SPIDAL Java: High Performance Data Analytics with Java and MPI on Large Multicore HPC Clusters

Saliya Ekanayake
School of Informatics and Computing
Indiana University, Bloomington
sekanaya@indiana.edu

Supun Kamburugamuve
School of Informatics and Computing
Indiana University, Bloomington
skamburu@indiana.edu

Geoffrey C. Fox
School of Informatics and Computing
Indiana University, Bloomington
gcf@indiana.edu

ABSTRACT

Within the last few years, there have been significant contributions to Java-based big data frameworks and libraries such as Apache Hadoop, Spark, and Storm. While these systems are rich in interoperability and features, developing high performance big data analytic applications is challenging. Also, the study of performance characteristics and high performance optimizations is lacking in the literature for these applications. By contrast, these features are well documented in the High Performance Computing (HPC) domain and some of the techniques have potential performance benefits in the big data domain as well. This paper presents the implementation of a high performance big data analytics library - SPIDAL Java - with a comprehensive discussion on five performance challenges, solutions, and speedup results. SPIDAL Java captures a class of global machine learning applications with significant computation and communication that can serve as a yardstick in studying performance bottlenecks with Java big data analytics. The five challenges present here are the cost of intra-node messaging, inefficient cache utilization, performance costs with threads, overhead of garbage collection, and the costs of heap allocated objects. SPIDAL Java presents its solutions to these and demonstrates significant performance gains and scalability when running on up to 3072 cores in one of the latest Intel Haswell-based multicore clusters.

Author Keywords
HPC; data analytics; Java; MPI; Multicore

ACM Classification Keywords
D.1.3 Concurrent Programming (e.g. Parallel Applications)

1. INTRODUCTION

A collection of Java-based big data frameworks have arisen since Apache Hadoop [24] - the open source MapReduce [5] implementation - including Spark, Storm, and Apache Tez. The High Performance Computing (HPC) enhanced Apache Big Data Stack (ABDS) [9] identifies over 300 frameworks and libraries across 21 different layers that are currently used in big data analytics. Notably, most of these software pieces are written in Java due to its interoperability, productivity, and ecosystem of supporting tools and libraries. While these big data frameworks have proven successful, a comprehensive study of performance characteristics is lacking in the literature. This paper investigates the performance challenges with Java big data applications through the implementation of Scalable, Parallel, and Interoperable Data Analytics Library (SPIDAL) Java - a highly optimized suite of parallel machine learning algorithms written in Java. It identifies five performance challenges that are not discussed well in the big data literature. Namely, these are the cost of intra-node messaging, inefficient cache utilization, performance costs with threads, overhead of garbage collection, and the cost of heap allocated objects. It presents optimizations in SPIDAL Java to overcome these and demonstrates significant performance gains on large multicore clusters. The novelty of this research is the comprehensive performance study of Java based big data analytics and the implementation SPIDAL Java application suite. While some of these challenges are studied in the High Performance Computing (HPC) domain, no published work appear to discuss these with regard to Java based big data applications.

Big data applications are diverse and this paper uses the Global Machine Learning (GML) class [7] as a yardstick due to its significant computation and communication. GML applications resemble Bulk Synchronous Parallel (BSP) model except the communication is global and does not overlap with computations. Also, they generally run multiple iterations of such compute and communicate phases until a stopping criterion is met.

SPIDAL Java is a suite of high performance GML applications with optimizations to overcome the above challenges. While these optimizations are not limited to HPC environments, SPIDAL Java is intended to run on large-scale modern day HPC clusters due to demanding computation and communication nature. The performance results of running on a latest production grade Intel Haswell HPC cluster - Juliet - demonstrate significant performance improvement over traditional implementations on the same advanced hardware.

The rest of the paper is organized as follows. Section 2 reviews related work and Section 3 introduces SPIDAL Java. Section 4 elaborates the high performance challenges and op-

SpringSim-HPC 2016 April 3-6, Pasadena, CA, USA
©2016 Society for Modeling & Simulation International (SCS)

timizations in SPIDAL Java. Section 5 presents experimental results from running the weighted Deterministic Annealing (DA) Multidimensional Scaling (MDS) algorithm [22] - in SPIDAL Java against real life health data. It includes speedup and scaling results for a variety of data sizes and compute cores (up to 3072 cores). Section 6 discusses open questions and possible improvements.Finally, Section 7 presents the conclusions of this work.

2. RELATED WORK

The high performance stream processing paper [13] describes a novel approach of using shared memory maps within Apache Storm. This is similar to SPIDAL Java's intra-node implementation except it implements a custom memory mapped based queue to coordinate between workers within a node. The results show considerable performance improvement over using TCP within a node.

Le Chai's Ph.D. [3] work identifies the bottleneck in intra-node communication with the traditional share-nothing approach of MPI and presents two approaches to exploit shared memory-based message passing for MVAPICH2. First is to use a user level shared memory map similar to SPIDAL Java. Second is to get kernel assistance to directly copy messages from one process's memory to the other. It also discusses how cache optimizations help in communication and how to address Non Uniform Memory Access (NUMA) environments.

Hybrid MPI (HMPI) [25] presents a similar idea to the zero intra-node messaging in SPIDAL Java. It implements a custom memory allocation layer that enables MPI ranks running within a node to have a shared heap space, thereby making it possible to copy messages directly within memory without external communication. HMPI optimizes for point-to-point messages only but provides seamless support over Xeon Phi accelerators.

An extension to HMPI that provides an efficient MPI collective implementation is discussed in [16]. It provides details on different techniques to implement collective primitives and how to select the best algorithm for a given collective in a NUMA setting. Also, it provides a comparison for reductions within a node against the popular OpenMP [4] library.

NCCL (pronounced "Nickel") [18] is an ongoing project from NVIDIA to provide an MPI-like collective library to use with multiple Graphical Processing Units (GPU) within a node. Traditional data transfer between GPUs involves communication with the host. NCCL instead avoids it and uses their GPUDirect implementation to copy data directly from one GPU to another. This is similar to our approach of data transfer between processes, except it happens between GPU nodes.

Project Tungsten from the company Databricks is a series of optimization techniques targeting Apache Spark [26] to bring its performance closer to native level. It includes several concepts similar to optimizations in SPIDAL Java such as off-heap data structures, in memory data transfer, and cache-aware computing.

3. SPIDAL JAVA

SPIDAL Java provides several multidimensional scaling, and clustering implementations. It is written in Java for productivity and interoperability. Also, with the optimizations discussed in Section 4 and the use of Just In Time (JIT) compiler, Java implementations give comparable performance to C/C++ code.

- **DA-MDS** implements an efficient weighted version of Scaling by MAjorization of a COmplicated Function (SMACOF) [1] that effectively runs in $O(N^2)$ compared to the original $O(N^3)$ implementation [22]. It also uses a deterministic annealing optimization technique [21, 14] to find the global optimum instead of local optima. Given an NxN distance matrix for N high dimensional data items, DA-MDS finds N lower dimensional points (usually 3 for visualization purposes) such that the sum of error squared is minimum. The error is defined as the difference between mapped and original distances for a given pair of points. DA-MDS also supports arbitrary weights and fixed points - data points that already have the same low dimensional mapping.
- **DA-PWC** is Deterministic Annealing Pairwise Clustering, which also uses the concept of DA, but for clustering [10, 21]. Its time complexity is $O(NlogN)$, which is better than existing $O(N^2)$ implementations [6]. Similar to DA-MDS, it accepts an NxN pairwise distance matrix and produces a mapping from point number to cluster number. It can also find cluster centers based on the smallest mean distance, i.e. the point with the smallest mean distance to all other points in a given cluster. If provided with a coordinate mapping for each point, it could also produce centers based on the smallest mean Euclidean distance and Euclidean center.
- **DA-VS** is Deterministic Annealing Vector Sponge, which is a recent addition to SPIDAL. It can perform clustering in both vector and metric spaces. Algorithmic details and an application of DA-VS to Proteomics data is available at [8].
- **MDSasChisq** is a general MDS implementation based on the LevenbergMarquardt algorithm [15]. Similar to DA-MDS, it supports arbitrary weights and fixed points. Additionally, it supports scaling and rotation of MDS mappings, which is useful when visually comparing 3D MDS outputs for the same data, but with different distance measures.

4. PERFORMANCE CHALLENGES WITH JAVA BIG DATA ANALYTICS

This section identifies the performance challenges with Java big data analytic applications and presents performance optimizations in SPIDAL Java to overcome them. While some of the optimizations are specific to SPIDAL Java, they can be used as general guidelines in developing other Java based big data applications.

4.1 Intra-node Communication

Intra-node communication on large multicore nodes causes a significant performance loss. Shared memory approaches have been studied as successful alternatives in Message Passing Interface (MPI) oriented researches [3],[25], and [16], however, none are available for Java. Also, popular big data frameworks too uses TCP for intra-node communications, which is prohibitively expensive for applications in SPIDAL Java.

SPIDAL Java uses OpenMPI for inter-node communication, but its shared memory support is very limited and does not include collectives such as variants of `allgather`, which is used in multidimensional scaling applications.

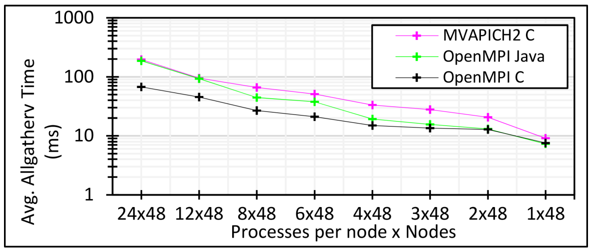

Figure 1: MPI `allgatherv` performance with different MPI implementations and varying intra-node parallelisms

Figure 1 plots arithmetic average (hereafter referred to simply as average) running times over 50 iterations of the MPI `allgatherv` collective against varying intra-node parallelism over 48 nodes in Juliet. Note all MPI implementations were using their default settings other than the use of Infiniband transport. This was a micro-benchmark based on the popular OSU Micro-Benchmarks (OMB) suite [20].

The purple and black lines show C implementations compiled against OpenMPI and MVAPICH2 [11], while the green is the same program in Java compiled against OpenMPI's Java binding. The Java binding is a thin wrapper around OpenMPI C implementation. All tests used a constant 24 million bytes (or 3 million double values) across different intra-node parallelism patterns to mimic the communication of DA-MDS, which uses `allgatherv` heavily, for large data. The experiment shows that the communication cost becomes significant with increasing processes per node and the effect is independent of the choice of MPI implementation and the use of Java binding in OpenMPI. However, an encouraging discovery is that all implementations produce a nearly identical performance for the single process per node case. While it is computationally efficient to exploit more processes, reducing the communication to a single process per node was further studied and successfully achieved with Java shared memory maps as discussed below.

SPIDAL Java's shared memory intra-node communication uses a custom memory maps implementation from OpenHFT's JavaLang[19] project to perform inter-process communication for processes within a node, thus eliminating any intra-node MPI calls. The standard MPI programming would require $O(R^2)$ of communications in a collective call, where R is the number of processes. In this optimization, we have effectively reduced this to $O(\hat{N}^2)$, where $\hat{N}$ is the number of nodes. Note this is an application level optimization rather than an improvement to a particular MPI implementation, thus, it will be possible for SPIDAL Java to be ported for future MPI Java bindings with minimal changes.

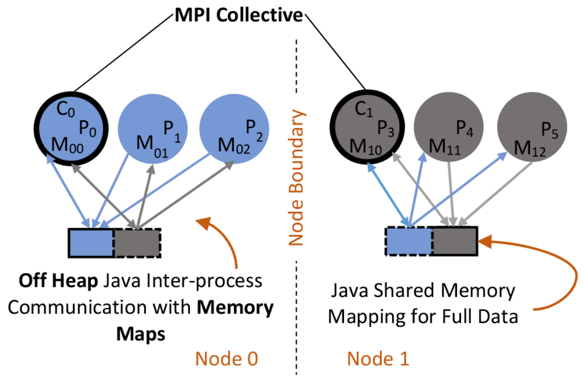

Figure 2: Intra-node message passing with Java shared memory maps

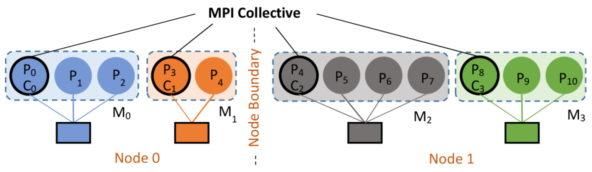

Figure 3: Heterogeneous shared memory intra-node messaging

Figure 2 shows the general architecture of this optimization where two nodes, each with three processes, are used as an example. Process ranks go from P_0 to P_5 and belong to `MPI_COMM_WORLD`. One process from each node acts as the communication leader - C_0 and C_1. These leaders have a separate MPI communicator called `COLLECTIVE_COMM`. Similarly, processes within a node belong to a separate `MMAP_COMM`, for example, M_{00} to M_{02} in one communicator for node 0 and M_{10} to M_{12} in another for node 1. Also, all processes within a node map the same memory region as an off-heap buffer in Java and compute necessary offsets at the beginning of the program. The takeaway point of this setup is the use of memory maps to communicate between processes and the reduction in communication calls. A typical call to an MPI collective will internally go through the following modified steps.

1. All processes, P_0 to P_5, write their partial data to the mapped memory region offset by their rank and node. See downward blue arrows for node 0 and gray arrows for node 1 in the figure.

2. Communication leaders, C_0 and C_1, wait for the peers, $\{M_{01}, M_{02}\}$ and $\{M_{10}, M_{11}\}$ to finish writing. Note leaders wait only for their peers in the same node.

3. Once the partial data is written, the leaders participate in the MPI collective call with partial data from their peers - upward blue arrows for node 0 and gray arrows for node

1. Also, the leaders may perform the collective operation locally on the partial data and use its results for the MPI communication depending on the type of collective required. MPI `allgatherv`, for example, will not have any local operation to be performed, but something like `allreduce` may benefit from doing the reduction locally. Note, the peers wait while their leader performs MPI communication.

4. At the end of the MPI communication, the leaders write the results to the respective memory maps - downward gray arrows for node 0 and blue arrows for node 1. This data is then immediately available to their peers without requiring further communication - upward gray arrows for node 0 and blue arrows for node 1.

This approach reduces MPI communication to just 2 processes, in contrast to a typical MPI program, where 6 processes would be communicating with each other. The two wait operations mentioned above can be implemented using memory mapped variables or with an MPI `barrier` on the `MMAP_COMM`; although the latter will cause intra-node messaging, experiments showed it to incur negligible costs compared to actual data communication.

While the uniform rank distribution across nodes and a single memory map group per node in Figure 2 is the optimal pattern to reduce communication, SPIDAL supports two heterogeneous settings as well. Figure 3 shows these two modes.

Non-uniform rank distribution - Juliet HPC cluster, for example, has two groups of nodes with different core counts (24 and 36) per node. SPIDAL Java supports different process counts per node to utilize all available cores in situations like this. Also, it automatically detects the heterogeneous configurations and adjusts its shared memory buffers accordingly.

Multiple memory groups per node - If more than 1 memory map per node (M) is specified, SPIDAL Java will select one communication leader per group even for groups within the same node. Figure 3 shows 2 memory maps per node. As a result, $O(\hat{N}^2)$ communication is now changed to $O((\hat{N}M)^2)$, so it is highly recommended to use a smaller M, ideally, $M = 1$.

4.2 Cache and Memory Utilization

While this is a generic performance consideration in computing, big data applications iterating over large arrays suffer significant performance loss if not properly utilized. SPIDAL Java employs 3 classic techniques from the linear algebra domain to improve cache and memory costs - blocked loops, 1D arrays, and loop ordering.

Blocked loops - Nested loops that access matrix structures use blocking such that the chunks of data will fit in cache and reside there for the duration of its use.

1D arrays for 2D data - 2D arrays representing 2D data require 2 indirect memory references to get an element. This is significant with increasing data sizes, so SPIDAL Java uses 1D arrays to represent 2D data. As such with 1 memory reference and computed indices, it can access 2D data efficiently.

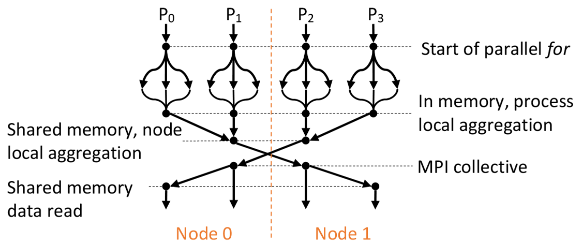

Figure 4: The architecture of utilizing threads for intra-process parallelism

This also improves cache utilization as 1D arrays are contiguous in memory.

Loop ordering - Data decomposition in SPIDAL Java blocks full data into rectangular matrices, so to efficiently use cache, it restructures nested loops that access these to go along the longest dimension within the inner loop. Note, this is necessary only when 2D array representation is necessary.

4.3 The Cost of Java Threads

While many of the big data frameworks employ threads to execute tasks, a comparison of performance against processes is not available for big data applications. SPIDAL Java applications support the hybrid approach of threads within MPI to create parallel `for` regions using Habanero Java library [12]. Note, threads perform computations only and do not invoke MPI operations. The parent process aggregates the results from threads locally as appropriate before using it in collective communications. Also, the previous shared memory messaging adds a second level of result aggregation within processes of the same node to further reduce communication. Figure 4 shows the usage of threads in SPIDAL Java with different levels of result aggregation.

Applications decompose data at the process level first and split further for threads. This guarantees that threads operate on non-conflicting data arrays; however, Figure 5, 6, and 9 show a rapid degrade in performance with increasing number of threads per process. Internal timings of the code suggest poor performance occurs in computations with arrays, which suggests possible false sharing and suboptimal cache usage. Therefore, while the communication bottleneck with default MPI implementations favored the use of threads, with shared memory intra-node messaging optimization in SPIDAL Java they offer no advantage, hence, processes are a better choice than threads for these applications.

4.4 The Overhead of Garbage Collection

It is critical to maintain a minimal memory footprint and reduce memory management costs in performance sensitive applications with large memory requirements such as those in SPIDAL Java. The Java Virtual Machine (JVM) automatically manages memory allocations and performs GC to reduce memory growth. It does so by segmenting the program's heap into regions called generations, and moving objects between these regions depending on their longevity. Every object starts in Young Generation (YG) and gets promoted to

Old Generation (OG) if they have lived long enough. Minor garbage collections happen in YG frequently and short-lived objects are removed without GC going through the entire Heap. Also, long-lived objects are moved to the OG. When OG has reached its maximum capacity, a full GC happens, which is an expensive operation depending on the size of the heap and can take considerable time. Also, both minor and major collections have to stop all the threads running in the process while moving the objects. Such GC pauses incur significant delays, especially for GML applications where slowness in one process affects all others as they have to synchronize on global communications.

Initial versions of SPIDAL Java followed the standard Object Oriented Programming (OOP), where objects were created as and when necessary while letting GC take care of the heap. The performance results, however, showed inconsistent behavior, and detailed GC log analysis revealed processes were paused most of the time to perform GC. Also, the max heap required (JVM -Xmx setting) to get reasonable timing quickly surpassed the physical memory in Juliet cluster with increasing data sizes.

The optimization to overcome these memory challenges was to compute the total memory allocation required for a given data size and statically allocate required arrays. Also, the computation codes reuse these arrays creating no garbage. Another optimization is to use off-heap buffers for communications and other static data, which is discussed in the next subsection.

While precomputing memory requirements is application dependent, static allocation and array reuse can bring down GC costs to negligible levels. Benefits of this approach in SPIDAL Java are as follows.

Zero GC - Objects are placed in the OG and no transfer of objects from YG to OG happens in run-time, which avoids full GC.

Predictable performance - With GC out of the way, the performance numbers agreed with expected behavior of increasing data and parallelism.

Reduction in memory footprint - A DA-MDS run of 200K points running with 1152 way parallelism required about 5GB heap per process or 120 GB per node (24 processes on 1 node), which hits the maximum memory per node in our cluster, which is 128GB. The improved version required less than 1GB per process for the same parallelism, giving about 5x improvement on memory footprint.

4.5 The Cost of Heap Allocated Objects

With traditional heap allocated objects, the JVM has to make extra copies whenever a native operation is performed on it. One reason for this is JVM cannot guarantee that the memory reference to a buffer will stay intact during a native call because it is possible for a GC compaction to happen and move the buffer to a different place in the heap. The solution employed in SPIDAL Java is to use direct buffers, which are off-heap data structures, that allows the JVM to perform fast native operations without data copying.

SPIDAL Java uses off-heap buffers efficiently for the following 3 tasks.

Initial data loading - Input data in SPIDAL Java are NxN binary matrices stored in 16-byte (short) big-endian or little-endian format. Java stream APIs such as the typical $DataInputStream$ class are very inefficient in loading these matrices. Instead, SPIDAL Java memory maps these matrices (each process maps only the chunk it operates on) as Java direct buffers.

Intra-node messaging - Intra-node process-to-process communications happen through custom off heap memory maps, thus avoiding MPI within a node. While Java memory maps allow multiple processes to map the same memory region, it does not guarantee writes from one process will be visible to the other immediately. The OpenHFT Java Lang Bytes [19] used here is an efficient off-heap buffer implementation, which guarantees write consistency.

MPI communications - While OpenMPI supports both on- and off-heap buffers for communication, SPIDAL Java uses statically allocated direct buffers, which greatly reduce the cost of MPI communication calls.

5. TECHNICAL EVALUATION

This section presents performance results of SPIDAL Java to demonstrate the improvements of previously discussed optimization techniques. These were run on Juliet, which is a production grade Intel Haswell HPC cluster with 128 nodes total, where 96 nodes have 24 cores (2 sockets x 12 cores each) and 32 nodes have 36 cores (2 sockets x 18 cores each) per node. Each node consists of 128GB of main memory and 56Gbps Infiniband interconnect. The total core count of the cluster is 3456, which can be utilized with SPIDAL Java's heterogeneous support, however, performance testings were done with a uniform rank distribution of 24x128 - 3072 cores.

Figures 5, 6, and 9 show the results for 3 DA-MDS runs with 100K (2E10 bytes), 200K (4E10 bytes), and 400K (1.6E11 bytes) data points. Note, with $O(N^2)$ runtime, 400K tests take 4 times that of 200k, hence these were done with less number of iterations to meet HPC resource allocation times. This does not affect performance characteristics in anyway as each iteration is independent and the number of iterations determine only the accuracy of results. The green line is for SPIDAL Java with shared memory intra-node messaging, zero GC, and cache optimizations. The blue line is for Java and OpenMPI with shared memory intra-node messaging and zero GC, but no cache optimizations. The red line represents Java and OpenMPI with no optimizations. The default implementation (red line) could not handle 400K points on 48 nodes, hence, it is not shown in Figure 9.

Patterns on the X-axis of the graphs show the combination of threads (T), processes (P), and the number of nodes. The total number of cores per node was 24 (12 on each socket), so the Figure 5 through 8 show all possible combinations that give 24-way parallelism per node. OpenMPI has a number of `allgather` implementations and these were using the linear ring implementation of MPI `allgatherv` as it gave the best performance. The Bruck [2] algorithm, which is an

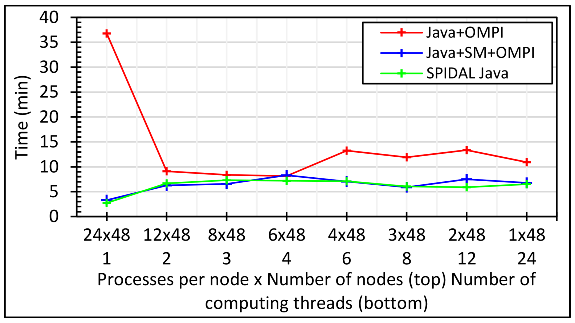

Figure 5: DA-MDS 100K performance with varying intra-node parallelism

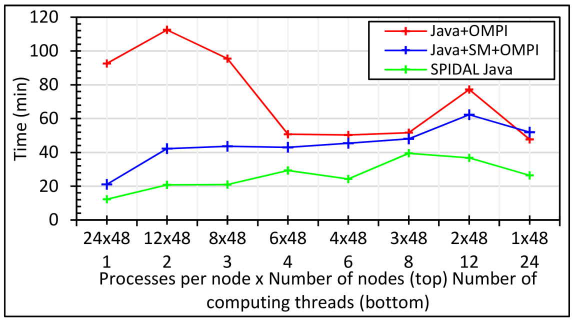

Figure 6: DA-MDS 200K performance with varying intra-node parallelism

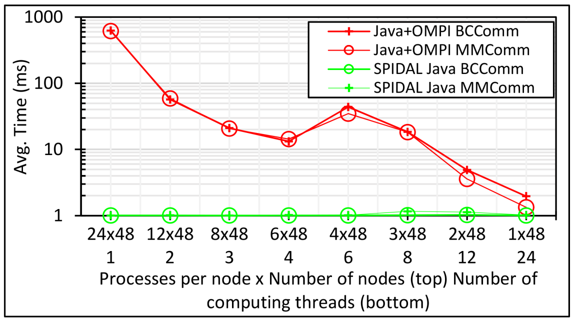

Figure 7: DA-MDS 100K `allgatherv` performance with varying intra-node parallelism

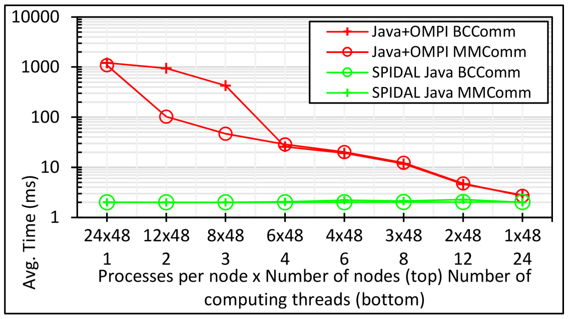

Figure 8: DA-MDS 200K `allgatherv` performance with varying intra-node parallelism

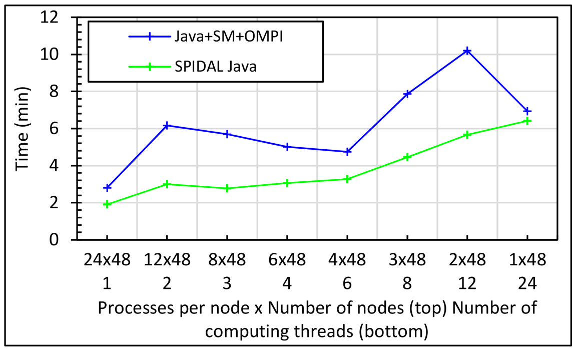

Figure 9: DA-MDS 400K performance with varying intra-node parallelism

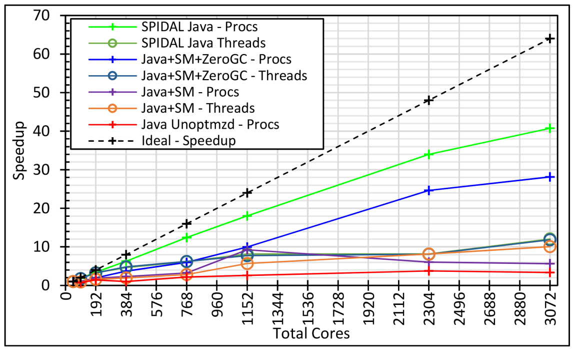

Figure 10: DA-MDS speedup for 200K with different optimization techniques

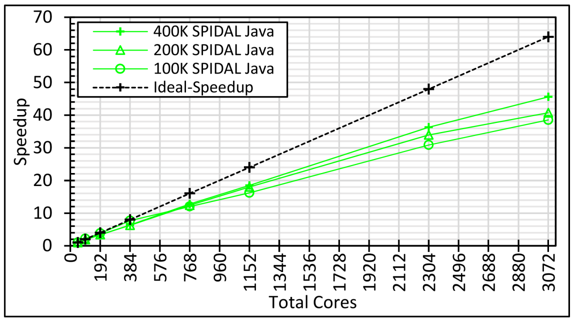

Figure 11: DA-MDS speedup with varying data sizes

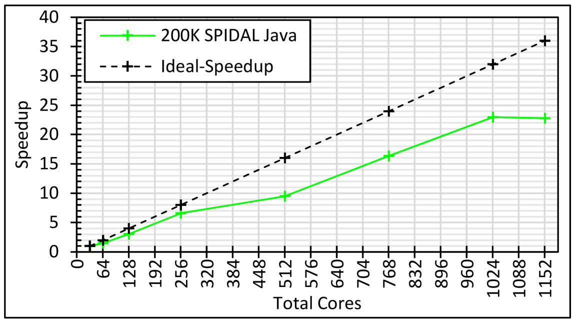

Figure 12: DA-MDS speedup on 36 core nodes for 200K data

efficient algorithm for all-to-all communications, performed similarly but was slightly slower than the linear ring for these tests.

Ideally, all these patterns should perform the same because the data size per experiment is constant, however, results show the default Java and OpenMPI based implementation significantly degrades in performance with large process counts per node (red-line). In addition, increasing the number of threads, while showing a reduction in the communication cost (Figure 7 and 8), does not improve performance. The Java and OpenMPI memory mapped implementation (blue-line) surpasses default MPI by a factor of 11x and 7x for 100K and 200K tests respectively for all process (leftmost 24x48) cases. Cache optimization further improves performance significantly across all patterns especially with large data, as can be seen from the blue line to the green line (SPIDAL Java).

The DA-MDS implementation in SPIDAL Java, for example, has two call sites to MPI `allgatherv` collective, BCComm and MMComm, written using OpenMPI Java binding [23]. They both communicate an identical number of data elements, except one routine is called more times than the other. Figures 7 and 8 show the average times in log scale for both of these calls during the 100K and 200K runs.

SPIDAL Java achieves a flat communication cost across different patterns with its shared memory-based intra-node messaging in contrast to the drastic variation in default OpenMPI. Also, the improved communication is now predictable and acts as a linear function of total points (roughly 1ms to 2ms when data size increased from 100K to 200K). This was expected and is due to the number of communicating processes being constant and 1 per node.

Figure 11 shows speedup for varying core counts for three data sizes - 100K, 200K, and 400K. These were run as all processes because threads did not result in good performance. None of the three data sizes were small enough to have a serial base case, so the graphs use the 48 core as the base, which was run as 1x48 - 1 process per node times 48 nodes. SPIDAL Java computations grow $O(N^2)$ while communications grow $O(N)$, which intuitively suggests larger data sizes should yield better speedup than smaller ones and the results confirm this behavior.

Figure 12 shows similar speedup results when run on Juliet's 36 core nodes. The number of cores used within a node is equal to the X-axis' value divided by 32 - the total 36 core node count. It shows a plateau in speedup after 32 processes per node, which is due to hitting the memory bandwidth.

Figure 10 shows DA-MDS speedup with different optimization techniques for 200K data. Also, for each optimization, it shows the speedup of all threads vs. all processes within a node. The total cores used range from 48 to 3072, where SPIDAL Java's 48 core performance was taken as the base case (green line) for all other implementations. The bottom red line is the Java and OpenMPI default implementation with no optimizations. Java shared memory intra-node messaging, zero GC, and cache optimizations were added on top of it. Results show that Java and OpenMPI with the first two optimizations (blue line) and all processes within a node surpass all other thread variations. This is further improved with cache optimization (SPIDAL Java - green line) and gives the best overall performance.

NAS BM	Class	Fortran -O3	NAS Java	Java with SPIDAL Optimizations
CG	A	0.97	1.11 s	0.81
CG	B	121	138	129
CG	C	337	418	353
LU	W	4.97	272	8.0
LU	A	32	2280	48
LU	B	152	–	215
LU	C	613	–	904

Table 1: NAS serial benchmark total time in seconds

PxN	Total Cores	C -O3	SPIDAL Java
24x48	1152	401	282
16x48	768	600	421
8x48	384	1260	890
1x48	48	9053	6363
1x1*	1	0.30	0.28

Table 2: DA-MDS block matrix multiplication time per iteration. 1x1* is the serial version and mimics a single process in 24x48.

Table 1 and 2 show native Fortran and C performance against Java for a couple of NAS [17] serial benchmarks and block matrix multiplication in DA-MDS. Note. the native NAS implementations are directly obtained from the NAS benchmark site. Table 1 also shows default NAS Java performance. Note, the 1x1 serial pattern in Table 2 mimics the matrix sizes for 1 process in 24x48. The results suggest Java yields competitive performance using the optimizations discussed in this paper.

6. FUTURE WORK

The current data decomposition in SPIDAL Java assumes a process would have enough memory to contain the partial input matrix and intermediate data it operates on. This sets an upper bound on the theoretical maximum data size it could handle, which is equal to the physical memory in a node. However, we could improve on this with a multi-step computing approach, where a computation step is split into multiple computation and communication steps. This will increase the number of communications, but will still be worthwhile to investigate further.

7. CONCLUSION

Performance results of SPIDAL Java show it scales and performs well in large HPC clusters. Also, the optimizations to overcome performance challenges made it possible to run SPIDAL Java applications on much larger data sets than what was possible in the past while still achieving excellent scaling results. The improved shared memory intra-node communication is pivotal to the gains in SPIDAL Java and it is the first such implementation for Java to the best of our knowledge.

ACKNOWLEDGMENTS

This work was partially supported by NSF CIF21 DIBBS 1443054 and AFOSR FA9550-13-1-0225 awards. We thank Intel for their support of the Juliet system. Also, we acknowledge the support of system administrators of the Digital Science Center (DSC) at Indiana University.

REFERENCES

1. Borg, I., and Groenen, P. *Modern Multidimensional Scaling: Theory and Applications*. Springer, 2005.
2. Bruck, J., Ho, C.-T., Upfal, E., Kipnis, S., and Weathersby, D. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Trans. Parallel Distrib. Syst. 8*, 11 (Nov. 1997), 1143–1156.
3. Chai, L. *High Performance and Scalable MPI Intra-Node Communication Middleware for Multi-Core Clusters*. PhD thesis, Graduate School of The Ohio State University, 2009.
4. Dagum, L., and Menon, R. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng. 5*, 1 (Jan. 1998), 46–55.
5. Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, USENIX Association (Berkeley, CA, USA, 2004), 10–10.
6. Fox, G. Robust scalable visualized clustering in vector and non vector semi-metric spaces. *Parallel Processing Letters 23*, 2 (2013).
7. Fox, G., Jha, S., Qiu, J., Ekanayake, S., and Luckow, A. Towards a comprehensive set of big data benchmarks. Tech. rep., 2015.
8. Fox, G., Mani, D., and Pyne, S. Parallel deterministic annealing clustering and its application to lc-ms data analysis. In *Big Data, 2013 IEEE International Conference on* (Oct 2013), 665–673.
9. Fox, G., Qiu, J., Kamburugamuve, S., Jha, S., and Luckow, A. Hpc-abds high performance computing enhanced apache big data stack. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on* (May 2015), 1057–1066.
10. Fox, G. C. Deterministic annealing and robust scalable data mining for the data deluge. In *Proceedings of the 2Nd International Workshop on Petascal Data Analytics: Challenges and Opportunities*, PDAC '11, ACM (New York, NY, USA, 2011), 39–40.
11. Huang, W., Santhanaraman, G., Jin, H., Gao, Q., and Panda, D. Design of high performance mvapich2: Mpi2 over infiniband. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, vol. 1 (May 2006), 43–48.
12. Imam, S., and Sarkar, V. 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*, PPPJ '14, ACM (New York, NY, USA, 2014), 75–86.
13. Kamburugamuve, S., Ekanayake, S., Pathirage, M., and Fox, G. Towards High Performance Processing of Streaming Data in Large Data Centers. `http://dsc.soic.indiana.edu/publications/high_performance_processing_stream.pdf`, 2016. Technical Report.
14. Klock, H., and Buhmann, J. M. Multidimensional scaling by deterministic annealing. In *Proceedings of the First International Workshop on Energy Minimization Methods in Computer Vision and Pattern Recognition*, EMMCVPR '97, Springer-Verlag (London, UK, UK, 1997), 245–260.
15. Levenberg, K. A method for the solution of certain non-linear problems in least squares. *Quarterly Journal of Applied Mathmatics II*, 2 (1944), 164–168.
16. Li, S., Hoefler, T., and Snir, M. Numa-aware shared-memory collective communication for mpi. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, ACM (New York, NY, USA, 2013), 85–96.
17. NAS Parallel Benchmarks. `http://www.nas.nasa.gov/publications/npb.html`.
18. NVIDIA NCCL. `https://github.com/NVIDIA/nccl`.
19. OpenHFT JavaLang Project. `https://github.com/OpenHFT/Java-Lang`.
20. OSU Micro-Benchmarks. `http://mvapich.cse.ohio-state.edu/benchmarks/`.
21. Rose, K., Gurewwitz, E., and Fox, G. A deterministic annealing approach to clustering. *Pattern Recogn. Lett. 11*, 9 (Sept. 1990), 589–594.
22. Ruan, Y., and Fox, G. A robust and scalable solution for interpolative multidimensional scaling with weighting. In *9th IEEE International Conference on eScience, eScience 2013, Beijing, China, October 22-25, 2013* (2013), 61–69.
23. Vega-Gisbert, O., Roman, J. E., Groß, S., and Squyres, J. M. Towards the availability of java bindings in open mpi. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, ACM (New York, NY, USA, 2013), 141–142.
24. White, T. *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.
25. Wickramasinghe, U. S., Bronevetsky, G., Lumsdaine, A., and Friedley, A. Hybrid mpi: A case study on the xeon phi platform. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '14, ACM (New York, NY, USA, 2014), 6:1–6:8.
26. Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, USENIX Association (Berkeley, CA, USA, 2010), 10–10.