, Big Data platforms primarily follow the data oriented execution model that is termed the dataflow model [1]. Flink [2] is a direct realization of the dataflow execution model, where as Spark resembles the dataflow model but executes the parallel program as a series of transformations over its distributed data model – Resilient Distributed Dataset (RDD) [3].

In the dataflow model, the parallel program is expressed as a Directed Acyclic Graph (DAG). Parallel tasks are assigned to nodes of the DAG and the flow of data between nodes completes the “wiring”. In contrast, classic parallel applications employ the message passing model, where long running tasks are orchestrated using point-to-point and collective communication calls. We sometimes term this an “in-place” execution model to distinguish it from dataflow. The dataflow

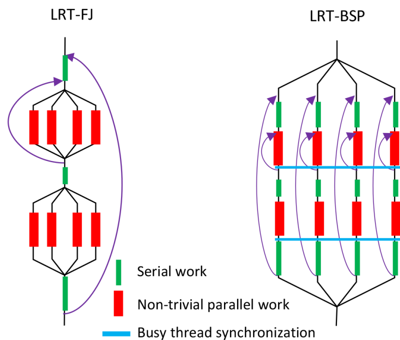

Fig. 1: Long Running Threads Fork-Join (LRT-FJ) vs. Long Running Threads Bulk Synchronous Parallel (LRT-BSP)

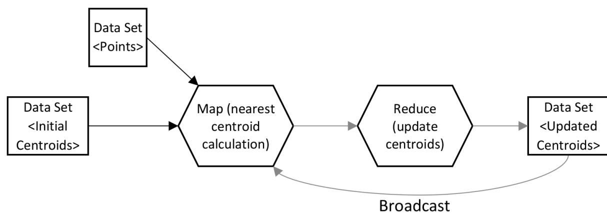

Fig. 2: Flink and Spark K-Means algorithm. Both Flink and Spark implementations follow the same data-flow

model permits both batch and stream processing [4] of data, which are supported in Flink and Spark. Apache Beam [5] is a unified dataflow programming Application Programming Interface (API), which can be used to write both streaming and batch data processing applications compatible to run on either Spark or Flink.

With MPI machine learning applications, all parts of the program are executed on a set of pre-allocated tasks that define the parallelism of the execution. The same (reflecting in-place model) tasks are responsible for the computing and communications of the program. On the other hand, dataflow implementations allocate separate tasks for different stages of the application and connect them through communication channels. These tasks and communication links form the execution graph. The MPI programming model permits complete control over the execution in a single task including memory management and thread execution. The dataflow execution model hides these details from the user and provides only a high level API.

With current implementations of Big Data frameworks, programming models and execution models are coupled together even though they could be independent of each other. For example the dataflow programming models in Spark and Flink are implemented as dataflow execution graphs compared to an in-place execution as in MPI applications.

It is important to note the differences in how the iterations are handled in pure dataflow applications. With dataflow applications, iterations are handled as unrolled for loops. Even though the user specifies a `for` loop execution, it translates to a lengthy dataflow graph. This implies that the data from one loop's tasks that is relevant to the next needs to be sent through communications. The MPI model doesn't have this requirement because a for loop is a regular in memory loop and data from the iteration is available to the next via the task's memory.

III. Performance Factors

A. Thread Models

Threads offer a convenient construct to implement shared memory parallelism. A common pattern used in both Big Data and HPC is the Fork-Join (FJ) thread model. In this approach, a master thread spawns parallel regions dynamically as required. FJ regions are implicitly synchronized at the end, after which the worker threads are terminated and only the master thread will continue until a new parallel region is created. Thread creation and termination are expensive; therefore, FJ implementations employ thread pools to hand over forked tasks. Pooled threads are long-lived yet short-activated; they release CPU resources and switch to idle state after executing their tasks. This model is subsequently referred to as LRT-FJ in this paper. Java has built-in support for LRT-FJ through its `java.util.concurrent.ForkJoinPool` [1]. Habanero Java [6], an OpenMP [7]-like implementation in Java, also supports LRT-FJ via its `forall` and `forallChunked` constructs.

We experimented with another approach to shared memory parallelism, hereafter referred to as LRT-BSP. It resembles the classic BSP style but with threads. Fig. 1 depicts a side-by-side view of LRT-FJ and LRT-BSP models. The notable difference is that in LRT-BSP, threads are busy from start to finish of the program, not just within the parallel region as in LRT-FJ. The next important difference is the use of explicit synchronization constructs (blue horizontal lines) after non-trivial parallel work (red bars in the figure) in LRT-BSP. There are constructs such as `CyclicBarrier` in Java to aid the implementation of these synchronization steps. However, we employed native compare-and-swap (CAS) operations and busy loops for performance as well as to keep threads "hot" on cores. A third difference in LRT-BSP is that the serial part of the code (green bars) is replicated across workers, where as in LRT-FJ it is executed by just the master thread. Performance results show that despite the replication of serial work in LRT-BSP, it does not add significant overhead. The reason for this behavior is that in a well-designed parallel application, the serial portions are trivial compared to the parallel work loads and the total amount of memory accesses in LRT-BSP is equal to that of LRT-FJ for these parts.

Beyond the differences in the execution model, we observed a significant performance improvement with LRT-BSP compared to LRT-FJ for parallel Java applications. Analyzing `perf` statistics revealed that LRT-FJ experiences a higher number of context switches, CPU migrations, and data Translation Lookaside Buffer (dTLB) load/store misses than

[1] https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html

TABLE I: Affinity patterns

		Process Affinity		
		Cores	Socket	None (All)
Thread Affinity	Inherit	CI	SI	NI
	Explicit per Core	CE	SE	NE

LRT-BSP. In an MDS run, the factors were over 15x and 70x for context switches and CPU migrations respectively. These inefficiencies coupled with the overhead of scheduling threads lead to noise in computation times within parallel FJ regions. Consequently, synchronization points become significantly expensive, and performance measurements indicate performance degredation with increasing number of threads in LRT-FJ.

B. Thread and Process Affinity Patterns

Modern multicore HPC cluster nodes typically contain more than one physical CPU. Although memory is shared between these central processing units (CPUs), memory access is not uniform. CPUs with their local memory compose NUMA domains or NUMA nodes. Developing parallel applications in these settings requires paying attention to the locality of memory access to improve performance.

In supported Operating Systems (OSs), process affinity determines where the OS can schedule a given process as well as the part of memory it can access. Threads spawned within a process by default inherit the affinity policy of the process. Also, it is possible to set affinity explicitly to threads as desired for performance reasons. This research explores six affinity patterns and identifies binding options that produce the best and worst performance.

Details of the three process affinity patterns in Table I are:
Core - binds the process to N cores, where N is the number of threads used for shared memory parallelism.
Socket - binds the process to a physical CPU or socket.
None (All) - binds the process to all available cores, which is equivalent to being unbound.

Worker threads may either inherit the process binding or be pinned to a separate core. K-Means and MDS performance tests revealed that selecting proper affinity settings out of these patterns can substantially improve overall performance.

C. Communication Mechanisms

Processes within a node offer an alternative approach from threads to exploiting intra-node parallelism. Long running processes as in MPI programs avoid frequent scheduling overheads and other pitfalls discussed with short-activated threads. However, the shared nothing nature of processes imposes a higher communication burden than with threads, especially when making collective calls. Increasing process count to utilize all cores on modern chips with higher core counts makes this effect even worse, degrading any computational advantages of using processes.

A solution typically employed in HPC is to use the node shared memory to communicate between processes running in the same node. In [8] we have shown significant performance improvement in Java inter-process communication by implementing a memory maps-based communication layer. We have later applied the same technique in [9] to improve communication between the Big Data Apache Storm tasks.

IV. APPLICATIONS

To evaluate the performance of different aspects discussed in Section III, we have implemented six variants of K-Means clustering. Four of them are OpenMPI-based in both Java and C supporting LRT-FJ and LRT-BSP thread models. The remainder are based on Flink and Spark. We have also implemented two flavors of Deterministic Annealing Multidimensional Scaling (DA-MDS) [10] with optimizations discussed in [8] to support the two thread models in Java and OpenMPI. The following subsections describe the details of these applications.

A. MPI Java and C K-Means

The two C implementations use OpenMPI for message passing and OpenMP for shared memory parallelism. The LRT-FJ follows the conventional MPI plus `#pragma omp parallel` regions. LRT-BSP, on the other hand, starts an OpenMP parallel region after `MPI_INIT` and continues to follow the models illustrated in Fig. 1. Intermediate thread synchronization is done through atomic built-ins of GNU Compiler Collection (GCC).

The Java implementations use OpenMPI's Java binding [11], [12] and Habanero-Java [6] thread library, which provides similar parallel constructs to OpenMP. In LRT-BSP, intermediate thread synchronization uses Java atomic support, which is more efficient than other lock mechanisms in Java.

B. Flink K-Means

Flink provides a dataflow-based programming and execution model. The dataflow computations composed by the user are converted to an execution dataflow graph by Flink and executed on a distributed set of nodes.

Flink K-Means' dataflow graph is shown in Fig. 2. Inputs to the algorithm are a set of points and a set of centroids read from the disk. At each iteration, a new set of centroids are calculated and fed back to the beginning of the iteration. The algorithm partitions the points into multiple map tasks and uses the full set of centroids in each map task. Each map task assigns its points to their nearest centroid. The average of such points is reduced (sum) for each centroid to get the new set of centroids, which are broadcast to the next iteration. This is essentially the same algorithm as that used in MPI but expressed as a stream of dataflow transformations. In particular, the Flink reduction and broadcast are equivalent to `MPI_Allreduce` semantics.

C. Spark K-Means

Spark is a distributed in-memory data processing engine. The data model in Spark is based around RDDs [3]. The execution model of Spark is based on RDDs and lineage graphs. The lineage graph captures dependencies between RDDs and their transformations. The logical execution model

is expressed through a chain of transformations on RDDs by the user.

We used a slightly modified version [2] of the K-Means implementation provided in Spark MLlib [13] library . The overall dataflow is shown in Fig. 2, which is as same as that of Flink K-Means. Also, the inputs are read in a similar fashion from disk. The points data file is partitioned and parallel map operations are performed on each partition. Each point in a data partition is cached to increase performance. Within the map operations, points are assigned to their closest centers. The reduce step gathers all this information to the driver program, where the new set of centers are calculated and boradcast to all the worker nodes for the next iteration.

D. MPI Java MDS

MDS is a technique to visualize high dimensional data in a lower dimension, typically in 3D. We extended our DA-MDS [8] implementation, which is based on LRT-FJ, to include a version of LRT-BSP as well. Also, both these versions include shared memory based inter-process communication support. Note, computations in DA-MDS grow $O(N^2)$ and communications $O(N)$. Moreover, unlike K-Means, where only one parallel region is required, DA-MDS requires multiple parallel regions revisited on each iteration until converged. This hierarchical iteration pattern (parallel conjugate gradient iteration inside a classic expectation maximization loop) causes issues with the Big Data frameworks that we will explore elsewhere.

V. Evaluation

The experiments were run on Juliet, which is an Intel Haswell HPC cluster with 128 nodes total. We tested on 96 nodes that have 24 cores over 2 sockets. The other 32 nodes have 36 cores per node. Each node consists of 128GB of main memory and 56Gbps Infiniband (IB) interconnect and 1Gbps dedicated Ethernet connections. MPI runs used the IB except when comparing against Flink and Spark, where all three frameworks used Transmission Control Protocol (TCP) communications. TCP over IB was not available in this cluster.

A. MPI Java and C K-Means

Fig. 3 and Fig. 4 show K-Means Java and C total runtime for 1 million 2D points and 1000 centroids respectively. Each figure presents performance of both LRT-FJ and LRT-BSP models over the six binding patterns identified in Table I. These were run on 24-core nodes; hence the abscissa shows all the eight possible combinations of threads and processes within a node to exploit the full 24-way parallelism. The left most pattern, 1x24, indicates all processes and the right most pattern, 24x1 indicates all threads within a node. Note, patterns 8x3 and 24x1 imply that processes span across NUMA memory boundaries, which is known to be inefficient but presented here for completeness. The red and orange lines represent inherited thread affinity for LRT-FJ and LRT-BSP respectively. Similarly, the black and green lines illustrate explicit thread pinning, each to a core, for these two thread models.

Java results suggest LRT-FJ is the worst whatever the affinity strategy for any pattern other than 1x24, which is all MPI and does not use thread parallel regions. A primary reason for this poor performance is the thread scheduling overhead in Java as FJ threads are short-activated. Also, the JVM spawns extra bookkeeping threads for Garbage Collection (GC) and other tasks, which compete for CPU resources as well. Of the LRT-BSP lines, the unbound threads (`NI`) show the worst performance. Affinity patterns `NE` and `CE` seem to give the best runtime with increasing number of threads.

C results show the same behavior for unbounded and explicitly bound threads. The two thread models, however, show similar performance, unlike Java. Further investigation of this behavior revealed OpenMP threads keep the CPUs occupied at 100% between FJ regions suggesting OpenMP internally optimizes threads similar to the Java LRT-BSP implementation introduced in this paper. This could be adopted in Java to give efficient FJ implementations employing LRT-BSP model.

Fig. 5 illustrates the effect of affinity patterns `CE` and `NE` for varying data sizes on LRT-BSP. They performed similar to each other, but numbers favor pattern `CE` over `NE`.

Fig. 6 compares Java and C LRT-BSP runtimes for K-Means over varying data sizes across thread and process combinations. Results demonstrate Java performance is on par with C.

Fig. 7 and presents LRT-FJ and LRT-BSP performance over varying data sizes for affinity pattern `CE`. In this figure, the number of centroids were incremented as 1k,10k, and 100k. LRT-BSP shows constant performance across thread and process combinations for all data sizes, where as LRT-FJ exhibits abysmal performance with increasing threads and data sizes.

B. MPI Java MDS

Fig. 8 through Fig. 10 illustrate DA-MDS performance for data sizes 50k, 100k, and 200k on 24-core and 36-core nodes. Each figure presents DA-MDS runtime for the two thread models and affinity patterns `CE`, `SE`, `NE`, and `NI`. Patterns `CI` and `SI` were omitted as they showed similar abysmal performance as `NI` in earlier K-Means results. Thread and process combinations for 24-core nodes are as same as the ones used in K-Means experiments. On 36-core nodes, nine patterns were tested from 1x36 to 36x1. However, as LRT-FJ allocates data for all threads at process level, 200k decomposition over 16 nodes produced more data than what Java 1D arrays could hold. Therefore, this pattern could not be tested for 200k data. Note, thread local data allocation could solve this but it would require a major rewrite of the code to support 2D arrays over 1D. LRT-BSP did not face this situation as data structures are local to threads and each allocates only data required for the thread, which is within Java's array limit of $2^{31} - 1$ elements.

The above results confirm that Java LRT-FJ has the lowest performance irrespective of the binding, data size or the num-

[2] https://github.com/DSC-SPIDAL/spark/tree/1.6.1.modifiedKmeans

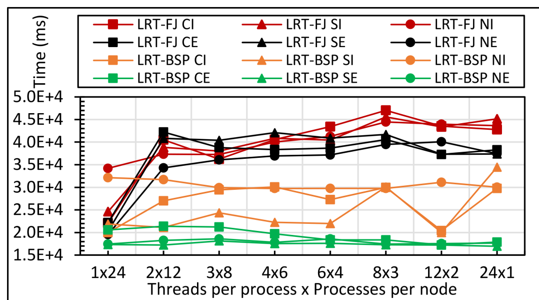

Fig. 3: Java K-Means 1 mil points and 1k centers performance on 16 nodes for LRT-FJ and LRT-BSP with varying affinity patterns over varying threads and processes.

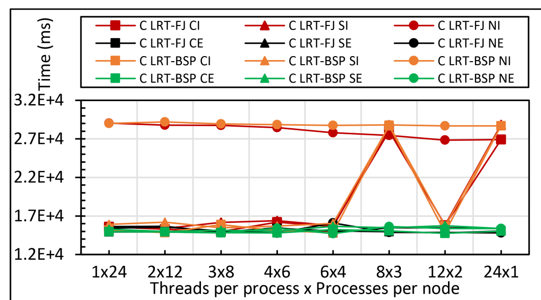

Fig. 4: C K-Means 1 mil points and 1k centers performance on 16 nodes for LRT-FJ and LRT-BSP with varying affinity patterns over varying threads and processes.

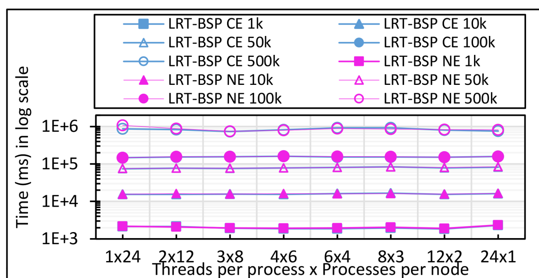

Fig. 5: Java K-Means LRT-BSP affinity CE vs NE performance for 1 mil points with 1k,10k,50k,100k, and 500k centers on 16 nodes over varying threads and processes.

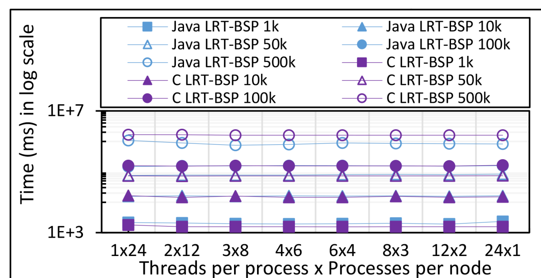

Fig. 6: Java vs C K-Means LRT-BSP affinity CE performance for 1 mil points with 1k,10k,50k,100k, and 500k centers on 16 nodes over varying threads and processes.

TABLE II: Linux `perf` statistics for DA-MDS run of 18x2 on 32 nodes. Affinity pattern is `CE`.

	LRT-FJ	**LRT-BSP**
Context Switches	477913	31433
CPU Migrations	63953	864
dTLB load misses	17226323	6493703

TABLE III: Java DA-MDS speedup for varying data sizes on 24-core nodes. Red values indicate the suboptimal performance of LRT-FJ model compared to LRT-BSP. Ideally, these values should be similar to their immediate left cell values.

	Data Size								
	50k			**100k**			**200k**		
24-Core Nodes	**1x24 LRT-BSP**	**12x2 LRT-BSP**	**12x2 LRT-FJ**	**1x24 LRT-BSP**	**12x2 LRT-BSP**	**12x2 LRT-FJ**	**1x24 LRT-BSP**	**12x2 LRT-BSP**	**12x2 LRT-FJ**
16	1	1	0.6	1	1	0.6	1	1	0.4
32	2.2	2	1.1	1.9	1.9	1.1	1.9	2	0.6
64	3.9	3.6	1.9	3.6	3.6	1.9	3.7	3.8	0.9

ber of threads. On the other hand, the LRT-BSP model produced constant high performance across all these parameters. Investigating these effects further, an 18x2 run for 100k data produced the `perf` stats in Table II, which show a vast number of context switches, CPU migrations, and data Translation Lookaside Buffer load misses for LRT-FJ compared to LRT-FJ. These statistics are directly related with performance and hence explain the poor performance of LRT-FJ model.

Table III presents scaling of DA-MDS across nodes for data sizes 50k, 100k, and 200k. Speedup values are measured against the all process – 1x24 or 1x36 – base case. With doubling of the nodes, the performance is expected to double. However, none of the 12x2 LRT-FJ values came close to the expected number; hence shown in red. In contrast, 12x2 of LRT-BSP follows the expected doubling in performance and also can produce slightly better results than 1x24 with increasing data.

C. Flink and Spark K-Means

We evaluated the performance of K-Means algorithm implemented in Flink and Spark to compare these frameworks against MPI. The evaluation was done in 16 nodes, each with 24 cores. We measured the difference between total time and computation time to estimate overheads including communication. Note, in both Spark and Flink, communications are handled internally to the framework and it is not possible to measure this through the available API functions. The results are shown in Fig. 11 for 1 million 2D data points with varying number of centroids. We observed significant communication overhead in these frameworks compared to MPI. The primary reason for such poor performance is the sub-optimal implementation of reductions in Flink and Spark.

Fig. 13 illustrates the dataflow reduction model implemented in Spark and Flink, where all parallel tasks send data to a single or multiple reduce tasks to perform the reduction.

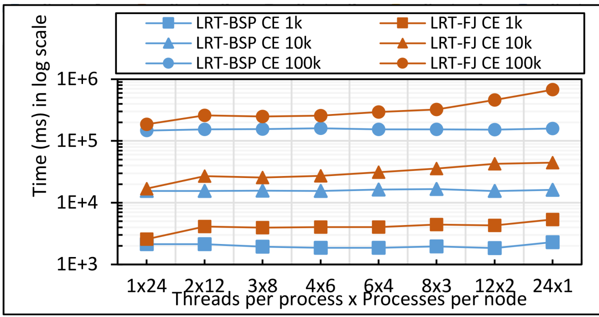

Fig. 7: Java K-Means 1 mil points with 1k,10k, and 100k centers performance on 16 nodes for LRT-FJ and LRT-BSP over varying threads and processes. The affinity pattern is CE.

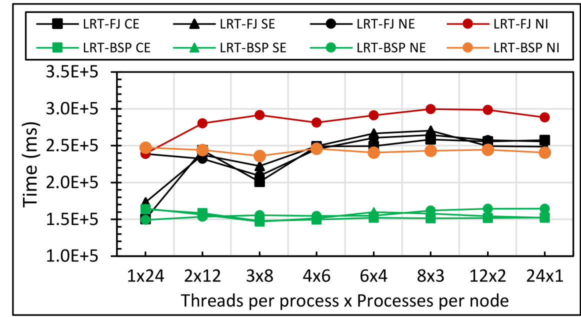

Fig. 8: Java DA-MDS 50k points performance on 16 nodes for LRT-FJ and LRT-BSP over varying threads and processes. Affinity patterns are CE,NE,SE, and NI.

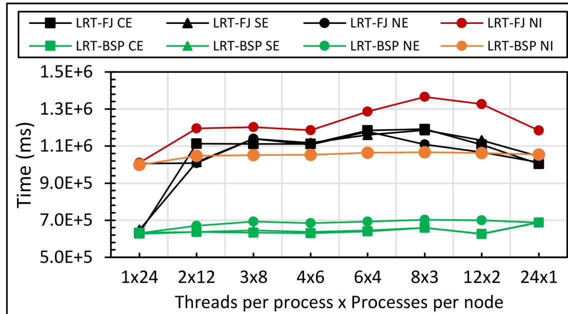

Fig. 9: Java DA-MDS 100k points performance on 16 nodes for LRT-FJ and LRT-BSP over varying threads and processes. Affinity patterns are CE,NE,SE, and NI.

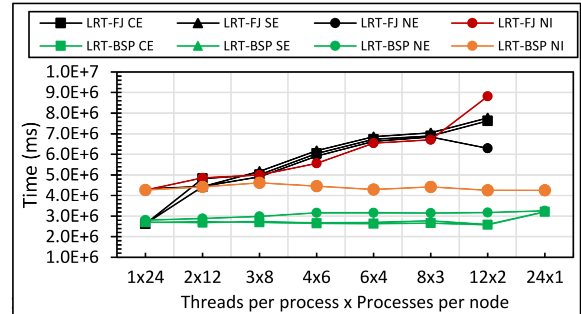

Fig. 10: Java DA-MDS 200k points performance on 16 nodes for LRT-FJ and LRT-BSP over varying threads and processes. Affinity patterns are T,S,V, and U.

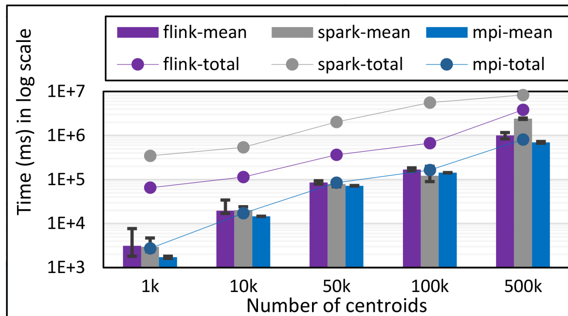

Fig. 11: K-Means total and compute times for 1 million 2D points and 1k,10,50k,100k, and 500k centroids for Spark, Flink, and MPI Java LRT-BSP `CE`. Run on 16 nodes as 24x1.

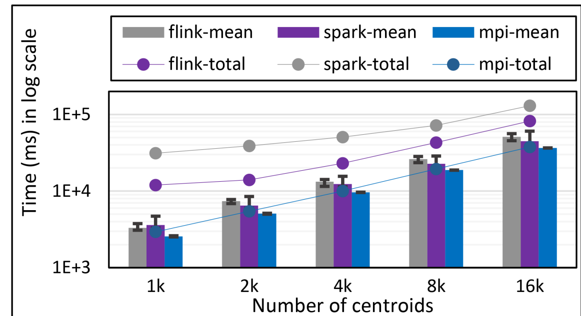

Fig. 12: K-Means total and compute times for 100k 2D points and 1k,2k,4k,8k, and 16k centroids for Spark, Flink, and MPI Java LRT-BSP `CE`. Run on 1 node as 24x1

K-Means requires an MPI like `Allreduce` semantics; hence the reduction in these programs is followed by a broadcast. Similar to the reduction operation, the broadcast is implemented serially as well. As the number of parallel tasks and the message size increase, this two-step approach becomes highly inefficient in performing global reductions. On the other hand, MPI uses a recursive doubling algorithm for doing the reduction and broadcast together, which is very efficient and happens in-place.

Since the communication overhead was dominant in K-Means algorithm, we performed a single node experiment with one process and multiple threads to look at computation costs more closely. With one process there is no network communication in Flink or Spark and Fig. 12 illustrates the results. Flink uses an actor-based execution model using Akka framework to execute the tasks. The framework creates and destroys LRT-FJ style threads to execute the individual tasks. Spark uses an executor/task model where an executor creates at most a single task for each core that is allocated to the executor. With this experiment, we have observed execution time imbalances among the parallel tasks for both Spark and Flink. The same has been observed with the LRT-FJ Java MPI implementation of K-Means and we could minimize these effects in MPI Java with the LRT-BSP style executions. Balanced parallel computations are vital to efficient parallel algorithms as the slowest task dominates the parallel computation time.

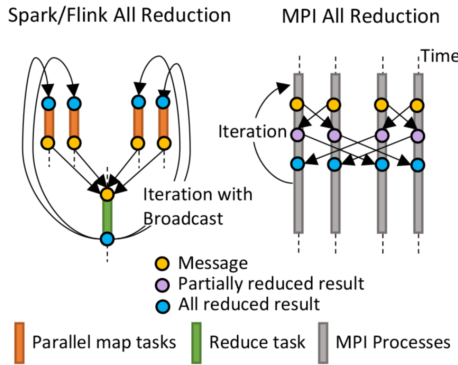

Fig. 13: Spark and Flink's all reduction vs MPI all reduction.

VI. Discussion

Spark and Flink both are in-memory computation platforms, unlike Hadoop, which is primarily a disk-based platform. These systems are designed to handle large amounts of data and be fault tolerant in case of failures. They can use disks as an auxiliary storage if the data is too large to fit in the memory. On the other hand, MPI is a lightweight framework with excellent communication and execution semantics that are well suited for HPC. We believe Java MPI implementations with careful design of threading, computations and communications as discussed in this work, provide top-notch performance for Java-based machine learning applications that match C implementations for big data platforms. This study shows the many factors that are critical for achieving the best possible performance and scalability and how they can be carefully tuned.

Current implementations of Big Data computation frameworks lack efficient communication algorithms as implemented in MPI. We have identified inefficient communication as the most detrimental feature in getting to the best possible performance with Spark and Flink. For example, a carefully tuned broadcast operation can work in $O(log\ n)$ steps while a sequential implementation needs $O(n)$ where n is the number of parallel communicators. As the parallelism increases the communication overhead in terms of both latency and bandwidth dramatically increases for the sequential approach compared to the optimized approach.

The computation time variations in the parallel tasks of Flink and Spark frameworks can be attributed to the LRT-FJ style invocations and GCs. It is hard to completely avoid GC overheads but Flink-like systems have adopted off-heap memory management for reducing this effect. The LRT-BSP style threads can also help in reducing the compute time variations, as evident in MPI Java applications. Another factor that can affect computation is the inefficient use of memory hierarchy. If cache optimizations are not considered, performance can show degrade drastically. Also, for larger data sets, it can be efficient to run multiple processes rather than a single process with threads due to large page tables required for the single process.

VII. Related Work

A plethora of libraries are available for Java in HPC environments including many MPI implementations; Guillermo et al. [14] discuss the performance of some of these frameworks in HPC environments. MPJ-Express [15] and JaMP [16] are popular pure Java implementation of the MPI standard. In our work, we used OpenMPI's Java bindings to develop the MPI applications. Our preliminary studies showed OpenMPI performed the best among the available Java MPI implementations. Rajesh et al. [17] discusses actor-based frameworks to exploit the multicore machines and Flink uses actor model for handling concurrency. Java Garbage Collection (GC) plays a vital role in HPC Java applications because of the slowest parallel tasks dominating the performance. Maria et al. [18] shows how to optimize the Java GC in multicore NUMA machines. Research on improving Spark performance by introducing its own memory management and cache handing system is being done in Project Tungsten [19], which aims to greatly reduce the usage of java objects and to reduce Spark's memory footprint.

Reaching high performance on multicore clusters using hybrid execution model of threads and processes is presented in [20], [21], [22]. They discuss the performance across NUMA sockets, as well as how threads and processes perform in conjunction. In this work, we apply these techniques in the context of machine learning algorithms to get scalable performance.

Because of the high number of cores available in multicore nodes, hybrid communication models involving shared memory communication and network communication are preferred. In these models, the tasks within a node first communicate using shared memory and then the results are forwarded to other nodes in the cluster. Previous work by the authors [8] focused on improving the collective communications for Java machine learning algorithms using this hybrid approach.

Hadoop became the first widely used system for big data processing and it uses a disk based communication among tasks with HDFS. Hadoop offers only the Map and Reduce dataflow operations. The later systems such as Twister [23], Spark, Flink and Google Cloud Dataflow [1] are using in-memory and network communications among the tasks and are offering a rich set of data-flow operations compared to Hadoop. Bacause of the way communication is handled Spark and Flink, they are much closer to MPI in run-time and can use the advanced communication features in MPI.

These big data frameworks follow the dataflow model and the equivalent of collective communications in MPI are implemented as dataflow operators. These implementations are elegant but inefficient compared to the optimized collective communication algorithms implemented in MPI [24], [25]. Recent work by the authors [9] have improved communications of Apache Storm streaming framework with classical collective algorithms found in MPI implementations. Harp [26] is a collective communication framework developed for Hadoop to speed up the machine learning applications. There has

being efforts to bring HPC enhancements such as RDMA [27] ASDAto big data frameworks and these have given excellent performance in HPC environments.

Our findings are in the spirit of HPC-ABDS (the High Performance Computing enhanced Apache Big Data Stack) [28] and help establish a Big Data - HPC convergence approach [29].

VIII. Conclusion and Future Work

In this paper we discussed how to obtain consistent scalable performance of machine learning algorithms implemented in Java in large multicore clusters. The deficiencies in performance we observed before the improvements in Java MPI machine learning applications can be observed on the current implementations of Big Data run-times such as Flink and Spark, and we are working on bringing these improvements to such frameworks. In particular we aim to improve the collective communications of Flink and Spark using efficient algorithms. As part of the SPIDAL (Scalable parallel interoperable data analytics library) [8] machine learning library we would like to apply these techniques to further algorithms.

Acknowledgment

This work was partially supported by NSF CIF21 DIBBS 1443054 and NSF RaPyDLI 1415459. We thank Intel for their support of the Juliet system. We extend our gratitude to the FutureSystems team and the Habanero Java team at Rice University.

References

[1] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt *et al.*, "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1792–1803, 2015.

[2] "Apache Flink: Scalable Batch and Stream Data Processing." [Online]. Available: https://flink.apache.org/

[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[4] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Presented as part of the*, 2012.

[5] "Apache Beam." [Online]. Available: http://beam.incubator.apache.org/

[6] S. Imam and V. Sarkar, "Habanero-java library: A java 8 framework for multicore programming," in *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, ser. PPPJ '14. New York, NY, USA: ACM, 2014, pp. 75–86. [Online]. Available: http://doi.acm.org/10.1145/2647508.2647514

[7] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998. [Online]. Available: http://dx.doi.org/10.1109/99.660313

[8] S. Ekanayake, S. Kamburugamuve, and G. Fox, "Spidal: High performance data analytics with java and mpi on large multicore hpc clusters," in *Proceedings of the 2016 Spring Simulation Multi-Conference (SPRINGSIM)*, Pasadena, CA, USA, 3–6, 2016.

[9] S. Kamburugamuve, S. Ekanayake, M. Pathirage, and G. Fox, "Towards High Performance Processing of Streaming Data in Large Data Centers," in *HPBDC 2016 IEEE International Workshop on High-Performance Big Data Computing in conjunction with The 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2016), Chicago, Illinois USA*, 2016.

[10] Y. Ruan and G. Fox, "A robust and scalable solution for interpolative multidimensional scaling with weighting," in *Proceedings of the 2013 IEEE 9th International Conference on e-Science*, ser. ESCIENCE '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 61–69. [Online]. Available: http://dx.doi.org/10.1109/eScience.2013.30

[11] O. Vega-Gisbert, J. E. Roman, S. Groß, and J. M. Squyres, "Towards the availability of java bindings in open mpi," in *Proceedings of the 20th European MPI Users' Group Meeting*, ser. EuroMPI '13. New York, NY, USA: ACM, 2013, pp. 141–142. [Online]. Available: http://doi.acm.org/10.1145/2488551.2488599

[12] J. M. S. Oscar Vega-Gisbert, Jose E. Roman, "Design and implementation of java bindings in open mpi," 2014. [Online]. Available: users.dsic.upv.es/ jroman/preprints/ompi-java.pdf

[13] X. Meng, J. Bradley, B. Yuvaz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *JMLR*, vol. 17, no. 34, pp. 1–7, 2016.

[14] G. L. Taboada, S. Ramos, R. R. Expósito, J. Touriño, and R. Doallo, "Java in the high performance computing arena: Research, practice and experience," *Science of Computer Programming*, vol. 78, no. 5, pp. 425–444, 2013.

[15] M. Baker, B. Carpenter, and A. Shafi, "MPJ Express: towards thread safe Java HPC," in *2006 IEEE International Conference on Cluster Computing*. IEEE, 2006, pp. 1–10.

[16] M. Klemm, M. Bezold, R. Veldema, and M. Philippsen, "JaMP: an implementation of OpenMP for a Java DSM," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 18, pp. 2333–2352, 2007.

[17] R. K. Karmani, A. Shali, and G. Agha, "Actor frameworks for the JVM platform: a comparative analysis," in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*. ACM, 2009, pp. 11–20.

[18] M. Carpen-Amarie, P. Marlier, P. Felber, and G. Thomas, "A performance study of java garbage collectors on multicore architectures," in *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM, 2015, pp. 20–29.

[19] (2015) Project Tungsten: Bringing Apache Spark Closer to Bare Metal. [Online]. Available: https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html

[20] M. J. Chorley and D. W. Walker, "Performance analysis of a hybrid MPI/OpenMP application on multi-core clusters," *Journal of Computational Science*, vol. 1, no. 3, pp. 168–174, 2010.

[21] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes," in *2009 17th Euromicro international conference on parallel, distributed and network-based processing*. IEEE, 2009, pp. 427–436.

[22] D. Camp, C. Garth, H. Childs, D. Pugmire, and K. Joy, "Streamline integration using MPI-hybrid parallelism on a large multicore architecture," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 11, pp. 1702–1713, 2011.

[23] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A Runtime for Iterative MapReduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 810–818. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851593

[24] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of MPI collective operations," *Cluster Computing*, vol. 10, no. 2, pp. 127–143, 2007.

[25] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[26] B. Zhang, Y. Ruan, and J. Qiu, "Harp: Collective communication on hadoop," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. IEEE, 2015, pp. 228–233.

[27] X. Lu, N. S. Islam, M. Wasi-Ur-Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda, "High-performance design of hadoop rpc with rdma over infiniband," in *2013 42nd International Conference on Parallel Processing*. IEEE, 2013, pp. 641–650.

[28] HPC-ABDS Kaleidoscope of over 350 Apache Big Data Stack and HPC Technologies. [Online]. Available: http://hpc-abds.org/kaleidoscope/

[29] G. Fox, J. Qiu, S. Jha, S. Ekanayake, and S. Kamburugamuve, "Big data, simulations and hpc convergence," Technical Report· January 2016, DOI: 10.13140/RG. 2.1, Tech. Rep., 1858.