Enabling High Performance Computing for Semantic Web Applications by Means of Open MPI Java Bindings

Alexey Cheptsov
High Performance Computing Center Stuttgart (HLRS)
University of Stuttgart
70550 Stuttgart, Germany
Email: cheptsov@hlrs.de

***Abstract*—The volume of data available in the Semantic Web has already reached the order of magnitude of billions of triples and is expected to further grow in the future. The availability of such an amount of data makes it attractive for Semantic Web applications to exploit High Performance Computing (HPC) infrastructures to effectively process such data. Unfortunately, most Semantic Web applications are written in the Java programming language, whereas current frameworks that make the most out of HPC infrastructures, such as the Message Passing Interface (MPI), only target C or Fortran applications. Attempts to port existing parallelization frameworks to the Java language prove to be very inefficient in terms of the performance benefits for applications. This paper presents an efficient porting based on the Open MPI framework.**

Keywords-High Performance Computing; Semantic Web; Data-Centric Computing; Performance; Scalability; Message-Passing Interface.

I. Introduction

The volume of data collected on the Semantic Web has already reached the order of magnitude of billions of triples and is expected to further grow in the future, which positions this Web extension to dominate the data-centric computing in the oncoming decade. Processing (e.g., inferring) such volume of data, such as generated in the social networks like Facebook or Twitter, or collected in domain-oriented knowledge bases like pharmacological data integration platform OpenPHACTS [1], is thus of a big challenge. Whereas there is a number of existing highly-scalable software solutions for storing data, such as Jena [2], the scalable data processing constitutes the major challenge for data-centric applications. The group of issues related to scaling the existing data processing techniques to the available volumes is often referred as the "Big Data" problem. Among those data-centric communities that address the Big Data, the Semantic Web enjoys a prominent position.

Semantic Data are massively produced and published at the speed that makes traditional processing techniques (such as reasoning) inefficient when applied to the real-scale data. The data scaling problem in the Semantic Web is considered in two its main aspects - horizontal and vertical scale. Horizontal scaling means dealing with diverse, and often unstructured data acquired from heterogeneous sources. The famous LOD cloud diagram [3] consists of hundreds of diverse data sources, ranging from geospatial cartographic sources to governmental data, opened to the publicity, like Open Government Data [4]. Vertical scaling implies scaling up the size of similarly structured data. Along the open government data spawns over 851,000 data sets across 153 catalogues from more than 30 countries, as estimated in [5] at the beginning of 2012. Processing data in such an amount is not straightforward and challenging for any of the currently existing frameworks and infrastructures. Whereas there are some known algorithms dealing with the horizontal scaling complexity, such as identification of the information subsets related to a specific problem, i.e., subsetting, the vertical scaling remains the major challenge for all existing algorithms. Another essential property of the Big Data is the complexity. Semantic applications must deal with rich ontological models describing complex domain knowledge, and at the same time highly dynamic data representing recent or relevant information, as produced by streaming or search-enabled data sources. A considerable part of the web data is produced as a result of automatic reasoning over streaming information from sensors, social networks, and other sources, which are highly unstructured, inconsistent, noisy and incomplete.

The availability of such an amount of complex data makes it attractive for Semantic Web applications to exploit High Performance Computing (HPC) infrastructures to effectively process Big Data. As a reaction on this challenge, a number of major software vendors in the Semantic Web domain have been collaborating with high performance computing centers, and this trend is expected to grow in the nearest future [6]. Both commodity and more dedicated HPC architectures, such as the Cray XMT [7], have been in focus of the data-intensive Web applications. The XMT dedicated system, however, proved successful only for a limited number of tasks so far, which is mainly due to the complexity of exploiting the offered software frameworks (mainly non-standard pragma-based C extensions. Unfortunately, most Semantic Web applications are written in the Java programming language, whereas current frameworks that make the most out of HPC infrastructures, such as the Message Passing Interface (MPI), only target C or Fortran applications. MPI is a process-based parallelization strategy, which is a de-facto standard in the area of parallel computing for C, C++, and Fortran applications. Known alternative parallelization frameworks to MPI that conform with Java, such as Hadoop [8] or Ibis [9], prove to be scalable though but are not even nearly as efficient or well-developed as numerous open-source implementations of MPI, such as MPICH or Open MPI [10]. We look at how to resolve the above-mentioned issues in a way that leverages the advances of the existing MPI frameworks.

The remainder of the paper is organized as follows. Section 2 gives an overview of the related work. Section 3 discusses the data-centric parallelization model based on MPI. Section 4 introduces our implementation of Java bindings for Open MPI. Section 5 gives examples of successful pilot scenarios implemented with our solution and discuss future work in terms of the development, implementation, and standardization activities.

II. Related Work

There are only a few alternatives to MPI in introducing the large-scale parallelism to Java applications. The most promising among those alternatives in terms of the performance and usability are solutions offered by IBIS/JavaGAT and MapReduce/Hadoop.

IBIS [11] is a middleware stack used for running Java applications in distributed and heterogeneous computing environments. IBIS leverages the peer-to-peer communication technology by means of the proprietary Java RMI (Remote Memory Invocation) implementation, based on GAT (Grid Application Toolkit) [12]. The Java realization of GAT (JavaGAT) is a middleware stack that allows the Java application to instatiate its classes remotely on the network-connected resource, i.e., a remote Java Virtual Machine. Along with the traditional access protocols. e.g., telnet or ssh, the advanced access protocols, such as ssh-pbs for clusters with PBS (cluster Portable Batch System)-like job scheduling or gsissh for grid infrastructures are supported. IBIS implements a mechanism of multiple fork-joins to detect and decompose the application's workload and execute its parts concurrently on distributed machines. While [9] indicates some successful Java applications implemented with IBIS/JavaGAT and shows a good performance, there is no clear evidence about the scalability of this solution for more complex communication patterns, involving nested loops or multiple split-joins. Whereas IBIS is a very effective solution for the distributed computing environments, e.g., Grid or Cloud, it is definitively not the best approach to be utilized on the tightly-coupled productional clusters.

MapReduce framework [8] and its most prominent implementation in Java, Hadoop, has got a tremendous popularity in modern data-intensive application scenarios. MapReduce

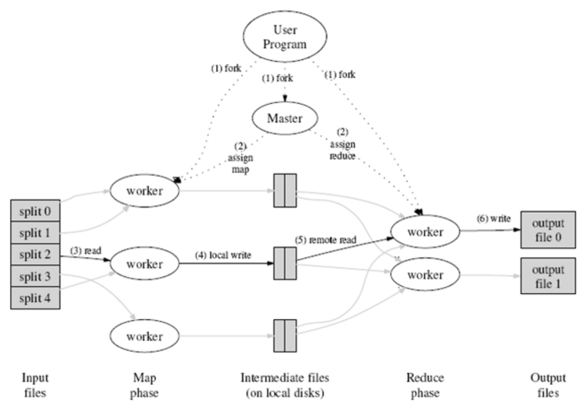


Figure 1. MapReduce processing schema

is a programming model for data-centric applications exploiting large-scale parallelism, originally introduced by Google in its search engine. In MapReduce, the application's workflow is divided into three main stages (see Figure 1): map, process, and reduce. In the map stage, the input data set is split into independent chunks and each of the chunks is assigned to independent tasks, which are then processed in a completely parallel manner (process stage). In the reduce stage, the output produced by every map task is collected, combined and the consolidated final output is then produced.

The Hadoop framework is a service-based implementation of MapReduce for Java. Hadoop considers a parallel system as a set of master and slave nodes, deploying on them services for scheduling tasks as jobs (Job Tracker), monitoring the jobs (Task Tracker), managing the input and output data (Data Node), re-executing the failed tasks, etc. This is done in a way that ensures a very high service reliability and fault tolerance properties of the parallel execution. In Hadoop, both the input and the output of the job are stored in a special distributed file-system. In order to improve the reliability, the file system also provides an automatic replication procedure, which however introduces an additional overhead to the inter-node communication. Due to this overhead, Hadoop provides much poorer performance than MPI, however offering better QoS characteristics related to the reliability and fault-tolerance. Since MPI and MapReduce paradigms have been designed to serve different purposes, it is hardly possible to comprehensively compare them. However they would obviously benefit from a cross-fertilization. As a possible scenario, MPI could serve a high-performance communication layer to Hadoop, which might help improve the performance by omitting the disk I/O usage for distributing the map and gathering the reduce tasks across the compute nodes.

III. Data-Centric Parallelization and MPI

By “data-centric parallelization” we mean a set of techniques for: (i) identification of non-overlapping application’s dataflow regions and corresponding to them instructions; (ii) partitioning the data into subsets; and (iii) parallel processing of those subsets on the resources of the high performance computing system. For Semantic Web applications utilizing the data in such well-established formats as RDF [13], parallelization relies mainly on partitioning (decomposing) the RDF data set on the level of statements (triples), see Figure 2a. The ontology data (also often referred as *tbox*) usually remains unpartitioned as its size is relatively small as compared with the actual data (*abox*), so that it is just replicated among all the compute nodes.

The Message-Passing Interface (MPI) is a process-based standard for parallel applications implementation. MPI processes are independent execution units that contain their own state information, use their own address spaces, and only interact with each other via interprocess communication mechanisms defined by MPI. Each MPI process can be executed on a dedicated compute node of the high performance architecture, i.e., without competing with the other processes in accessing the hardware, such as CPU and RAM, thus improving the application performance and achieving the algorithm speed-up. In case of the shared file system, such as Lustre [14], which is the most utilized file system standard of the modern HPC infrastructures, the MPI processes can effectively access the same file section in parallel without any considerable disk I/O bandwidth degradation. With regard to the data decomposition strategy presented in Figure 2a, each MPI process is responsible for processing the data partition assigned to it proportionally to the total number of the MPI processes (see Figure 2b). The position of any MPI process within the group of processes involved in the execution is identified by an integer R (rank) between 0 and N-1, where N is a total number of the launched MPI processes. The rank R is a unique integer identifier assigned incrementally and sequentially by the MPI runtime environment to every process. Both the MPI process’s rank and the total number of the MPI processes can be acquired from within the application by using MPI standard functions, such as presented in Listing 1. The typical data processing workflow with MPI can be depicted as shown in Figure 3. The MPI jobs are executed by means of the *mpirun* command, which is an important part of any MPI implementation. *mpirun* controls several aspect of parallel program execution, in particular launches MPI processes under the job scheduling manager software like OpenPBS [15]. The number of MPI processes to be started is provided with the *-np* parameter to *mpirun*. Normally, the number of MPI processes corresponds to the number of the compute nodes, reserved for the execution of parallel job. Once the MPI process is started, it can request its rank as well as the

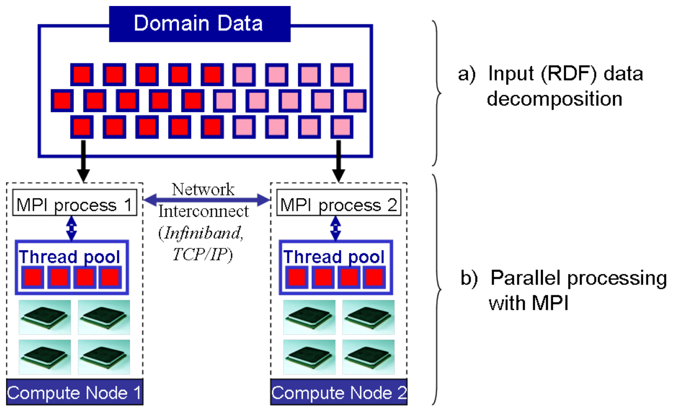

Domain Data
a) Input (RDF) data decomposition
MPI process 1
Network Interconnect (Infiniband, TCP/IP)
MPI process 2
Thread pool
Thread pool
b) Parallel processing with MPI
Compute Node 1
Compute Node 2

Figure 2. Data decomposition and parallel execution with MPI.

total number of the MPI processes associated with the same job. Based on the rank and total processes number, each MPI process can calculate the corresponding subset of the input data and process it. The data partitioning problem remains beyond the scope of this work; particularly for RDF, there is a number of well-established approaches discussed in several previous publications, e.g., horizontal [16], vertical [17], and workload driven [18] partitioning.

Since a single MPI process owns its own memory space and thus can not access the data of the other processes directly, the MPI standard foresees special communication functions, which are necessary, e.g., for exchanging the data subdomain’s boundary values or consolidating the final output from the partial results produced by each of the processes. The MPI processes communicate with each other by sending messages, which can be done either in “point-to-point”(between two processes) or collective way (involving a group of or all processes).

```
import java.io.*;
import mpi.*;

class Hello {
  public static void main(String[] args) throws
       MPIException
  {
    int my_pe, npes; // rank and overall number of MPI
         processes
    int N;       // size of the RDF data set (number of
         triples)

    MPI.Init(args); // intialization of the MPI RTE

    my_pe = MPI.COMM_WORLD.Rank();
    npes  = MPI.COMM_WORLD.Size();

    System.out.println("Hello␣from␣MPI␣process" + my_pe +
         "␣out␣of␣" + npes);
    System.out.println("I’m␣processing␣the␣RDF␣triples␣
         from␣" + my_pe/npes + "␣to␣" + (my_pe+1)/npes);

    MPI.Finalize(); // finalization of the MPI RTE
  }
}
```

Listing 1. Acquiring rank and total number of processes in a simple MPI application

More details about the MPI communication can also be

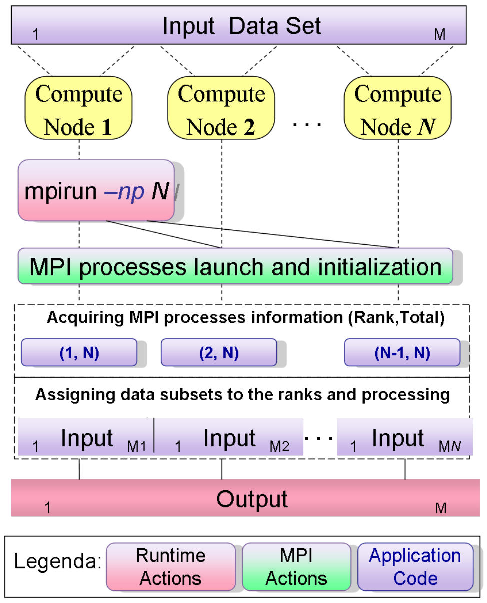


Figure 3. Typical MPI data-centric application's execution workflow

found in our previous publication [19].

IV. Open MPI Java Bindings

A. MPI bindings for Java

Although the official MPI standard's bindings are limited to C and Fortran languages, there has been a number of standardization efforts made towards introducing the MPI bindings for Java. The most complete API set, however, has been proposed by mpiJava [20] developers.

There are only a few approaches to implement MPI bindings for Java. These approaches can be classified in two following categories:

- Pure Java implementations, e.g., based on RMI (Remote Method Invocation) [21], which allows Java objects residing in different virtual machines to communicate with each other, or lower-level Java sockets API.
- Wrapped implementations using the native methods implemented in C languages, which are presumably more efficient in terms of performance than the code managed by the Java run-time environment.

In practice, none of the above-mentioned approaches satisfies the contradictory requirements of the Web users on application portability and efficiency. Whereas the pure Java implementations, such as MPJ Express [22] or MPJ/Ibis [9], do not benefit from the high speed interconnects, e.g., InfiniBand [23], and thus introduce communication bottlenecks and do not demonstrate acceptable performance on the majority of today's production HPC systems [24], a wrapped implementation, such as mpiJava [25], requires a native C library, which can cause additional integration and interoperability issues with the underlying MPI implementation.

In looking for a trade-off between the performance and the usability, and also in view of the complexity of providing Java support for high speed cluster interconnects, the most promising solution seems to be to implement the Java bindings directly in a native MPI implementation in C.

B. Native C Implementation

Despite a great variety of the native MPI implementations, there are only a few of them that address the requirements of Java parallel applications on process control, resource management, latency awareness and management, and fault tolerance. Among the known sustainable open-source implementations, we identified Open MPI [26] and MPICH2 [27] as the most suitable to our goals to implement the Java MPI bindings. Both Open MPI and MPICH2 are open-source, production quality, and widely portable implementations of the MPI standard (up to its latest 2.0 version). Although both libraries claim to provide a modular and easy-to-extend framework, the software stack of Open MPI seems to better suit the goal of introducing a new language's bindings, which our research aims to. The architecture of Open MPI [26] is highly flexible and defines a dedicated layer used to introduce bindings, which are currently provided for C, F77, F90 and some other languages (see also Figure 5). Extending the OMPI-Layer of Open MPI with the Java language support seems to be a very promising approach to the discussed integration of Java bindings, taking benefits of all the layers composing Open MPI's architecture.

C. Design and Implementation in Open MPI

We have based our Java MPI bindings on the *mpiJava* code [28]. mpiJava provides a set of Java Native Interface (JNI) wrappers to the native MPI v.1.1 communication methods, as shown in Figure 4. JNI enables the programs running inside a Java run-time environment to invoke native C code and thus use platform-specific features and libraries [29], e.g., the InfiniBand software stack. The application-level API is constituted by a set of Java classes, designed in conformance to the MPI v.1.1 and the specification in [20]. The Java methods internally invoke the MPI-C functions using the JNI stubs. The realization details for mpiJava can be obtained from [30][31].

Open MPI is a high performance, production quality, MPI-2 standard compliant implementation. Open MPI consists of three combined abstraction layers that provide a full featured MPI implementation: (i) OPAL (Open Portable Access Layer) that abstracts from the peculiarities of a specific system away to provide a consistent interface adding portability; (ii) ORTE (Open Run-Time Environment) that provides a uniform parallel run-time interface regardless of system capabilities; and (iii) OMPI (Open MPI) that provides the application with the expected MPI standard interface. Figure 5 shows the enhanced Open MPI architecture, enabled with the Java bindings support.

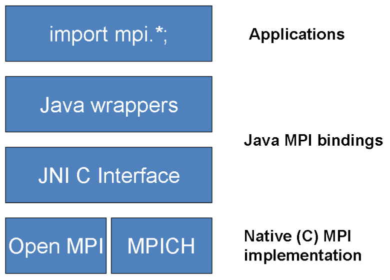


Figure 4. mpiJava architecture

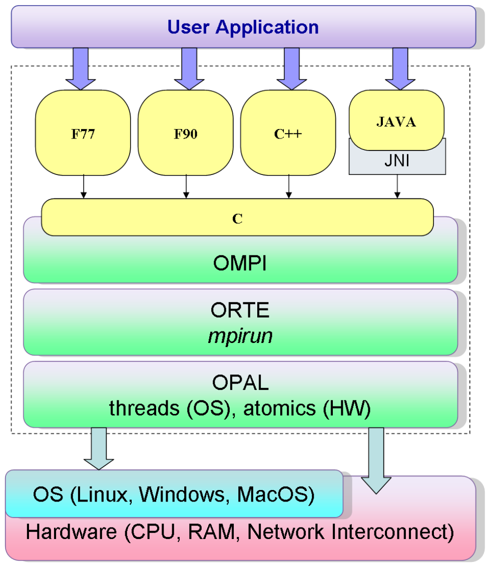

User Application
F77
F90
C++
JAVA
JNI
C
OMPI
ORTE
mpirun
OPAL
threads (OS), atomics (HW)
OS (Linux, Windows, MacOS)
Hardware (CPU, RAM, Network Interconnect)

Figure 5. Open MPI architecture

The major integration tasks we performed were as follows:

- Extend the Open MPI architecture to support Java bindings
- Extend the previously available mpiJava bindings to MPI-2 (and possibly upcoming MPI-3) standard
- improve the native Open MPI configuration, build, and execution system to seamlessly support the Java bindings
- Redesign the Java interfaces that use JNI in order to better conform to the native realization
- optimize the JNI code to minimize its invocation overhead
- Create test applications for performance benchmarking

Both Java classes and JNI code for calling the native methods were integrated into Open MPI. However, the biggest integration effort was required at the OMPI (Java classes, JNI code) and the ORTE (run-time specific options) levels. The implementation of the Java class collection followed the same strategy as for the C++ class collection, for which the opaque C objects are encapsulated into suitable class hierarchies and most of the library functions are defined as class member methods. Along with the classes implementing the MPI functionality (MPI package), the collection includes the classes for error handling (Errhandler, MPIException), datatypes (Datatype), communicators (Comm), etc. More information about the implementation of both Java classes and JNI-C stubs can be found in previous publications [30][24].

D. Performance

In order to evaluate the performance of our implementation, we prepared a set of Java benchmarks based on those well-recognized in the MPI community, e.g., NAS [32]. Based on those benchmarks, we compared the performance of our implementation based on Open MPI and the other popular implementation (MPJ Express) that follows a "native Java" approach. Moreover, in order to evaluate the JNI overhead, we reproduced the benchmarks also in C and ran them with the native Open MPI. Therefore, the following three configurations were evaluated:

- **ompiC** - native C implementation of Open MPI (the actual trunk version), built with the GNU compiler (v.4.6.1),
- **ompiJava** - our implementation of Java bindings on top of *ompiC*, running with Java JDK (v.1.6.0), and
- **mpj** - the newest version of MPJ Express (v.0.38), a Java native implementation, running with the same JDK.

We examined two types of communication: point-to-point (between two nodes) and collective (between a group of nodes), varying the size of the transmitted messages. We did intentionally not rely on the previously reported benchmarks, e.g. [33], in order to eliminate the measurement deviations that might be caused by running tests in a different hardware or software environment. Moreover, in order to ensure a fair comparison between all these three implementations, we ran each test on the absolutely same set of compute nodes.

The point-to-point benchmark implements a "ping-pong" based communication between two single nodes; each node exchanges the messages of growing sizes with the other node by means of blocking Send and Receive operations. As expected, our *ompiJava* implementation was not as efficient as the underlying *ompiC*, due to the JNI function calls overhead, but showed much better performance than the native Java based *mpj* (Figure 6). Regardless of the message size, *ompiJava* achieves around eight times higher throughput than *mpj* (see Figure 7).

The collective communication benchmark implements a single blocking message gather from all the involved nodes. Figure 8 shows the results collected for $P = 2^k$ (where *k=2-7*) nodes, with a varying size of the gathered messages. The maximal size of the aggregated data was 8 GByte on 128 nodes. Figure 9 demonstrates the comparison of collective gather performance for all tested implementations on the maximal number of the available compute nodes (128).

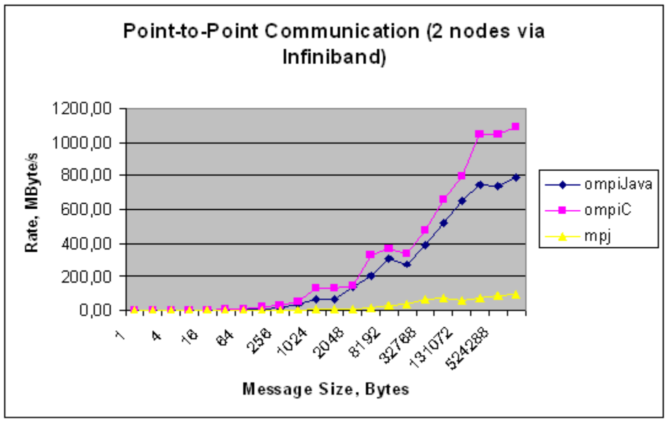
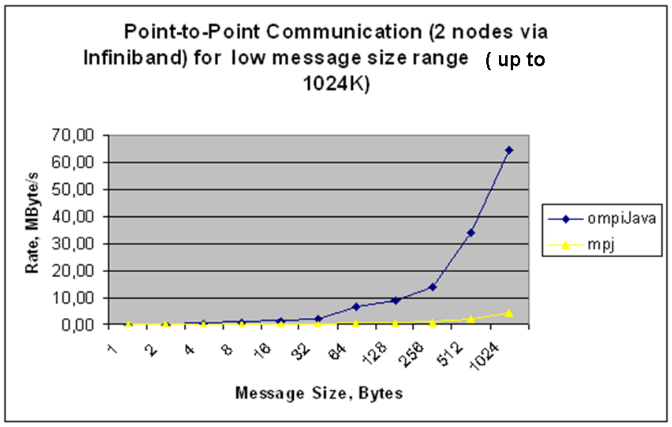
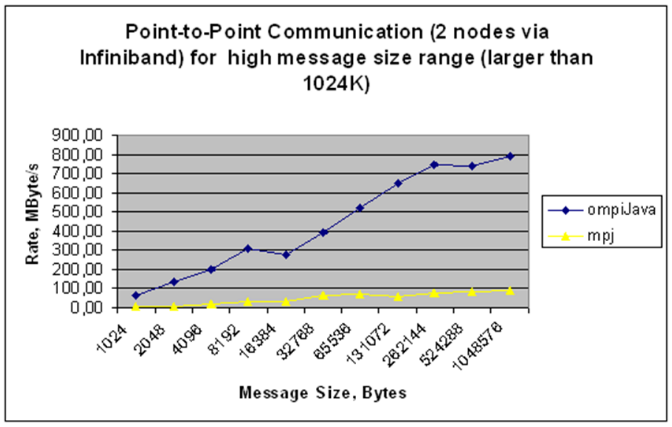


Figure 6. Message rate for the point-to-point communication



a)



b)

Figure 7. Comparison of the message rate for ompiJava and mpj for a) low and b) high message size range

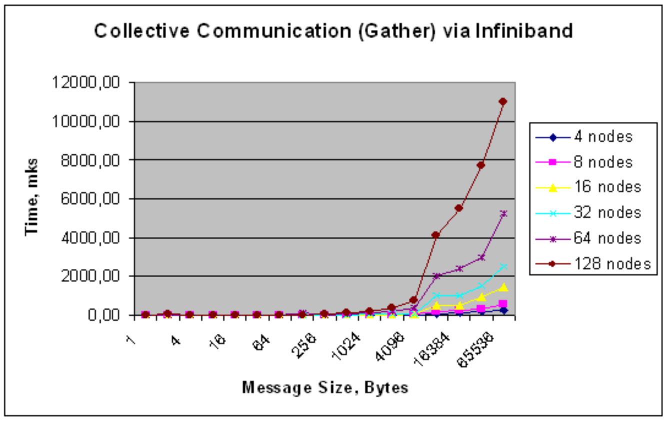


Figure 8. Collective gather communication performance of ompiJava

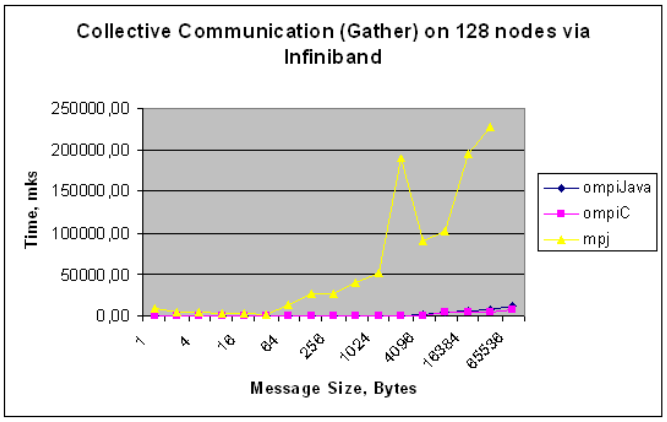


Figure 9. Collective gather communication performance on 128 nodes

Whereas the InfiniBand-aware *ompiJava* and *ompiC* scaled quite well, the native Java based *mpj* has shown very poor performance; for the worst case (on 128 nodes) a slow-down up to 30 times compared with *ompiJava* was observed.

V. MPI IMPLEMENTATION OF RANDOM INDEXING

Random indexing [34] is a word-based co-occurrence statistics technique used in resource discovery to improve the performance of text categorization. Random indexing offers new opportunities for a number of large-scale Web applications performing the search and reasoning on the Web scale [35].

The main challenges of the random indexing algorithms lay in the following:

- Huge and high-dimensional vector space. A typical random indexing search algorithm performs traversal over all the entries of the vector space. This means, that the size of the vector space to the large extent determines the search performance. The modern data stores, such as Linked Life Data or Open PHACTS consolidate many billions of statements and result in vector spaces

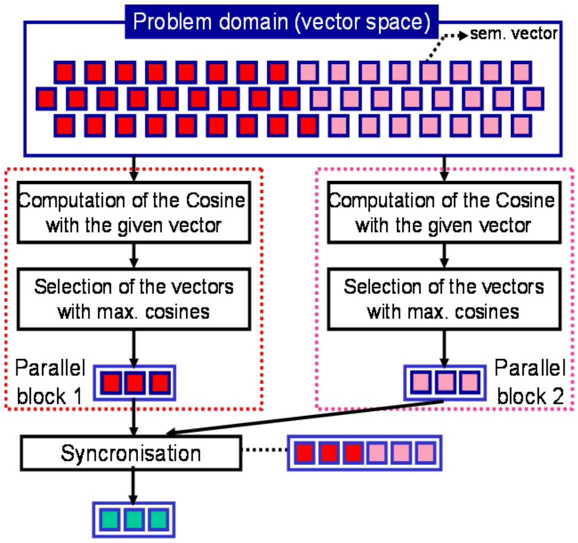

Problem domain (vector space)
sem. vector
Computation of the Cosine with the given vector
Selection of the vectors with max. cosines
Parallel block 1
Computation of the Cosine with the given vector
Selection of the vectors with max. cosines
Parallel block 2
Syncronisation

Figure 10. MPI-based parallel implementation of Airhead Search

of a very large dimensionality. Performing Random indexing over such large data sets is computationally very costly, with regard to both execution time and memory consumption. The latter poses a hard constraint to the use of random indexing packages on the serial mass computers. So far, only relatively small parts of the Semantic Web data have been indexed and analyzed.

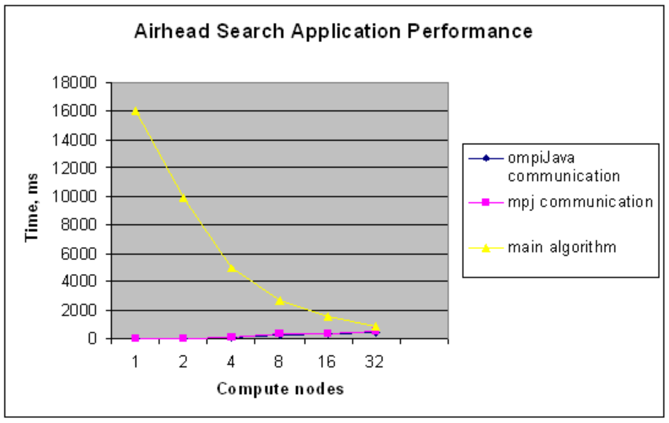
- High call frequency. Both indexing and search over the vector space is highly dynamic, i.e., the entire indexing process repeats from scratch every time new data is encountered.

In our previous work [36], we have already reported on the efforts done on parallelizing Airhead - an open source Java implementation of Random Indexing algorithm. Our MPI implementation of the Airhead search is based on a domain decomposition of the analyzed vector space and involves both point-to-point and collective gather and broadcast MPI communication (see the schema in Figure 10). In our current work, we evaluated the MPI version of Airhead with both *ompijava* and *mpj* implementations.

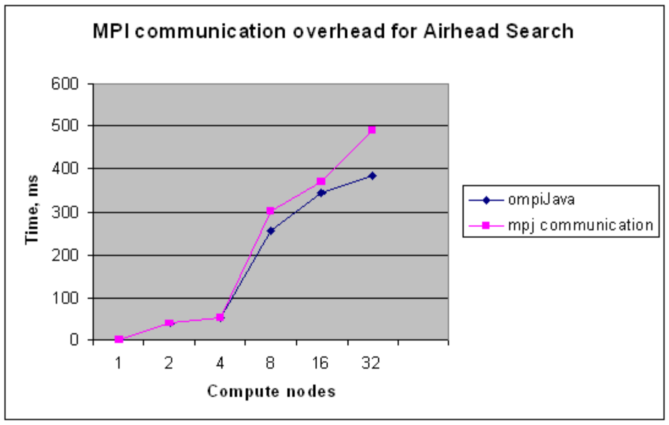
We performed the evaluation for the largest of the available data sets reported in [36] (namely, Wiki2), which comprises 1 Million of high density documents and occupies 16 GByte disk storage space. The overall execution time (wall clock) was measured. Figure 11a shows that both *ompijava* and *mpj* scale well until the problem size is large enough to saturate the capacities of a single node. Nevertheless, our implementation was around 10% more efficient over *mpj* (Figure 11b).

VI. Future Work

Our future work will concentrate on promoting both MPI standard and our ompiJava implementation to Semantic Web



a)



b)

Figure 11. Airhead performance with ompiJava and mpj

applications as well as improving the current realization of the Java bindings in Open MPI. With regard to promotion activities, we will be introducing our data-centric and MPI-based parallelization approach to further challenging data-intensive applications, such as Reasoning [37]. Regarding this application, there are highly successful MPI imlementations in C, e.g., the parallel RDFS graph closure materialization presented in [38], which are indicatively much more preferable over all the existing Java solutions in terms of performance. Our implementation will allow the developed MPI communication patterns to be integrated in existing Java-based codes, such as Jena [2] or Pellet [39], and thus drastically improve the competitiveness of the Semantic Web application based on such tools.

The development activities will mainly focus on extending the Java bindings to the full support of the MPI-3 specification. We will also aim at adding Java language-specific bindings into the MPI standard, as a reflection of the Semantic Web importance in supercomputing.

VII. Conclusion

High Performance Computing is relatively a new trend for the Semantic Web, which however has gained a tremendous popularity thanks to the recent advances in developing data-intensive applications.

The Message Passing Interface seems to provide a very promising approach for developing parallel data-centric applications. Unlike its prominent alternatives MapReduce and IBIS, the MPI functionality is delivered on the library-level, and thus does not require any considerable development efforts in order to be implemented in the existing serial applications. Using MPI, the Semantic Web applications can take full advantage of modern parallel computing resources. For the RDF processing algorithms, MPI allows for achieving higher scalability and eliminates the need of approximation and dependency minimization in partitioning the work load, used in the previous known implementations as a workaround to overcome the performance limitations on the serial hardware.

We introduced a new implementation of the Java bindings for MPI that is integrated in one of the most popular open source MPI-2 libraries nowadays - Open MPI. The integration allowed us to deliver a unique software environment for flexible development and execution of parallel MPI applications, integrating the Open MPI framework's capabilities, such as portability and usability, with those of mpiJava, such as an extensive set of Java-based API for MPI communication. We evaluated our implementation for Random Indexing, which is one of the most challenging Semantic Web applications in terms of the computation demands currently. The evaluation has confirmed our initial considerations about the high efficiency of MPI for parallelizing Java applications. In the following, we are going to investigate further capabilities of MPI for improving the performance of data-centric applications, in particular by means of MPI-IO (MPI extension to support efficient file input-output). We will also concentrate on promoting the MPI-based parallelization strategy to the other challenging and performance-demanding applications, such as Reasoning. We believe that our implementation of Java bindings of MPI will attract Semantic Web development community to increase the scale of both its serial and parallel applications. The successful pilot application implementations done based on MPI, such as materialization of the finite RDFS closure presented in [38], offer a very promising outlook regarding the future perspectives of MPI in this community.

Acknowledgment

The author would like to thank the Open MPI consortium for the support with porting mpiJava bindings as well as the EU-ICT Project LarKC [40], partly funded by the European Commission's ICT activity of the 7th Framework Programme (ICT-FP7-215535), for the provided pilot use case.

References

[1] Openphacts eu project website. [Online]. Available: http://www.openphacts.org [retrieved: June, 2012]

[2] P. McCarthy. Introduction to jena. IBM developerWorks. [Online]. Available: http://www.ibm.com/developerworks/xml/library/j-jena [retrieved: June, 2012]

[3] Lod cloud diagram. [Online]. Available: http://richard.cyganiak.de/2007/10/lod/ [retrieved: June, 2012]

[4] Open government data website. [Online]. Available: http://opengovernmentdata.org/ [retrieved: June, 2012]

[5] R. Gonzalez. (2012) Closing in on a million open government data sets. [Online]. Available: http://semanticweb.com/closinginona-millionopen-governmentdatasets_b29994 [retrieved: June, 2012]

[6] A. Cheptsov and M. Assel, "Towards high performance semantic web experience of the larkc project," *inSiDE - Journal of Innovatives Supercomputing in Deutschland*, vol. 9(1), pp. 569–571, Spring 2011.

[7] E. Goodman, D. J. Haglin, C. Scherrer, D. Chavarria, J. Mogill, and J. Feo, "Hashing strategies for the cray xmt," in *Proc. 24th IEEE Int. Parallel and Distributed Processing Symp.*, 2010.

[8] J. Dean and S. Ghemawat, "Mapreduce- simplified data processing on large clusters," in *Proc. OSDI04: 6th Symposium on Operating Systems Design and Implementation*, 2004.

[9] M. Bornemann, R. van Nieuwpoort, and T. Kielmann, "Mpj/ibis: A flexible and efficient message passing platform for java," *Concurrency and Computation: Practice and Experience*, vol. 17, pp. 217–224, 2005.

[10] (1995) Mpi: A message-passing interface standard. Message Passing Interface Forum. [Online]. Available: http://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-1.1/mpi-report.htm [retrieved: June, 2012]

[11] R. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. Bal, "Ibis: a flexible and efficient java based grid programming environment," *Concurrency and Computation: Practice and Experience*, vol. 17, pp. 1079–1107, June 2005.

[12] R. van Nieuwpoort, T. Kielmann, and H. Bal, "User-friendly and reliable grid computing based on imperfect middleware," in *Proc. ACM/IEEE Conference on Supercomputing (SC'07)*, November 2007.

[13] (2004, February) Resource description framework (RDF). RDF Working Group. [Online]. Available: http://www.w3.org/RDF/ [retrieved: June, 2012]

[14] "Lustre file system - high-performance storage architecture and scalable cluster file system," White Paper, SunMicrosystems, Inc., December 2007.

[15] Portable batch systems. [Online]. Available: http://en.wikipedia.org/wiki/Portable_Batch_System [retrieved: June, 2012]

[16] A. Dimovski, G. Velinov, and D. Sahpaski, "Horizontal partitioning by predicate abstraction and its application to data warehouse design," in *ADBIS*, 2010, pp. 164–175.

[17] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, "Scalable semantic web data management using vertical partitioning," in *Proc. The 33rd international conference on Very large data bases (VLDB'07))*.

[18] C. Curino, E. P. C. Jones, S. Madden, and H. Balakrishnan, "Workload-aware database monitoring and consolidation," in *SIGMOD Conference*, 2011, pp. 313–324.

[19] A. Cheptsov, M. Assel, B. Koller, R. Kbert, and G. Gallizo, "Enabling high performance computing for java applications using the message-passing interface," in *Proc. The Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering (PARENG'2011)*.

[20] B. Carpenter, G. Fox, S.-H. Ko, and S. Lim, "mpiJava 1.2: Api specification," Northeast Parallel Architecture Center. Paper 66, 1999. [Online]. Available: http://surface.syr.edu/npac/66 [retrieved: June, 2012]

[21] T. Kielmann, P. Hatcher, L. Boug, and H. Bal, "Enabling java for high-performance computing: Exploiting distributed shared memory and remote method invocation," *Communications of the ACM*, 2001.

[22] M. Baker, B. Carpenter, and A. Shafi, "MPJ Express: Towards thread safe java hpc," in *Proc. IEEE International Conference on Cluster Computing (Cluster'2006)*, September 2006.

[23] R. K. Gupta and S. D. Senturia, "Pull-in time dynamics as a measure of absolute pressure," in *Proc. IEEE International Workshop on Microelectromechanical Systems (MEMS'97)*, Nagoya, Japan, Jan. 1997, pp. 290–294.

[24] G. Judd, M. Clement, Q. Snell, and V. Getov, "Design issues for efficient implementation of mpi in java," in *Proc. the 1999 ACM Java Grande Conference*, 1999, pp. 58–65.

[25] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox, "MPJ: Mpi-like message passing for java," *Concurrency and Computation - Practice and Experience*, vol. 12(11), pp. 1019–1038, 2000.

[26] E. G. et al., "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proc., 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[27] Mpich2 project website. [Online]. Available: http://www.mcs.anl.gov/research/projects/mpich2/ [retrieved: June, 2012]

[28] mpijava website. [Online]. Available: http://sourceforge.net/projects/mpijava/ [retrieved: June, 2012]

[29] S. Liang, Ed., *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley, 1999.

[30] M. Baker, B. Carpenter, G. Fox, S. Ko, and S. Lim, "mpiJava: An object-oriented java interface to mpi," in *Proc. International Workshop on Java for Parallel and Distributed Computing IPPS/SPDP*, San Juan, Puerto Rico, 1999.

[31] M. Vodel, M. Sauppe, and W. Hardt, "Parallel high-performance applications with mpi2java - a capable java interface for mpi 2.0 libraries," in *Proc. The 16th Asia-Pacific Conference on Communications (APCC)*, Nagoya, Japan, 2010, pp. 509–513.

[32] Nas parallel benchmark website. [Online]. Available: http://sourceforge.net/projects/mpijava/ [retrieved: June, 2012]

[33] Mpj express benchmarking results. [Online]. Available: http://mpj-express.org/performance.html [retrieved: June, 2012]

[34] M. Sahlgren, "An introduction to random indexing," in *Proc. Methods and Applications of Semantic Indexing Workshop at the 7th International Conference on Terminology and Knowledge Engineering (TKE)'2005*, 2005, pp. 1–9.

[35] D. Jurgens, "The S-Space package: An open source package for word space models," in *Proc. the ACL 2010 System Demonstrations*, 2010, pp. 30–35.

[36] M. Assel, A. Cheptsov, B. Czink, D. Damljanovic, and J. Quesada, "Mpi realization of high performance search for querying large rdf graphs using statistical semantics," in *Proc. The 1st Workshop on High-Performance Computing for the Semantic Web*, Heraklion, Greece, May 2011.

[37] D. Fensel and F. van Harmelen, "Unifying reasoning and search to web scale," *IEEE Internet Computing*, vol. 11(2), pp. 95–96, 2007.

[38] J. Weaver and J. A. Hendler, "Parallel materialization of the finite rdfs closure for hundreds of millions of triples," in *Proc. International Semantic Web Conference (ISWC) 2009*, A. B. et al., Ed., 2009.

[39] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: a practical owl-dl reasoner. Journal of Web Semantics. [Online]. Available: http://www.mindswap.org/papers/PelletJWS.pdf [retrieved: June, 2012]

[40] Eu-fp7 project large knowledge collider (larkc). [Online]. Available: http://www.larkc.eu [retrieved: June, 2012]