SPIDAL: 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

High Performance Computing (HPC) cluster nodes with multicore chips offer a large number of parallel computing units per node and are equipped with fast interconnects and large memory. We identify a class of big data machine learning problems that demands substantial communication, but find that despite the rich ecosystem of big data frameworks - many written in Java - it is challenging to achieve high performance for reasons such as costly inter-process communication, sub-optimal cache utilization, and higher memory footprint. Our approach to solving this is written on Java to enable integration with existing big data software stacks such as those from Apache. We employ MPI due to its proven high performance and improves further by exploiting Java off heap memory maps to avoid intra-node messaging. Also, we do cache optimization and reduce memory reference costs. Further, we utilize static arrays and other off heap data structures to reduce memory usage and Garbage Collection (GC) costs. We also investigate the hybrid use of threads and processes, but find processes to outperform threads in all cases on our current 24 core and 36 core Haswell nodes. We apply these techniques to implement a high performance data analytics library, SPIDAL, and present performance results of running it on a latest Intel Haswell HPC cluster consisting 3456 cores total.

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

ACM Classification Keywords
D.1.3 Concurrent Programming (e.g. Parallel Applications): See: `http://www.acm.org/about/class/1998/` for more information and the full list of ACM classifiers and descriptors.

1. INTRODUCTION

The past few years have witnessed a rise in big data problems both in industry and academia. Unlike traditional HPC problems, which are mostly compute intensive, the nature of these problems cannot be characterized along a single dimension. Our previous work on Ogres [11] present a multidimensional and multifaceted classification scheme as a solution. One of the axes in Ogres is the problem architecture, which identifies the ”shape” of the problem and we identify six prominent classes of big data problems shown in Figure 1

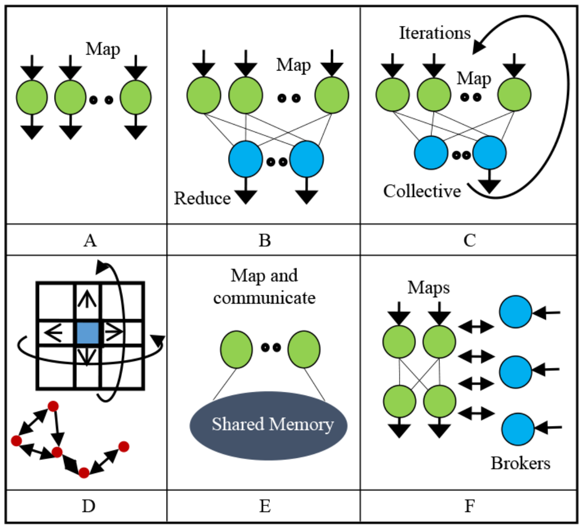

Figure 1. Six prominent problem architectures of big data

Figure 1 A to F represent pleasingly parallel, classic map-reduce, map-collective, graph or point to point, shared memory, and streaming classes of applications. We focus on map-collective architecture, which is common in many of the data science problems [6] and is interesting for two reasons - multiple iterations and global communication. Let us consider the parallelization of the $k - means$ algorithm over a given N points to illustrate these features of map-collective type. Each parallel unit of computing starts with the same set of k initial vectors (cluster centers) and assigns its points to the nearest center. They **collectively** communicate the sum of points (i.e. sum of vector components) for each center in the next step. The average of **global sums** determines the new cluster centers. If the difference between new and old centers is larger than a given threshold the program continues to the next **iteration** replacing initial centers with centers found in the current iteration. The boldfaced words highlight the fact that computations happen independently followed by a global communication, the result of which is necessary to continue next step of computing. A single or multiple of such map and

SpringSimSim-TMSDEVS 2015 July 21-26 Pasadena, CA, USA

collective phases will be iterated over until a stopping criterion is met. This closely resembles the Bulk Synchronous Parallel (BSP) model with the exceptions of global communication, and non-overlapping computations and communication.

We identify modern day HPC clusters as best suited for such map-collective applications for reasons - 1. large number of computing units per node and as a total, 2. memory per node is high, and 3. high speed interconnect between nodes. However, even with these advanced hardware, we find it is challenging to develop high performance data analytic applications of map-collective nature. Namely, these issues are expensive and varying intra-node communication, suboptimal cache and memory utilization, performance variation with threads, large memory requirement per process, and Garbage Collection (GC) cost and inefficient use of heap.

Scalable, Parallel, and Inter-operable Data Analytics Library (SPIDAL) is aimed at providing a highly optimized suite of Global Machine Learning (GML) applications to analyze big data problems; hence it is vital to overcome these challenges, especially to leverage HPC infrastructure. In this paper we present optimization techniques that yield significant performance improvement in SPIDAL when run on large multicore HPC clusters. The techniques we present are not specific to SPIDAL applications and can be applied to other applications requiring high performance on modern HPC clusters.

The rest of the paper is organized as follows. We introduce the SPIDAL in Section 2, and elaborate our high performance optimizations in Section 3 to overcome the above challenges. Section 4 presents experiment results of running the weighted Deterministic Annealing (DA) Multidimensional Scaling (MDS) algorithm [24] - in SPIDAL against real life health data. We include speedup and scaling results for a variety of data sizes and compute cores (up to 3072 cores). Section 5 discuss open questions and possible improvements. Section 6 reviews related work and we conclude this paper in Section 7.

2. SPIDAL

SPIDAL applications are written in Java and uses memory maps and Message Passing Interface (MPI) for inter-process communication. There is a well established collection of open big data software such as those available in Apache Big Data Stack (ABDS) [9] and many of these are either written in Java or support Java Application Program Interface (API). We believe it is important to be able to integrate SPIDAL with such solutions in the long run, hence the choice of Java to author SPIDAL.

We currently include the following GML applications in SPIDAL.

- 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 [24]. Also, it uses deterministic annealing optimization technique [23, 15] 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 (usually 3 for visualization purposes) points 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 too uses the concept of DA, but for clustering [10, 23]. Its time complexity is $O(NlogN)$, which is better than existing $O(N^2)$ implementations [7]. 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 this to Proteomics data is available at [8]
- MDSasChisq is a general MDS implementation based on LevenbergMarquardt algorithm [16]. 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.

In addition to GML application, SPIDAL also includes a Web based interactive 3D data visualization tool - PlotViz [22]. A real life use case on using DA-MDS, DA-PWC, and PlotViz to analyze gene sequences is available at [18].

3. HIGH PERFORMANCE OPTIMIZATIONS

In our attempt to achieve high performance with SPIDAL, we identified intra-node communication of a participating global collective primitive poses the most overhead. Threads within a node is a natural choice to overcome this, but we found processes to perform better due to cache coherency issues in threads. Cache and memory reference costs were bottlenecks in general, so we discuss optimization techniques below. Also, SPDIAL algorithms have $O(N^2)$ memory requirement, so keeping a minimal memory footprint without GC costs was a challenge. Moreover, loading initial NxN distance matrices was costly and required techniques other than the commonly used stream APIs in Java. We elaborate the details and our optimization techniques to overcome these challenges in SPIDAL below.

3.1 Zero Intra-node Communication

The Haswell cluster we use consists up to 36 cores available per node for computation, but when spawned that many processes, the global communication primitives produced non-negligible performance degrade. We found this to be true across different MPI implementations and language bindings as shown in Figure 2.

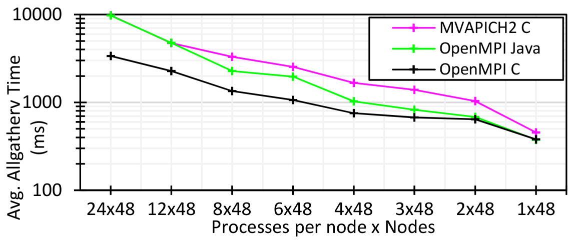

Figure 2. Allgatherv performance with different MPI implementations and varying intra-node parallelisms

We plot arithmetic average (hereafter referred simply as average) running times of the MPI $allgatherv$ collective against varying intra-node parallelism over 48 nodes in Figure 2. Note, all MPI implementations were run with their default settings, except were using Infiniband transport. This was a micro-benchmark based on the popular OSU Micro-Benchmarks (OMB) suite [21] and we implemented two separate programs - one in C and the other in Java using OpenMPI Java binding. The native C code was compiled and run against two standard MPI implementations - OpenMPI and MVAPICH2 [12]. The total number of bytes was kept constant at 24 million bytes (or 3 million double values) across different patterns. This was because we wanted to mimic the communication of DA-MDS, which uses $allgatherv$ heavily, for large data. The experiment shows 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, the encouraging discovery is that all implementations produce nearly identical performance for the single process per node case. While it is computationally efficient exploit more processes, reducing the communication to a single process per node was hence further studied and successfully achieved with Java shared memory maps as discussed below.

We note that shared memory collectives support is limited in OpenMPI and does not implement any variant of the $allgather$. Also, we found our implementation performed better even over existing ones such as $allreduce$. Previous work on shared memory collectives [3] and [17] suggest good performance, but they do not support a Java binding, hence could not be used for SPIDAL.

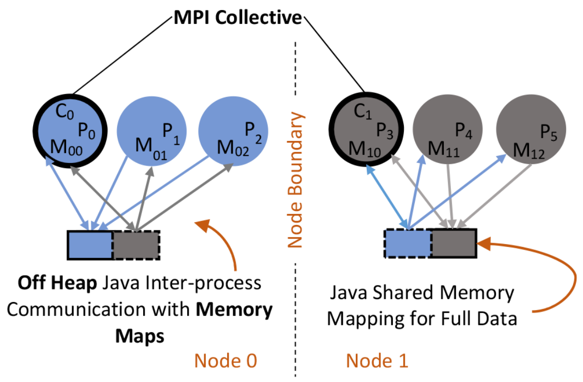

Figure 3. Intra-node message passing with Java shared memory maps

Our solution exploits Java shared memory maps 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 our implementation, 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. This will make it possible for SPIDAL to be ported for future MPI Java bindings with minimal changes.

Figure 3 shows the general architecture of this optimization where two nodes, each with three processes, are shown as an example. Processes are ranked from P_0 to P_5 and they belong to MPI_COMM_WORLD. One process from each node is designated as the communication leader - C_0 and C_1. We group them into a separate MPI communicator called COLLECTIVE_COMM. Similarly, processes within a node are put into MMAP_COMM. These are shown as M_{00} to M_{02} for node 0 and M_{10} to M_{12} 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. With this setup, a typical call to an MPI collective is carried out with reduced communication using the following 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 its peers - upward blue arrow for node 0 and gray arrow 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 the respective memory maps - downward gray arrow for node 0 and blue arrow for node 1. This data is then immediately available to their peers without requiring further communications - upward gray arrows for node 0 and blue arrows for node 1.

We reduce MPI communication to just 2 processes, in contrast to a typical MPI program, where 6 processes would be communicating with each other. Also, the two wait operations mentioned above can be implemented using memory mapped variables. One could also use an MPI $barrier$ on the MMAP_COMM, which even though will cause intra-node messaging, we found it to incur negligible costs compared to actual data communication.

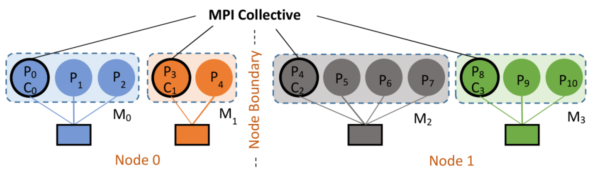

Figure 4. Heterogeneous shared memory intra-node messaging

Figure 3 shows uniform rank distribution across nodes and a single memory map group per node. While this is the optimal pattern we would recommend, SPIDAL supports two heterogeneous settings as shown in Figure 4. These are described below.

Non-uniform rank distribution - Our production HPC cluster, for example, has two groups of nodes with different core counts (24 and 36) per node. In such situations, we support running different process counts per node. SPIDAL automatically detects such heterogeneous configurations and adjust its shared memory buffers accordingly.

Multiple memory groups per node - If more than 1 memory maps per node (M) is specified, SPIDAL will select one communication leader per group even for groups within the same node. Figure 4 shows 2 memory maps per node. Note, $O(\hat{N}^2)$ communication is now changed to $O((\hat{N}M)^2)$, so we highly recommend using a smaller M, ideally $M = 1$.

3.2 Cache and Memory Optimization

We employ 3 classic techniques from the linear algebra domain to improve cache and memory costs - blocked loops, 1D arrays, and loop ordering.

Blocked loops - We block the parts of code that access matrix structures in nested loops 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 data represented as 2D arrays costs 2 indirect memory references to get an element. This is significant with increasing data sizes, so have reduced all such arrays to 1D arrays, so with 1 memory reference and computed indices we can access 2D data efficiently. Also, this improve cache utilization as 1D arrays are contiguous in memory.

Loop ordering - Data decomposition in SPIDAL algorithms blocks full data into rectangular matrices, so we have restructured nested loops that access these to go along the longest dimension within the inner loop to efficiently use cache.

3.3 MPI Over Threads

All SPIDAL applications supports the hybrid approach of threads within MPI. Threads are used to create parallel for regions. While it is not complicated to implement parallel loops with Java thread constructs, We use an OpenMP [4] like alternative in Java - Habanero Java library [13] - from Rice University for productivity and performance. Note, threads perform computations only and do not invoke MPI operations. The results from threads are locally aggregated as appropriate before the parent process use it in collective communications. Note, with our previous zero intra-node messaging optimization a further local aggregation happens even within processes of the same node to reduce communication. The architecture of utilizing threads is shown in Figure 5

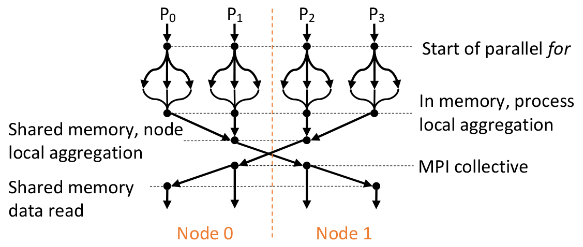

Figure 5. The architecture of utilizing threads for intra-process parallelism

Data decomposition is done at the process level first and splitting further for threads initially. This guarantees that threads operate on non conflicting data arrays; however, false sharing is still possible and we find this to be a significant bottleneck with increasing number of threads per process as shown in Figure 6 and Figure 7. While the communication bottleneck with default MPI implementations favored the use of threads, with our zero-intra node messaging we find threads offer no advantage due to this behavior, hence we recommend the use of all MPI over threads in SPIDAL. Also, techniques such as padding to avoid false sharing increase the memory requirement per process, which is not affordable with SPIDAL applications.

3.4 Minimal Memory and Zero Full GC

Maintaining a minimal memory footprint and reducing memory management costs are two aspects critical to performance sensitive applications with large memory requirements such as those in SPIDAL. Java Virutal Machine (JVM) automatically manges memory allocations and performs GC to reduce memory growth by deallocating unused objects. 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 a 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 followed the standard Object Oriented Programming (OOP) of Java, where objects were created as and when necessary, while letting GC take care of the heap. The performance results with this, however,showed inconsistent behavior, and detailed GC log analysis revealed processes were paused most of the time to perform GC. Also, we noticed the max heap required (JVM -Xmx setting) to get reasonable timing quickly surpassed the physical memory in our cluster with increasing data sizes. After careful analysis of the code we observed that total memory allocation required for a given data size could be computed and allocated ahead of time. Also, we changed the code to make the memory required the minimum by reusing arrays used computations, and doing communications with static off-heap buffers (see next optimization). We achieved the following improvements with these techniques.

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.

3.5 Off Heap Data Structures

Java off-heap data structures, as the name implies, are allocated outside the GC managed heap and are represented as Java direct buffers. With traditional heap allocated buffers, the JVM has to make extra copies whenever a native operation is performed on it. One reason for this is that JVM cannot guarantee 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 heap. Direct buffers, being outside of heap, overcomes this problem, thus letting JVM to perform fast native operations without data copying.

We use off-heap buffers efficiently for the following 3 tasks in SPIDAL algorithms.

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

Intra-node messaging - We use memory mapped buffers to do intra-node process to process communication, 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. Therefore, we use OpenHFT Java Lang Bytes [20], which is an efficient off-heap buffer implementation with guarantees on write consistency.

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

4. TECHNICAL EVALUATION

Here we present our experimental results to demonstrate the performance advantages of previously discussed optimization techniques. We tested these on a production grade Intel Haswell HPC cluster, Juliet, which has 128 nodes total, where 96 nodes has 24 cores (2 sockets x 12 cores each) and 32 nodes has 36 cores (2 sockets x 18 cores each) per node.Each node consists 128GB of main memoryand 56Gbps Infiniband interconnect. Note. the total core count of the cluster is 3456, which we can utilize with SPIDAL's heterogeneous support, but for performance testing we did uniform rank distribution of 24x128 - 3072 cores.

Figures 6, 7, and 8 show the results for 3 full DA-MDS runs with 100K, 200K, and 400K data points. The red line is with zero intra-node messaging, zero GC, and cache optimization. The blue is with zero intra-node messaging and zero GC, but no cache optimization. The green is with no optimizations. Note, the default implementation could not handle 400K points on 48 nodes, hence not shown in Figure 8.

Patterns on the X-axis of the graphs show the combination of threads (T), processes (P), and number of nodes. The total number of cores per node was 24 (12 on each socket), so we tested all possible combinations that give 24-way parallelism per node. We use process pinning to avoid the Operating System (OS) from moving processes within a node, which would diminish data locality benefits of allocated buffers. The number of cores pinned to a process was $24/P$ and any threads within a process is also pinned to a separate core. We use OpenHFT Thread Affinity [14] library to bind Java threads to cores. OpenMPI has a number of $allgather$ implementations and we used the linear ring implementation of MPI $allgatherv$ as it gave the best performance. The Bruck [2] algorithm, which is an efficient algorithm for all-to-all communications, performed similarly, but was slightly slower than linear ring in this case.

Ideally, all these patterns should perform the same as we keep constant data size per experiment. However, we see default MPI based implementation significantly degrades in performance with large process counts per node (green-line). Also, increasing the number of threads, though reduce the communication cost, does not improve performance. The SPDIAL memory mapped implementation surpasses default MPI by a factor of 11X and 7X for 100K and 200K tests respectively for all process (leftmost 24x48) case. Cache optimization further improves performance significantly across all patterns, especially with large data as can be seen from the blue line to the red line. we achieved similar performance for other data sizes, 800K and 1 million, but are not shown in here to save space.

The DA-MDS implementation in SPIDAL, for example, has two call sites to MPI $allgatherv$ collective, BCComm and MMComm, written using OpenMPI Java binding [25]. They both communicate identical number of data elements, except one routine is called more times than the other. Figures 9 and 10 show the average times in log scale for both of these calls during the 100K and 200K runs.

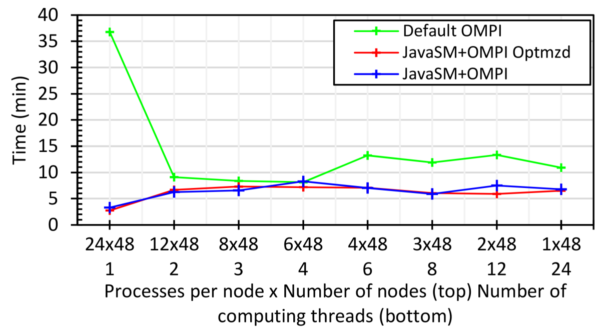

Figure 6. DA-MDS 100K performance with varying intra-node parallelism

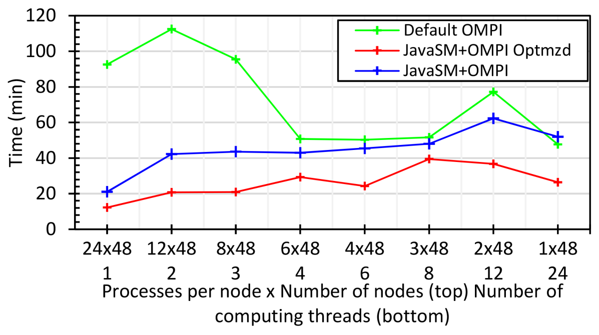

Figure 7. DA-MDS 200K performance with varying intra-node parallelism

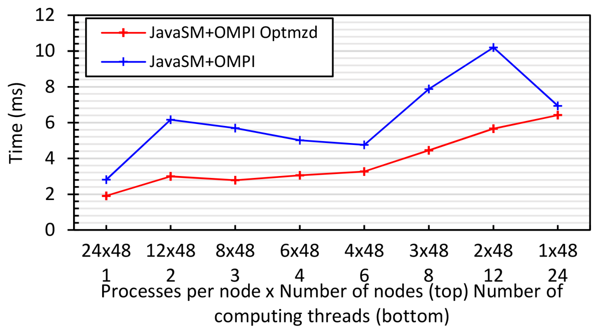

Figure 8. DA-MDS 400K performance with varying intra-node parallelism

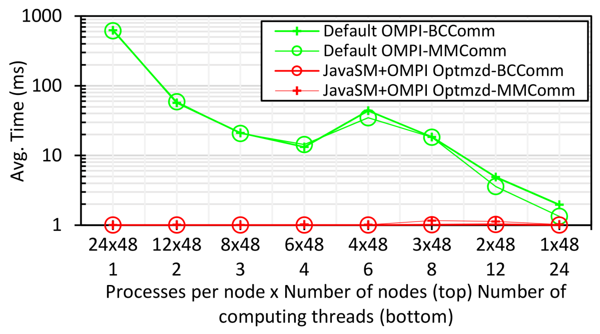

Figure 9. DA-MDS 100K Allgatherv performance with varying intra-node parallelism

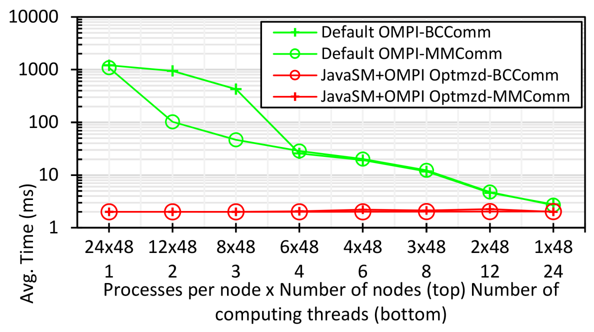

Figure 10. DA-MDS 200K Allgatherv performance with varying intra-node parallelism

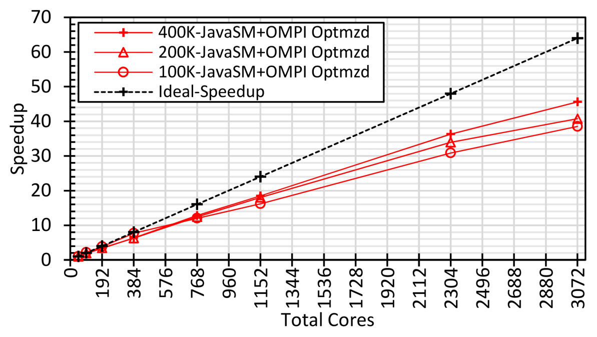

Figure 11. DA-MDS speedup with varying data sizes

We note the flat communication across different patterns with SPDIAL's shared memory based intra-node messaging in contrast to the drastic variation in default MPI. 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 is as 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. Neither of the three data sizes was small enough to have a serial base case, so we used the 48 core as the base, which was run as 1x48 - 1 process per node times 48 nodes. SPIDAL computations grow $O(N^2)$ while communications grow $O(N)$, which intuitively suggests larger data sizes should yield better speedup than smaller ones and we see our results agree with it.

We also tested the speedup with different optimization techniques for both MPI and threads within a node for 200K data as shown in Figure 12. Total cores used range from 48 to 3072. We used the 48 core case of all optimized (red line) version as the base case for all other implementations. The bottom green line is the default MPI implementation with no optimizations. We add Java shared memory (JavaSM), zero GC, and cache optimizations on top of it. We find JavaSM with zero GC (blue line) and all processes surpass all other thread variations. This is further improved with cache optimization (red line) and gives the best performance overall.

5. FUTURE WORK

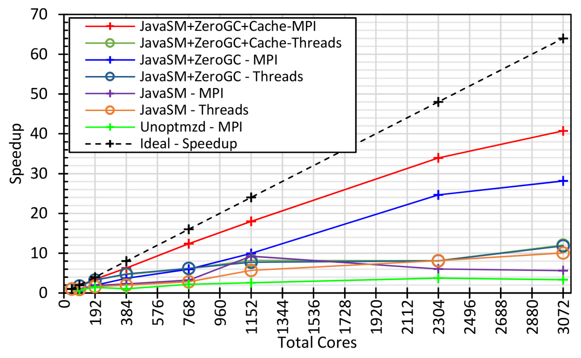

Figure 12. DA-MDS speedup for 200K with different optimization techniques

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

6. RELATED WORK

Le Chai's PhD [3] work present similar content to our improvements, but done on MVAPICH2. It 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. First is to use a user level shared memory map similar to what we do in SPIDAL. 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) [26] presents a similar idea to the zero intra-node messaging in SPIDAL. It implements a custom memory allocation layer that enables MPI ranks running within a node to have a shared heap space and 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 [17]. 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 [5] library.

NCCL (pronounced "Nickel") [19] is a recent attempt 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 involve communication with the host and NCCL 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 databricks company is a series of optimization techniques targeting Apache Spark [27] to bring its performance closer to native level. It includes several concepts similar to our work on SPIDAL such as off-heap data structures, in memory data transfer, and cache aware computing.

7. CONCLUSION

Our results show that the SPIDAL applications scale and perform well in large HPC clusters. Also, with our optimizations we could execute SPIDAL applications on much larger data sets than what we could in the past and still achieve excellent scaling results. Our improved intra-node communication is pivotal to the gains we have made and with the developer friendly Java interface, we believe SPIDAL applications will help us in the future to work with other big data data platforms such as Apache Hadoop, Spark, and Storm.

ACKNOWLEDGMENTS

This work was partially supported by NSF CIF21 DIBBS 1443054 and AFOSR FA9550-13-1-0225 awards. We also like to express our gratitude to the staff and system administrators at Digital Science Center (DSC) at Indiana University for supporting us on this work.

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 Enon, R. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE 5*, 1 (1998), 46–55.
5. Dagum, L., and Menon, R. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng. 5*, 1 (Jan. 1998), 46–55.
6. Ekanayake, J. *Architecture and Performance of Runtime Environments for Data Intensive Scalable Computing*. PhD thesis, Indianapolis, IN, USA, 2010. AAI3439561.
7. Fox, G. Robust scalable visualized clustering in vector and non vector semi-metric spaces. *Parallel Processing Letters 23*, 2 (2013).
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. Geoffrey C. Fox, S. J. J. Q. S. E., and Luckow, A. Towards a comprehensive set of big data benchmarks. Tech. rep., 2015.

12. 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.

13. 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.

14. OpenHFT Java Thread Affinity. `https://github.com/OpenHFT/Java-Thread-Affinity`.

15. 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.

16. 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.

17. 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.

18. The Million Sequence Clustering Project. `http://salsahpc.indiana.edu/millionseq/`.

19. NVIDIA NCCL. `https://github.com/NVIDIA/nccl`.

20. OpenHFT JavaLang Project. `https://github.com/OpenHFT/Java-Lang`.

21. OSU Micro-Benchmarks. `http://mvapich.cse.ohio-state.edu/benchmarks/`.

22. PlotViz on Web. `https://github.com/DSC-SPIDAL/WebPViz`.

23. Rose, K., Gurewwitz, E., and Fox, G. A deterministic annealing approach to clustering. *Pattern Recogn. Lett. 11*, 9 (Sept. 1990), 589–594.

24. 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.

25. 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.

26. 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.

27. 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.