Evaluation of Java Message Passing in High Performance Data Analytics

Saliya Ekanayake, Geoffrey Fox
School of Informatics and Computing
Indiana University
Bloomington, Indiana, USA
{sekanaya, gcf}@indiana.edu

***Abstract*—In the last few years, Java gain popularity in processing "big data" mostly with Apache big data stack – a collection of open source frameworks dealing with abundant data, which includes several popular systems such as Hadoop, Hadoop Distributed File System (HDFS), and Spark. Efforts have been made to introduce Java to High Performance Computing (HPC) as well in the past, but were not embraced by the community due to performance concerns. However, with continuous improvements in Java performance an increasing interest has been placed on Java message passing support in HPC. We support this idea and show its feasibility in solving real world data analytics problems.**

***Index Terms*— Message passing, Java, HPC**

I. Introduction

MESSAGE passing has been the *de-facto* model in realizing distributed memory parallelism [1] where Message Passing Interface (MPI) with its implementations such as MPICH2 [2] and OpenMPI [3] have been successful in producing high performance applications [4]. We use message passing with threads [5] in our data analytics applications on Windows High Performance Computing (HPC) environments [6] using MPI.NET [7] and Microsoft's Task Parallel Library (TPL) [8]. However, the number of available Windows HPC systems are limited and our attempts to run these on traditional Linux based HPC clusters using Mono – a cross platform .NET development framework – have been unsuccessful due to poor performance. Therefore, we decided to migrate our applications to Java for reasons 1) productivity offered by Java and its ecosystem; and 2) emerging success of Java in HPC [9]. In this paper, we present our experience in evaluating the performance of our parallel deterministic annealing clustering programs – vector sponge (DAVS) [10] and pairwise clustering (PWC) [11]. We also present micro benchmark results for two Java message passing frameworks – OpenMPI with Java binding and FastMPJ [12] – compared against native OpenMPI and MPI.NET. The results show clear improvement of application performance over C# with MPI.NET and near native performance in micro benchmarks.

II. Related Work

Message passing support for Java can be classified as pure Java implementations or Java bindings for existing native MPI libraries (i.e. wrapper implementations). Pure Java implementations advocate portability, but may not be as efficient as Java bindings that call native MPI (see III.C and [12]). There are two proposed Application Programming Interfaces (API) for Java message passing – mpiJava 1.2 [13] and Java Grande Forum (JGF) Message Passing interface for Java (MPJ) [14]. However, there are implementations that follow custom API as well [9].

Performance of Java MPI support has been studied with different implementations and recently in [12, 15]. The focus of these studies is to evaluate MPI kernel operations and little or no information given on applying Java MPI to scientific applications. However, there is an increasing interest [16] on using MPI with large scale data analytics frameworks such as Apache Hadoop [17]. MR+ [18] is a framework with similar intent, which allows Hadoop MapReduce [19] programs to run on any cluster under any resource manager while providing capabilities of MPI as well.

III. Technical Evaluation

The DAVS code is about 15k lines and PWC is about 6k lines of C# code. We used a combination of commercially available code converter [20] and carefully inspected manual rewrites to port the C# code to Java. Furthermore, we performed a series of serial and parallel tests to confirm correctness is preserved during the migration, prior to evaluating performance.

Our interest in this experiment was to compare application performance when run on Linux based HPC clusters against results on Windows HPC environments. We noticed from initial runs that two of the MPI operations – allreduce, and send and receive – contribute to the most of inter-process communication. Therefore, we extended the evaluation by performing micro benchmarks for these, which further supported our choice to use Java.

```
Input:  maxMsgSize // maximum message size in bytes

$msgSizeL = 8192$ // messages to be considered large
$numItr = 1000$ // iterations for small messages
$numItrL = 100$ //iterations for large messages
$skip = 200$ // skip this many for small messages
$skipL = 10$ // skip this many for large messages

$comm$ = MPI_COMM_WORLD
$me$ = MPI_Comm_rank (comm)
$size$ = MPI_Comm_size (comm)

$sbuff[maxMsgSize/4]$ // float array –initialized to 1.0
$rbuff[maxMsgSize/4]$ // float array –initialized to 0.0

For $i = 1$ to $i \leq maxMsgSize/4$
    If $i > msgSizeL$
        $skip = skipL$
        $numItr = numItrL$
    MPI_Barrier ($comm$)
    $duration = 0.0$
    For $j = 0$ to $j < numItr + skip$
        $t$ = MPI_Wtime ( )
        MPI_Allreduce ($sbuff$,$rbuff$,$i$
                                MPI_SUM,$comm$)
        If $j \geq skip$
            $duration$+=MPI_Wtime ( ) $-$ $t$
        MPI_Barrier ($comm$)
        $j = j + 1$
    $latency = duration/numItr$
    $avgTime$ = MPI_Reduce ($latency$, MPI_SUM,0)
    If $me == 0$
        Print ($i, avgTime$)
    $i = i * 2$
    MPI_Barrier ($comm$)
```

Fig. 1 Pseudo code for allreduce benchmark

A. *Computer Systems*

We used two Indiana University clusters, Madrid and Tempest, and one FutureGrid [21] cluster – India, as described below.

Tempest: 32 nodes, each has 4 Intel Xeon E7450 CPUs at 2.40GHz with 6 cores, totaling 24 cores per node; 48 GB node memory and 20Gbps Infiniband (IB) network connection. It runs Windows Server 2008 R2 HPC Edition – version 6.1 (Build 7601: Service Pack 1).

Madrid: 8 nodes, each has 4 AMD Opteron 8356 at 2.30GHz with 4 cores, totaling 16 cores per node; 16GB node memory and 1Gbps Ethernet network connection. It runs Red Hat Enterprise Linux Server release 6.5

FutureGrid (India): 128 nodes, each has 2 Intel Xeon X5550 CPUs at 2.66GHz with 4 cores, totaling 8 cores per node; 24GB node memory and 20Gbps IB network connection. It runs Red Hat Enterprise Linux Server release 5.10.

B. *Software Environments*

We used .NET 4.0 runtime and MPI.NET 1.0.0 for C# based tests. DAVS Java version uses a novel parallel tasks library called Habanero Java (HJ) library from Rice University [22, 23], which requires Java 8. Therefore, we used an early access (EA) release – build 1.8.0-ea-b118. Early access releases may not be well optimized, but doing a quick test with threading switched off we could confirm DAVS performs equally well on stable Java 7 and Java 8 EA.

There have been several message passing frameworks for Java [24], but due to the lack of support for IB network and to other drawbacks discussed in [12], we decided to evaluate OpenMPI with its Java binding and FastMPJ, which is a pure Java implementation of mpiJava 1.2 [13] specification. OpenMPI's Java binding [25] is an adaptation from the original mpiJava library [26]. However, OpenMPI community has recently introduced major changes to its API, and internals, especially removing MPI.OBJECT type and adding support for direct buffers in Java. These changes happened while we were evaluating DAVS, thus we tested OpenMPI Java binding in one of its original (nightly snapshot version 1.9a1r28881) and updated forms (source tree revision 30301). PWC was tested with an even newer version – 1.7.5.We will refer to these as OMPI-nightly, OMPI-trunk, and OMPI-175 for simplicity.

C. *MPI Micro Benchmarks*

We based our experiments on Ohio MicroBenchmark (OMB) suite [27], which is intended to test native MPI implementations' performance. Therefore, we implemented the selected allreduce, and send and receive tests in all three Java MPI flavors and MPI.NET, in order to test Java and C# MPI implementations.

Fig. 1 presents the pseudo code for OMB allreduce test. Note the syntax of MPI operations do not adhere to a particular language and the number of actual parameters are cut short for clarity. Also, depending on the language and MPI framework used, the implementation details such as data structures used and buffer allocation were different from one another.

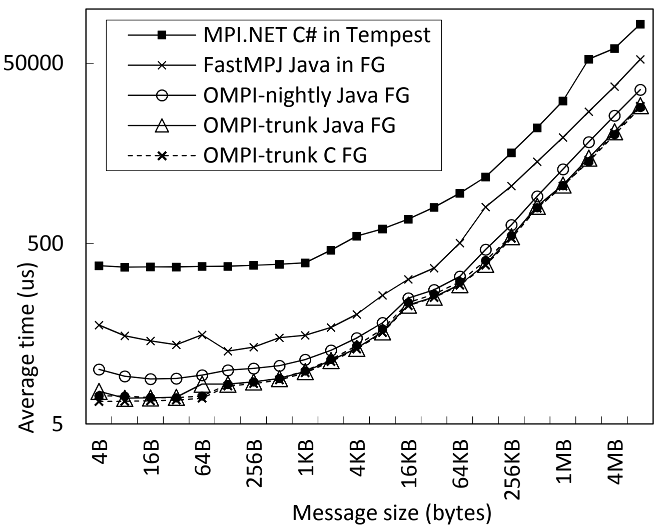

Fig. 2. Performance of MPI allreduce operation

Fig. 2 shows the results of allreduce benchmark for different MPI implementations with message size ranging from 4 bytes (B) to 8 megabytes (MB). These are averaged values over patterns 1x1x8, 1x2x8, and 1x4x8 where pattern format is number of threads per process x number of processes per node x number of nodes (i.e. TxPxN). The best performance came with C versions of OpenMPI, but interestingly OMPI-trunk Java performance overlaps on these indicating its near zero overhead. The older, OMPI-nightly Java performance is near as well, but shows more overhead than its successor. FastMPJ performance is better than MPI.NET, but slower than OpenMPI versions. The slowest performance came with MPI.NET, which may be improved with further tuning, but as our focus was to evaluate Java versions we did not proceed in this direction.

We experienced a similar pattern with MPI send and receive benchmark (*Fig. 3*) as shown in *Fig. 4*.

```
Input:   maxMsgSize // maximum message size in bytes

$msgSizeL = 8192$ // messages to be considered large
$numItr = 1000$ // iterations for small messages
$numItrL = 100$ //iterations for large messages
$skip = 200$ // skip this many for small messages
$skipL = 10$ // skip this many for large messages
$comm$ = MPI_COMM_WORLD
$me$ = MPI_Comm_rank (comm)
$size$ = MPI_Comm_size (comm)

$sbuff[maxMsgSize]$ // byte array –initialized to 1.0
$rbuff[maxMsgSize]$ // byte array –initialized to 0.0

For $i = 0$ to $i \leq maxMsgSize$
    If $i > msgSizeL$
        $skip = skipL$
        $numItr = numItrL$
    MPI_Barrier ($comm$)
    $duration = 0.0$
    If $me == 0$
        For $j = 0$ to $j < numItr + skip$
            If $j == skip$
                $t$ = MPI_Wtime ( )
            MPI_Send ($sbuff$, $i$,1,1)
            MPI_Recv (r$buff$, $i$,1,1)
        $duration$+=MPI_Wtime ( ) $- t$
    Else If $me == 1$
        For $j = 0$ to $j < numItr + skip$
            MPI_Recv ($rbuff$, $i$,0,1)
            MPI_Send sr$buff$, $i$,0,1)
    If $me == 0$
     $latency = duration/2 * numItr$
        Print ($i, latency$)
     $i = i == 0\ ?\ 1 :\ i * 2$
    MPI_Barrier ($comm$)
```

Fig. 3 Pseudo code for send and receive benchmark

Note message sizes were from 0B to 1MB for this test, where 0B time corresponds to average latency.

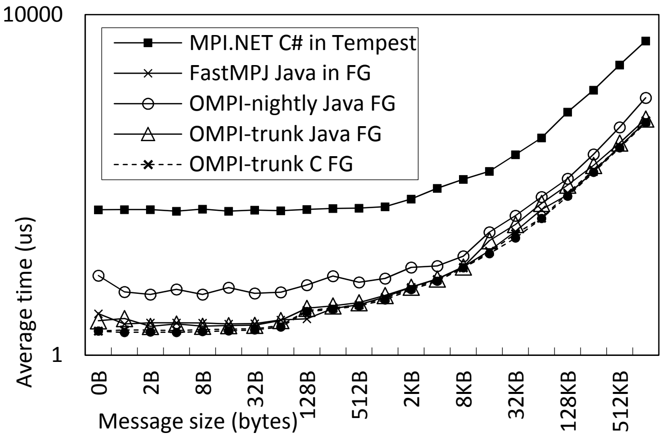

Fig. 4 Performance of MPI send and receive operations

We performed tests in *Fig. 2* and Fig. 4 on Tempest and FutreGrid-India as these were the two clusters with Infiniband interconnect. However, after deciding OMPI-trunk as the best performer, we conducted these two benchmarks on Madrid as well to compare performance against usual Ethernet connection. Fig. 5 and Fig. 6 show the clear advantage of using Infiniband with performance well above 10 times for smaller messages and around 5-8 times for larger messages compared to Ethernet.

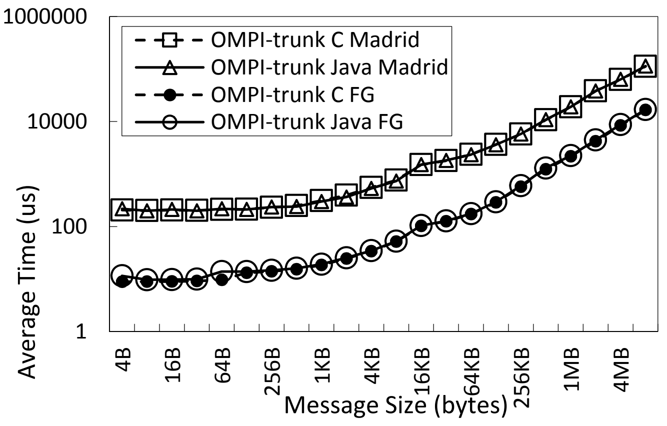

Fig. 5 Performance of MPI allreduce on Infiniband and Ethernet

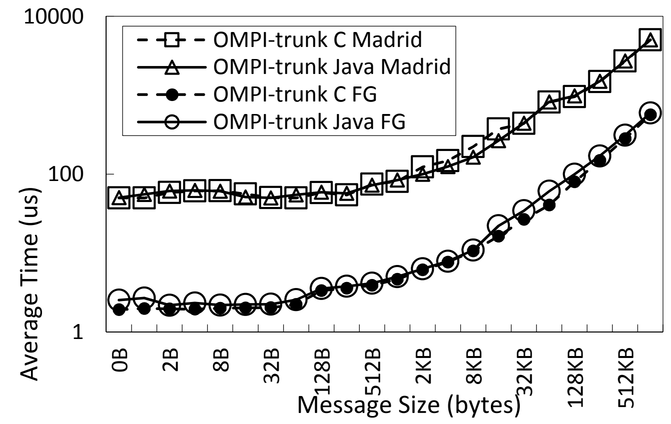

Fig. 6 Performance of MPI send and receive on Infiniband and Ethernet

D. DAVS Performance with MPI

We decided to apply DAVS to the same “peak-matching” problem [10] that its C# variant used to solve, so we could verify accuracy and compare performance. We performed clustering of the LC-MS data [10] in two modes – Charge2 and Charge5 – where the former processed 241605 points and found on average 24.5k clusters. Charge5 mode handled 16747 points producing an average of 28k clusters.

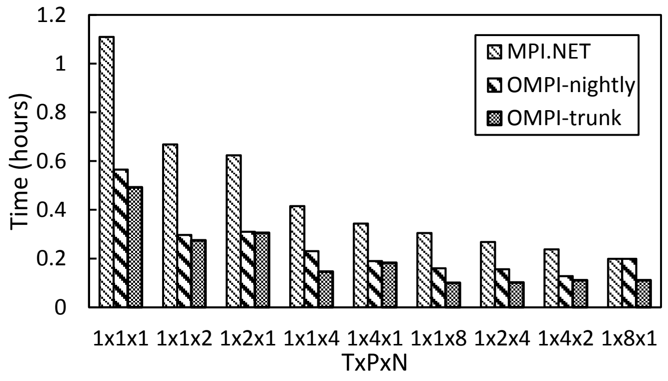

Fig. 7 DAVS Charge5 performance

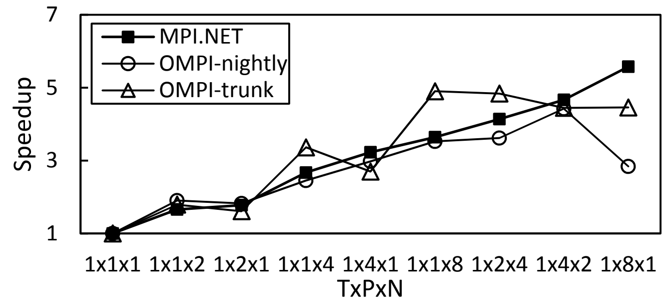

Fig. 8 DAVS Charge5 speedup

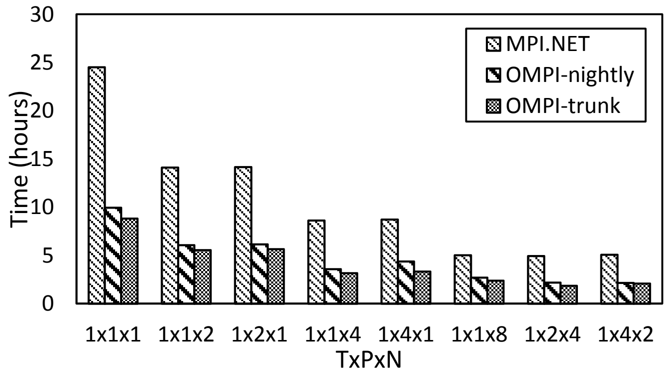

Fig. 9 DAVS Charge2 performance

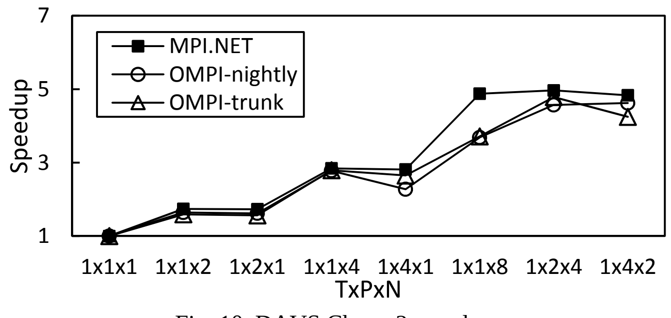

Fig. 10 DAVS Charge2 speedup

These modes exercises different execution flows in DAVS where Charge2 is more intense in both computation and communication than Charge5. DAVS supports threading too which, we have studied separately in section E. Also note tests based on OMPI Java versions were done on FutureGrid-India and MPI.NET C# on Tempest in following figures.

Fig. 7 and Fig. 8 show Charge5 performance and speedup under different MPI libraries. OMPI-trunk happens to give the best performance and it achieves nearly double the performance in all cases compared to MPI.NET. Charge2 performance and speedup show similar results as given in Fig. 9 and Fig. 10. Note we could not test pattern 1x8x1 due to insufficient memory.

E. DAVS Performance with Threads

DAVS has the option to utilize threads for its pleasingly parallel shared memory computations. These code segments follow a fork-join style $forall$ loops, which are well supported in HJ library. It is worth noting that DAVS semantics for parallelism with threads and message passing are different such that they complement rather replace one another. Therefore, comparing performance of N-way MPI processes versus N-way threads does not make sense. Instead, what is studied in this paper is the performance variance with threads for different number of MPI processes.

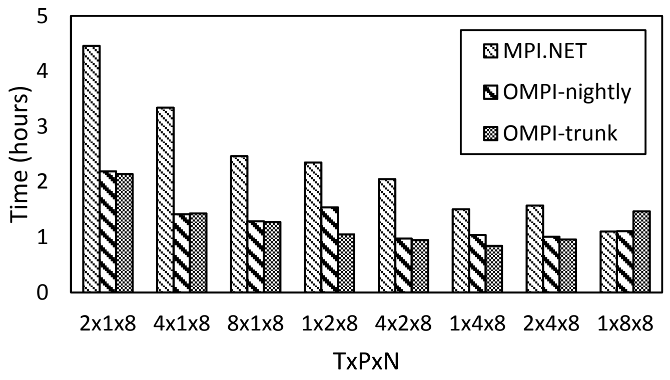

Fig. 11 DAVS Charge2 performance with threads

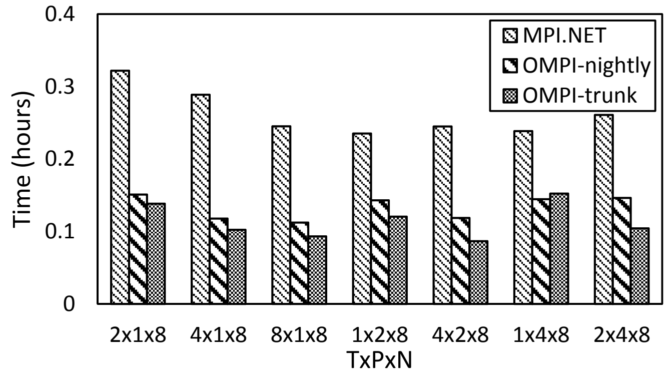

Fig. 12 DAVS Charge5 performance with threads

Fig. 11 and Fig. 12 show the performance of DAVS application with threads in its Charge2 and Charge5 modes respectively. We encountered intermittent errors with OMPI-trunk when running 8 processes per node and are currently working for a resolution with help from OpenMPI community. Overall OMPI-nightly and OMPI-trunk versions show similar performance and are about two times better than MPI.NET.

OpenMPI is continuously improving its process binding support, which affects the behavior of threads. Therefore, we still need to perform additional testing to fully comprehend the behavior with threads.

F. *DAVS Single Node Performance*

While it is important to understand behavior with MPI and threads on multiple nodes, it is essential to study the single node performance of DAVS in different environments as a baseline.

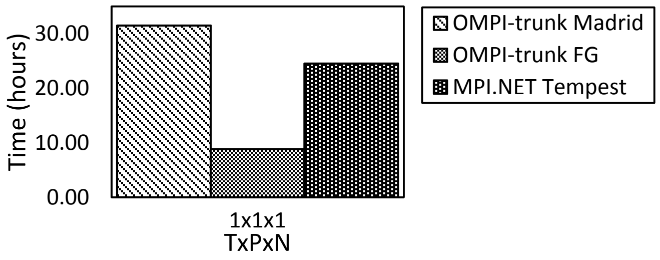

Fig. 13 DAVS Charge2 performance on single node

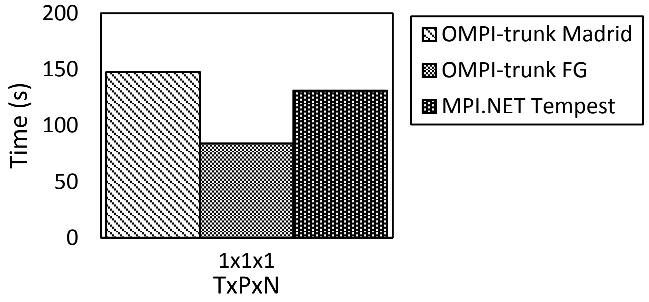

Fig. 14 DAVS Charge6 performance on single node

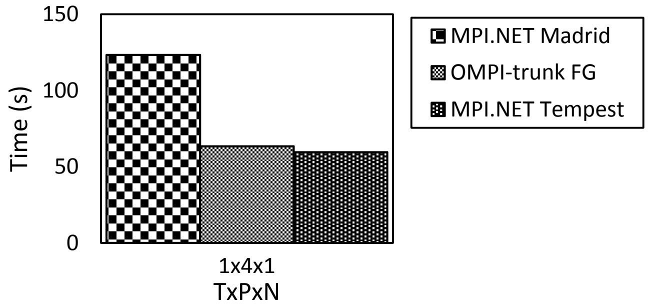

Fig. 15 DAVS Charge6 performance on single node with multiple processes

Fig. 13 shows Charge2 serial performance on Madrid, FutureGrid-India, and Tempest. Note, we have shown performance with OMPI-trunk only for Java cases as this was the best performer and 1x1x1 does not use any MPI. Fig. 14 shows similar performance for a different mode – Charge6 – in DAVS, again run as 1x1x1. Madrid used to be a Windows HPC cluster and we had results from an earlier run for Charge6, which tested performance of 1x4x1. We compare that with other clusters and MPI frameworks in Fig. 15.

G. *PWC Performance with MPI and Threads*

Pairwise clustering is a heavily used program in our work of analyzing gene sequences [28]. It supports both distributed and shared memory parallelism where unlike in DAVS the program logic is same between these modes in PWC. This makes it possible to compare MPI and thread performance in a single chart.

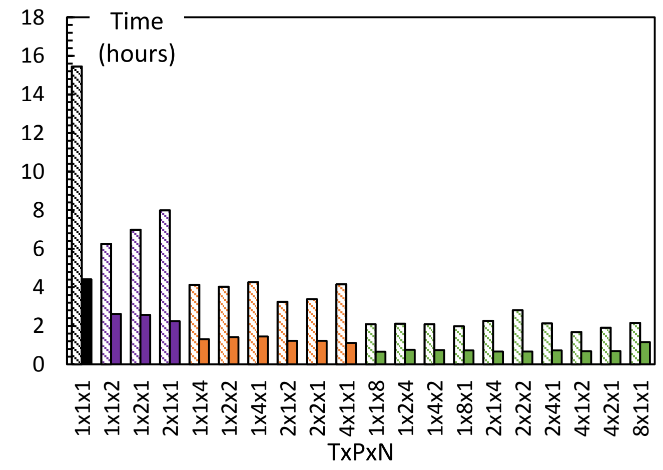

Fig. 16 PWC performance - parallelism 1 to 8

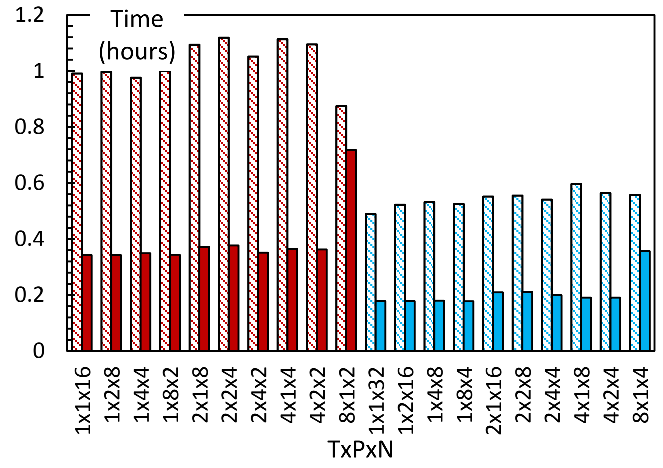

Fig. 17 PWC performance - parallelism 16 to 32

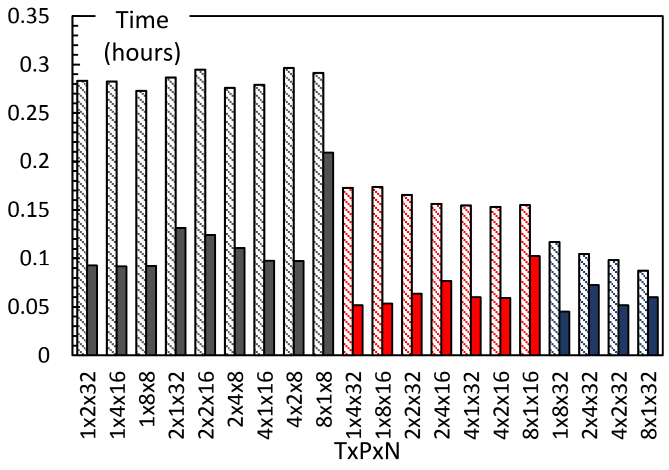

Fig. 18 PWC performance - parallelism 64 to 256

Fig. 16 to Fig. 18 present performance of PWC with varying parallelism. The results are presented for OMPI-175 as it was the best among different MPI flavors and MPI.NET. The different colors indicate groups with same parallelism. The diagonal patterned patterns bars represent MPI.NET while solid colored bars represent OMPI-175. Note 8x1xN pattern has unusually high computing times in each group for OMPI-175, which is due to inefficient binding of processes to cores. FutureGrid has two quad core sockets and it is inefficient to let thread run across CPU boundaries, which is what happens when a single process with 8 threads is bound to multiple CPUs in this case.

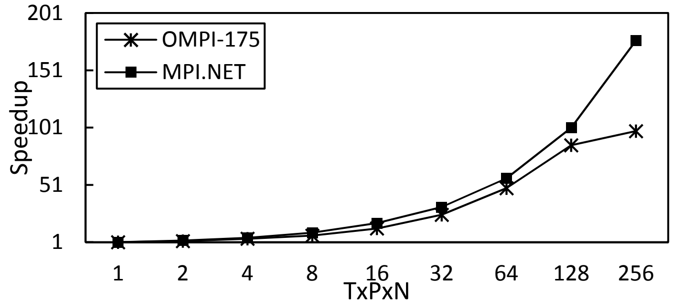

Fig. 19 PWC speedup

Fig. 19 shows speedup of different implementations compared to their 1x1x1 case. Even though the speedups show similar curves the Java based implementation shows a 2X speedup over C# based PWC on average.

PWC has an intensive Eigen solver code, which uses point to point communication. Therefore, we timed this separately as a benchmark of MPI send-receive operation.

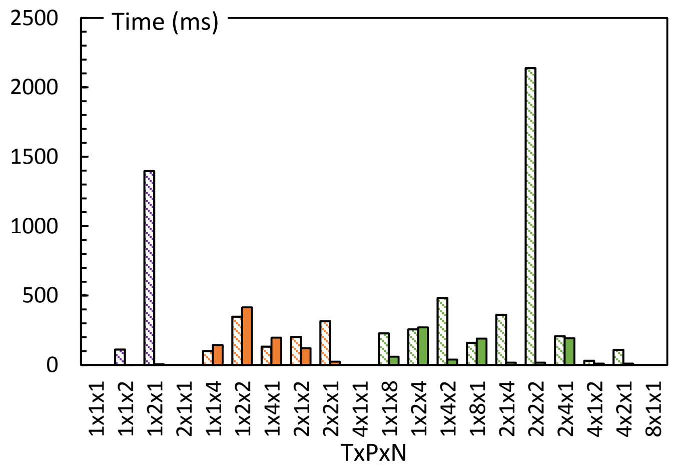

Fig. 20 PWC Eigen solver MPI send-receive time - parallelism 1 to 8

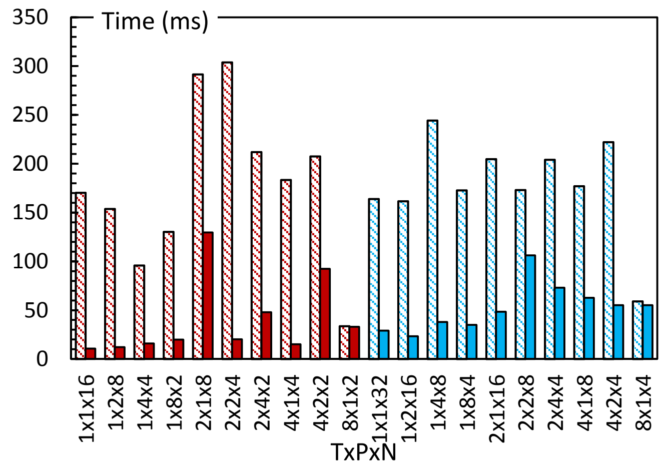

Fig. 21 PWC Eigen solver MPI send-receive time - parallelism 16 to 32

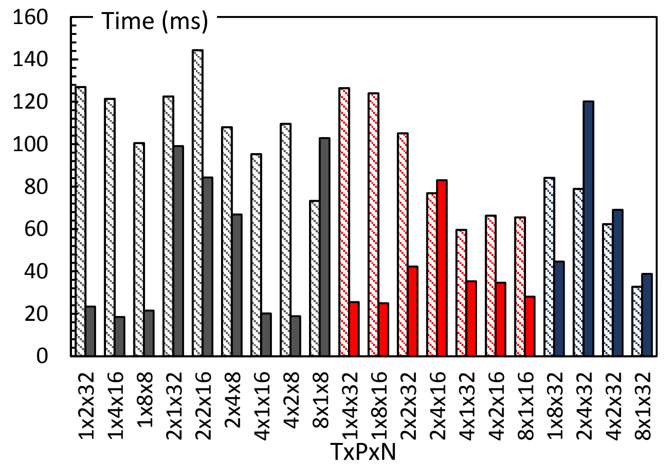

Fig. 22 PWC Eigen solver MPI send-receive time - parallelism 64 to 256

Fig. 20 to Fig. 22 show the MPI send-receive timing of the Eigen solver in PWC. Note zero times corresponds to patterns where only threading is used. In fact, the send-receive time is governed by the PxN factor, we provide in Fig. 23 average times for each PxN group.

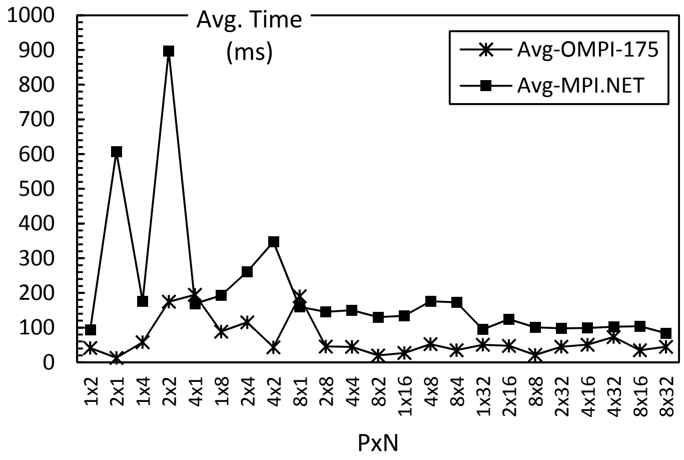

Fig. 23 Average MPI send-receive time for different PxN groups

Values from Eigen solver timing shows on average a 3X improvement with OMPI-175 over MPI.NET, however, as visible from Fig. 23 the difference tends to be small with increasing inter-node communication.

SUMMARY

Scientific applications written in C, C++, and Fortran have embraced MPI since its inception and various attempts have been made over the years to establish this relationship for applications written in Java. However, only few implementations such as OpenMPI and FastMPJ are in active development with support for fast interconnect systems. OpenMPI in particular has recently introduced improvements to its Java binding to close the gap between Java and native performance.

We performed MPI kernel benchmarks based on OMB algorithms and the results depicts latest OpenMPI Java binding as the best among selected Java MPI implementations. Also, they confirmed that OpenMPI Java performance on both collective and point to point operations is virtually identical to its native C counterpart.

Our aim of this effort has been to migrate existing C# based code to Java in hope of running on traditional HPC clusters while utilizing the rich programming environment of Java. The initial runs of DAVS and PWC show promising performance and speedup with over 2X improvement over C# version. We also noticed from PWC that MPI.NET and OpenMPI tends to perform similar in point to point operations as number of inter-node communications increases.

The experiments also exercised shared memory parallelism with a novel Java thread library – HJ. The results show decent performance, yet there is room for improvement. Also, we learnt it is essential to load balance MPI processes through MPI process mapping and binding to get the best performance in the presence of threads.

In conclusion, moving code to Java eco system has given us the advantage to utilize traditional Linux based HPC systems while achieving higher performance than with C#. It will also pave the way for integrating these algorithms with other big data analytic frameworks.

Acknowledgement

We like to express our sincere gratitude to Prof. Vivek Sarkar and his team at Rice University for giving us access and continuous support for HJ library, which made it possible to incorporate threads in DAVS and PWC. We are equally thankful to Prof. Guillermo López Taboada for giving us free unrestricted access to commercially available FastMPJ MPI library. We are also thankful to FutureGrid project and its support team for their support with HPC systems.

References

[1] LUSK, E. and YELICK, K. LANGUAGES FOR HIGH-PRODUCTIVITY COMPUTING: THE DARPA HPCS LANGUAGE PROJECT. *Parallel Processing Letters*, 17, 01 2007), 89-102.
[2] Laboratory, A. N. *MPICH2*,. City.
[3] Gabriel, E., Fagg, G., Bosilca, G., Angskun, T., Dongarra, J., Squyres, J., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R., Daniel, D., Graham, R. and Woodall, T. *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation*. Springer Berlin Heidelberg, City, 2004.
[4] Gropp, W. Learning from the Success of MPI. In *Proceedings of the Proceedings of the 8th International Conference on High Performance Computing* (2001). Springer-Verlag, [insert City of Publication],[insert 2001 of Publication].
[5] Ekanayake, S. *Survey on High Productivity Computing Systems (HPCS) Languages*. Pervasive Technology Institute, Indiana University, Bloomington, 2013.
[6] Qiu, J., Beason, S., Bae, S.-H., Ekanayake, S. and Fox, G. Performance of Windows Multicore Systems on Threading and MPI. In *Proceedings of the Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing* (2010). IEEE Computer Society, [insert City of Publication],[insert 2010 of Publication].
[7] Gregor, D. and Lumsdaine, A. Design and implementation of a high-performance MPI for C\# and the common language infrastructure. In *Proceedings of the Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (Salt Lake City, UT, USA, 2008). ACM, [insert City of Publication],[insert 2008 of Publication].
[8] Daan Leijen, J. H. *Optimize Managed Code For Multi-Core Machines*. City.
[9] Taboada, G. L., Touri, J., #241, Ram, #243 and Doallo, n. Java for high performance computing: assessment of current research and practice. In *Proceedings of the Proceedings of the 7th International Conference on Principles and Practice of Programming in Java* (Calgary, Alberta, Canada, 2009). ACM, [insert City of Publication],[insert 2009 of Publication].
[10] Fox, G., Mani, D. R. and Pyne, S. *Parallel deterministic annealing clustering and its application to LC-MS data analysis*. IEEE, City, 2013.
[11] Fox, G. C. Deterministic annealing and robust scalable data mining for the data deluge. In *Proceedings of the Proceedings of the 2nd international workshop on Petascal data analytics: challenges and opportunities* (Seattle, Washington, USA, 2011). ACM, [insert City of Publication],[insert 2011 of Publication].
[12] Expósito, R., Ramos, S., Taboada, G., Touriño, J. and Doallo, R. FastMPJ: a scalable and efficient Java message-passing library. *Cluster Computing*(2014/02/06 2014), 1-20.
[13] Bryan Carpenter, G. F., Sung-Hoon Ko and Sang Lim. *mpiJava 1.2: API Specification*. 1999.
[14] Carpenter, B., Getov, V., Judd, G., Skjellum, A. and Fox, G. MPJ: MPI-like message passing for Java. *Concurrency: Practice and Experience*, 12, 11 2000), 1019-1038.
[15] Taboada, G. L., Tourino, J. and Doallo, R. *Performance analysis of Java message-passing libraries on fast Ethernet, Myrinet and SCI clusters*. City, 2003.
[16] Squyres, J. *Resurrecting MPI and Java*. City, 2012.
[17] Tom White *Hadoop: The Definitive Guide*. Yahoo Press; Second Edition edition, 2010.
[18] Ralph H. Castain, W. T. *MR+ A Technical Overview*. 2012.
[19] Dean, J. and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. *Sixth Symposium on Operating Systems Design and Implementation*2004), 137-150.
[20] Inc., T. S. S. *C# to Java Converter*. City.
[21] von Laszewski, G., Fox, G. C., Fugang, W., Younge, A. J., Kulshrestha, A., Pike, G. G., Smith, W., Vo, x, ckler, J., Figueiredo, R. J., Fortes, J. and Keahey, K. *Design of the FutureGrid experiment management framework*. City, 2010.
[22] Cav, V., #233, Zhao, J., Shirako, J. and Sarkar, V. Habanero-Java: the new adventures of old X10. In *Proceedings of the Proceedings of the 9th International Conference on Principles and Practice of Programming in Java* (Kongens Lyngby, Denmark, 2011). ACM, [insert City of Publication],[insert 2011 of Publication].
[23] Sarkar, V. a. I., Shams Mahmood *HJ Library*. City.
[24] Taboada, G. L., Ramos, S., Exp, R. R., #243, sito, Touri, J., #241, Ram, #243 and Doallo, n. Java in the High Performance Computing arena: Research, practice and experience. *Sci. Comput. Program.*, 78, 5 2013), 425-444.
[25] Project, T. O. M. *FAQ: Where did the Java interface come from?* , City.
[26] Baker, M., Carpenter, B., Fox, G., Hoon Ko, S. and Lim, S. *mpiJava: An object-oriented java interface to MPI*. Springer Berlin Heidelberg, City, 1999.
[27] Laboratory, T. O. S. U. s. N.-B. C. and (NBCL) *OMB (OSU Micro-Benchmarks)*. City.
[28] Ruan, Y., Ekanayake, S., Rho, M., Tang, H., Bae, S.-H., Qiu, J. and Fox, G. DACIDR: deterministic annealed clustering with interpolative dimension reduction using a large collection of 16S rRNA sequences. In *Proceedings of the Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine* (Orlando, Florida, 2012). ACM, [insert City of Publication],[insert 2012 of Publication].