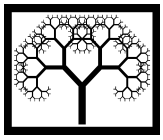
Paper 74?



Proceedings of the Eighth International Conference
on Engineering Computational Technology,
B.H.V. Topping, (Editor),
Civil-Comp Press, Stirlingshire, Scotland

Efficient Parallelization of Java Applications for Semantic Web by means of the Message-Passing Interface

A. Cheptsov
HLRS - High Performance Computing Center Stuttgart
University of Stuttgart, Germany

Abstract

A number of software applications developed in Java, such as ones coming from information retrieval and semantic web research domains, have been faced with the performance and scalability issues in view of the growing computation demands. The Message-Passing Interface (MPI) has proved to be an efficient solution for a wide variety of parallel applications developed with "traditional" high performance computing languages such as C and Fortran. We demonstrate that the design features of Java prevent the native MPI realization to massively scale on the production high performance computing systems. As a reaction on this challenge, we present a solution that allows the performance issues of the native implementation to be overcome by integration in the highly-scalable C realization such as Open MPI. Implementation of the proposed parallelization technique, based on the domain decomposition, for a pilot Java application (Airhead), which performs random indexing and search in large semantically annotated text sets, has allowed us already to improve the performance by up to 33 times on 16 nodes of the test bed cluster system.

Keywords: semantic web, Java, parallelization, message-passing interface, random indexing, Open MPI.

1 Introduction

Driven by the concepts of portability and interoperability, Java has become a widely accepted general-purpose programming language with a large existing code base and programmer communities. Among others, Java has gained a wide adoption in data-centric computing such as information retrieval and semantic web that has a potential demand for parallel and high-performance computing. Whereas the recent advances of those communities require their Java applications to scale up to the requirements of the vast and rapidly increasing data, e.g. coming from millions of

sensors in Smart Cities domain, Java fairly lacks mechanisms that would enable the Java applications to scale beyond the single NUMA-node across the network interconnect of a modern supercomputing system.

At present, the interest in Java is largely shifting towards the exploitation of the computation power of distributed-memory parallel computers. There are already some developments such as MapReduce framework's implementation of Hadoop or parallel programming environment Ibis, which prove that the Java execution environment can successfully exploit HPC, scaling to sizes unreachable by native thread-based implementations. However, none of them is near as efficient or well-developed as found in the Message-Passing Interface (MPI).

MPI has become a de-facto standard in the area of parallel computing for C, C++, and Fortran applications, providing API that allows processes running on different nodes of a supercomputing system to be synchronized with each other by means of messages communicated between or among those nodes. But given the vast problem sizes addressed by the modern Java applications, and given the emergence of the new Java communities interested in adopting MPI, it seems natural to explore the benefits of MPI for Java applications on HPC platforms as well.

Introducing MPI for Java applications poses several challenges. First, the API set should be compliant with the MPI standard, but not downgrading the flexibility of the native Java language constructions. Second, the hardware support should be offered in a way that overcomes the limitation of the Java virtual machine, but ensures such important features as thread-safety. Third, MPI support should be seamlessly integrated in the parallel applications' run-time environment. These three issues of functionality, adaptivity, and usability must be addressed in complex to make the use of MPI in Java applications practical and useful.

We demonstrate that the design features of Java prevent the native MPI realization to massively scale on the productional high performance computing systems and look for solutions of resolving these issues in a way that leverages the advances of the existing MPI frameworks. The paper is organized as follows. In Section 2, we present the case study Java application. In Section 3, we describe the parallelisation technique elaborated for the pilot application; the technique is quite generic and can be adopted by any other Java application. Section 4 provides implementation details. Section 5 collects performance evaluation results. Section 6 contains the conclusions.

2 Case Study Java Application

The tremendous increase of the structured data operations on the Web, observed in the last few years in Science (e.g. the Linked Life Data repository for biomedical science concentrating over 4 billion RDF statements), Government (such as Linked Open Data[1] repository containing more as 4 billion statements), Commerce (as provided by Ontotext[2] and other companies), and many other domains, put a number

[1] http://www.data.gov/

[2] http://ontotext.com/

of requirements to the software packages using those data. The massive amount of data, in particular described by RDF (Resource Description Framework) [3] – a standard model for data interchange on the web, is a key challenge for many Semantic Web applications. Given that Java is widely used in the Semantic Web, not surprisingly, we chose the Semantic Web domain when looking for the large-scale applications that would potentially benefit from the distributed parallelisation. As one of the most challenging application areas of the Semantic Web, in terms of the allocated resources, we identified the random indexing [1].

Random indexing is a distributional statistic technique used for extracting semantically similar words from the word co-occurrence statistics in the text data, based on high-dimensional vector spaces. Random indexing is a promising technique offering new opportunities for a number of large-scale Web applications performing the search and reasoning on the Web scales. The prominent examples of applications using random indexing are query expansion and subsetting. Query expansion [2] is used in information retrieval with the aim to expand the document collection returned as a result to a query, thus covering the larger portion of the documents. Subsetting (also known as selection) [3], on the contrary, deprecates the unnecessary items from a data set in order to achieve faster processing. Both presented problems are complementary, as change properties of the query to best adapt it to the search needs.

The main complexity of the random indexing algorithms lies in the following:

- High dimensionality of the underlying vector space. The vector space consists of context vectors whose relative directions are assumed to indicate semantic similarity. 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 defines the search performance. The modern data stores, such as Wikipedia[4], or FactForge[5], consolidate many billion of statements and result in vector spaces of a very large dimensionality.
- High call frequency. Both indexing and search over the vector space is typically a one-time operation, which means that the entire process should be repeated from scratch every time new data is encountered.

Due to the above-mentioned reasons, random indexing over large data sets is computationally very costly, with regard to both execution time and memory consumption. The latter is of a special drawback for performing random indexing on the mass computers. So far, only relatively small parts of the Semantic Web data have been successfully indexed and analyzed.

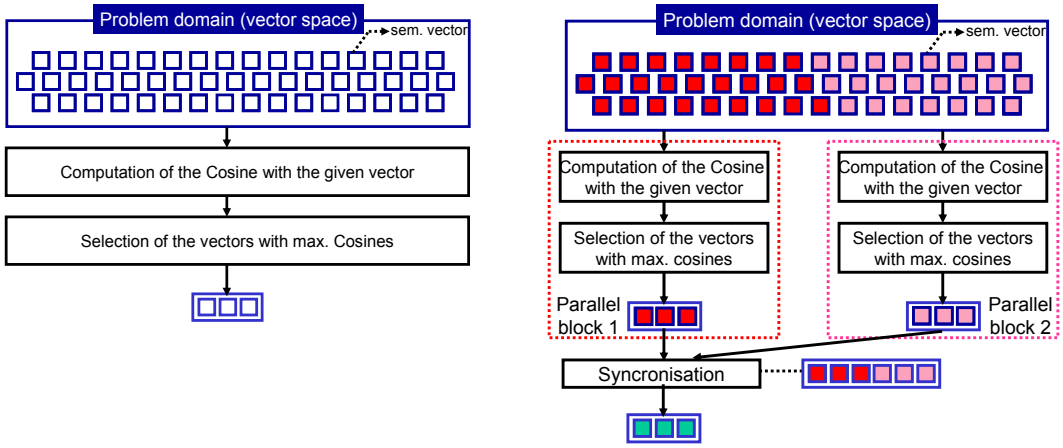
One of the most successful software packages for random indexing, widely recognised in the Semantic Web community, is Airhead [4], developed in University of California. Airhead is a Java library performing both random indexing and search. Currently, Airhead is able to take full advantage of a multi-core environment. However, the algorithm can not be executed on a distributed parallel system, such as a cluster of workstations or a supercomputer, which makes processing terabytes of

[3] http://www.w3.org/RDF/
[4] http://wikipedia.org
[5] http://factforge.net/

text increasingly infeasible. As a workaround, we have developed an MPI version of the application that performs the search on the most similar words to the given one in the semantic vector space.

3 Parallelisation approach

Random indexing search is performed over the entries of the semantic vector space according to the schema depicted in Figure 1a. All the vectors are processed independently and concurrently. The trivial parallelisation can be achieved by mapping the contiguous sets of vectors in the vector space to a parallel block, each running on a compute node. The execution on single nodes is followed then by a synchronization to wait for other parallel blocks and gather the partial results of all the blocks. The division of the vectors among the parallel blocks is specified by the domain decomposition (Figure 1b). The domain decomposition ensures the optimal load balancing among the compute nodes and therefore the highest performance of the parallelised algorithm.



a) b)

Figure 1: The sequential (a) and parallel (b) realisation of the implemented random indexing algorithm; in the example $n=3$ words are selected which vectors are the most similar to the given word.

The results of the search in the part of the vector space, assigned to the parallel block/process, are stored in the block's memory space and can not be accessible from another block. However, this is needed to perform the final selection among the results of each of the blocks. For this purpose, all the partial outputs might be gathered, in one of the blocks (the "root" one), where then the final selection is performed. The necessity of passing the results (n selected words) from each block to the root one as well as the following final selection prevents the parallelised application performance from super-linear scalability. Nevertheless, the optimal realisation of the synchronisation allows the parallel algorithm to minimise the computation overhead of this operation in total execution time.

4 Distributed Parallelisation with MPI

4.1 Introduction to MPI

MPI is a wide-spread process-oriented implementation standard for parallel applications [5], implemented in many programming languages, also including Java [6]. Each MPI processes has an own memory space and communicates with the other process/processes to access its/there data. In MPI, each process is identified by means of its rank, which is unique within a group of processes involved in the execution (Figure 2).

```
int my_rank = MPI.COMM_WORLD.Rank();
int total_processes = MPI.COMM_WORLD.Size();
System.out.println("Process " + my_rank + " out of " + total_processes);
```

Figure 2: Requesting the rank of the process and the size of the group (number of the involved processes), following the Java specification of MPI.

Communication between or among the MPI processes performs by means of messages. In case of such an object-oriented language as Java, a message encloses any object, including a single variable, an array, or a class. For the latter, a type casting operation is required on the receiver's side to match the "Object" type to the needed one.
The message can be transmitted either between two processes, a sender and a receiver (point-to-point communication), or among several processes involved in a group (group communication). The table below summarises the main MPI operations for Java used in the parallel version of the described application.

Communication	Description
Point-to-point communications (between two nodes)	
MPI.COMM.Send MPI.COMM.Recv	Synchronous send/receive of a message
Collective communications (among many/all nodes)	
MPI.COMM.Bcast	Broadcast of a message from the specified node/process to all the other processes in the group
MPI.COMM.Gather	Gather of a message from all the nodes/processes in the group by the specified process

Table 1: Main MPI communication operations for Java

For more detailed information about the MPI communications, please refer to [6].

4.2 MPI libraries for Java

Among the available implementations of MPI for Java, we considered the following two in this paper as being the most popular and sustainable at the moment:

- mpiJava
- MPJ Express

mpiJava [7] is an outcome of the HPJava project. The main feature of mpiJava is that it extents a native MPI library by wrapping its MPI calls (mainly for C and Fortran) to the Java interfaces (Figure 3).

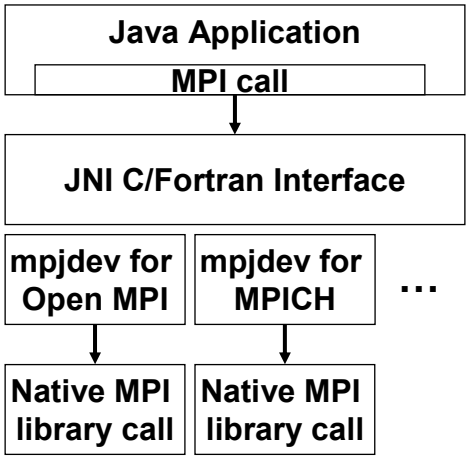


Figure 3: Architecture of mpiJava.

Thanks to its flexible architecture, mpiJava is portable to any platform that provides compatible Java development and native MPI environments. The lightweight realization of the JNI interfaces ensures the minimum overhead when executing the native MPI calls. A great variety of underlying MPI libraries are supported by mpiJava, including such popular ones as Open MPI, MPICH, LAM and others. The mpiJava's APIs has become de-facto a standard specification for Java language.
MPJ Express [8] is a relatively new development, promising in terms of installation and maintenance ease. Unlike mpjJava, MPJ Express provides an own realisation of the MPI calls, which might be benefitial for some cases[6]. Among the main benefits of MPJ Express are also such as ease of application deployment and debugging on the user's home machine, no need in underlying native MPI installation for C or Fortran, and MPI via multithreading support.
In order for a Java application to make use of the both described MPI libraries, no changes in the application code are needed. In our evaluation, we successfully used both mpiJava and MPJ Express for implementation of the parallel algorithms.

4.3 Realisation for Airhead

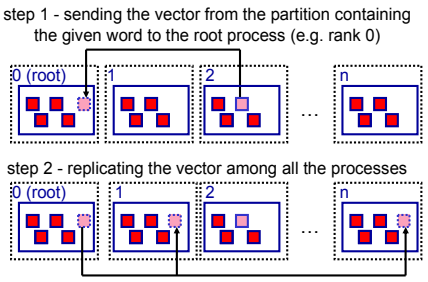
According to the domain decomposition in Figure 1b, each process is assigned to a subdomain, the boundary elements of which are calculated dynamically based on the MPI process's rank in the communication group (Figure 4).

```
int domain_begin = VectorSpace.size() * (my_rank);
int domain_end   = VectorSpace.size() * (my_rank + 1);
```

Figure 4: Using MPI for calculating size of the processes' subdomains.

[6] See some benchmarking results at http://mpj-express.org/performance.html

Besides the vectors assigned, each subdomain also requires the vector containing the given searched word. Thus, the process whose subdomain contains the given word's vector must replicate its value over all the other processes. The replication schema as well as the Java code for this operation are presented in Figure 5.



a)

```
if (MPI.COMM_WORLD.Rank() == 0) {

//check whether the process contains the searched vector
if (searched_vector == null) {
                MPI.COMM_WORLD.Recv(searched_vector, 0, 1, MPI.OBJECT,
                                                MPI.ANY_SOURCE, 1);
        }

}
else
if (vector != null)
        MPI.COMM_WORLD.Send( vector_tosend, 0, 1, MPI.OBJECT, 0, 1);

MPI.COMM_WORLD.Bcast(searched_vector, 0, 1, MPI.OBJECT, 0);
```

b)

Figure 5: Expansion of the searched word's vector to the distributed vector space's partitions: a) replication schema b) Java code.

Realisation of the main algorithm (Figure 1b) remains mainly unchanged. The only adaptation needed is specification of the vector space's domain to be searched in by each of the MPI processes, based on the values from Figure 4. After the search is finished, the partial outputs of each of the MPI processes are to be collected by the root process, as shown in Figure 6.

```
Object local[] = variable_to_gather;
Object common[] = new Object[comm_size];
MPI.COMM_WORLD.Gather(local, 0, 1,
        MPI.OBJECT, common, 0, 1, MPI.OBJECT, 0);
if (my_rank == 0)     //  only for the root process
        for (int i=0; i<comm_size; i++) {
                Map<Double,String> map =
                        (Map<Double,String>) common[i];
            …//further processing
        }
```

Figure 6: Gathering the partial search results by the root process using MPI.Gather operation.

5 Performance evaluation

Performance of the parallelised algorithms was evaluated on the three test data sets taken from the random indexing application developed in LarKC. The data sets' parameters are described in Table 2.

Vector space	Nr. of entries	Size on the disk, GB	Description
LLD	0,5 M	0,65	A subset of Linked Life Data
Wiki1	1 M (low density, terms only)	1,6	A term set from Wikipedia articles
Wiki2	1 M (high density, entire documents)	16	A document set from Wikipedia articles

Table 2: Benchmarked Vector Spaces

The evaluation was performed on two compute clusters of High Performance Computing Center Stuttgart – *BW-Grid* and *Nehalem*. BW-Grid offers compute nodes of Xeon "Harpertown" architecture (2,83GHz CPU freq., 12GB RAM). BW-Grid deploys MPJ-Express v.3.6, supporting communication over Gigabit Ethernet. Nehalem provides Intel "Nehalem" based compute nodes (2,8GHz CPU freq., 16GB RAM). Nehalem deploys mpiJava on top of native Open MPI v.1.3.6 implementation, supporting communication through Infiniband interconnect.
Configuration of 1, 2, 4, 8, and 16 compute nodes were benchmarked to evaluate the scalability of the developed algorithms (Table 3).

a)

Data set	Number of compute nodes	Time, s.				Speed-up, times
		Data set load	*Search*	*MPI comm.*	*Total*	
LLD	1	14,7	4,7	0	21	1
	2	7,7	2,5	0,028	10,8	2
	4	4,1	1,4	0,2	6,2	3.4
	8	2,3	0,9	0,22	3,8	5.5
	16	1,6	0,65	0,375	2,8	7.5
Wiki1	1	28,5	1,5	0	30,5	1
	2	15,4	0,99	0,027	16,9	1.8
	4	7,8	0,76	0,039	9,1	3.4
	8	4,4	0,6	0,42	5,5	5.6
	16	2,7	0,54	0,64	3,9	7.8
Wiki2	1	n.a.				
	2	81	4,8	0,35	89	1
	4	67	2,7	0,28	71	1.25
	8	33,3	1,5	0,22	35	2.5
	16	16,8	0,9	0,2	18,4	4.8

b)

Vector Space	Number of compute nodes	Time, s.				Speed-up, times
		Loading	*Search*	*MPI comm.*	*Total*	
LLD	1	12	6	-	19,5	1
	2	4	3,3	0,03	7,9	2,47
	4	2,4	1,8	0,23	4,6	4,24
	8	1,2	1	0,16	2,9	6,72
	16	0,6	0,7	0,2	2	9,75
Wiki1	1	18	4	-	22	1
	2	8,9	3,8	1	13,3	1,65
	4	4,6	2	0,08	7,4	2,97
	8	2,3	1,3	0,23	4,4	5
	16	1,2	0,75	0,52	2,8	7,86
Wiki2	1	309	83	-	395	1
	2	59	27	0,58	88	4,5
	4	35	13	16	59,1	6,7
	8	20	8	4	32,2	12,3
	16	10	3,7	0,16	14,6	27

Table 3: Performance characteristics for parallel realisation of airhead search: a) BW-Grid b) Nehalem

The evaluation reveals that the MPI realisation allows the random indexing application to dramatically improve its performance. The maximum speed-up of about 27 times was achieved on 16 Nehalem nodes for the largest of the tested data sets (Wiki2). Good scalability of the parallelised algorithm over increasing number of compute nodes was observed for the both testbed configurations (Figure 7).

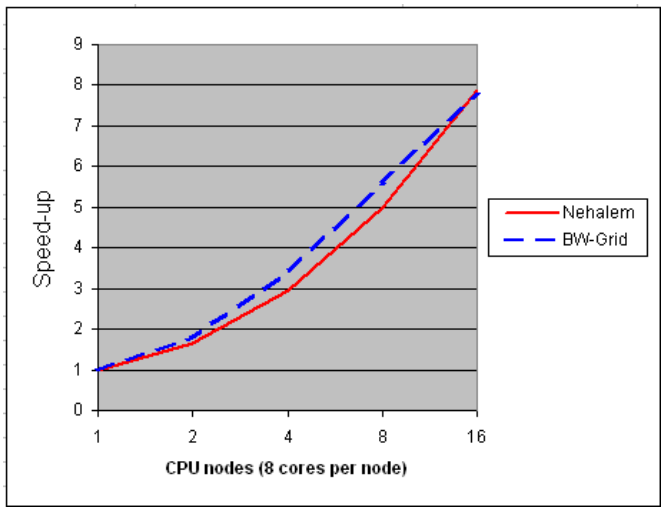


Figure 7: Performance speed-up of the parallel realisation on the parallel architectures (for the Wiki1 data set).

Along with qualitative performance improvement, the parallel realisation enables much larger data sets to be processed using the random indexing technique in scope of the available resources. As shown in Table 2, we were unable to execute the sequential application on a single BW-Grid node (marked as "n.a.") due to RAM limitation of a single node. The parallelisation by means of MPI allows the application not only to share the computation load among the nodes of a parallel system, but also to consolidate the resource characteristics of the single nodes to fit to the experiment requirements (mainly defined by the size of the vector space). For example, even on a generic cluster, such as BW-Grid, the overall amount of RAM is enough for processing even the largest data sets available at the moment of preparing this paper with random indexing.

6 Conclusions and further research

Since there have been many large-scale scientific applications implemented in Java, for which performance becomes a major challenge, much interest has recently arisen around enabling High Performance Computing for such applications. Similar to any other programming language, parallelisation is a key factor in achieving the necessary performance and scalability over the required problem size for Java applications. For some domains, such as Semantic Web, parallelisation is the only way of developing successful and scalable applications.

The Message-Passing Interface is the most efficient technique of implementation of parallel applications, also introduced in Java. Nevertheless, for a long time this technique was underestimated in use for Java developments due to many reasons; perhaps main of them is complexity of applying a process based programming model. This paper is an attempt to close the gap between Java and MPI. Presenting a common parallelisation strategy, which is based on domain decomposition, we

implemented the parallel version of the search operation from the Airhead library with MPI, based on the sequential code. The described technique allows any other Java developer to apply parallelism to his/her application with the minimum knowledge about MPI. For the tested application, we achieved a speed-up of almost 33 times on 16 compute nodes, as compared with the sequential version. Moreover, the parallel implementation allowed us to perform a complex experiment on the resource, whose capacities were not enough to run the sequential version of the application. With our experience we would like to encourage other researchers to apply the MPI-based parallelization for their Java applications as well.

In our performance tests, we used different hardware and software resource configurations to demonstrate the robustness of the chosen approach. We aimed at not comparing the available MPI implementations for Java, but rather demonstrating the efficiency of the developed parallelisation techniques, in order to promote the MPI based parallelisation approach into the wider communities. However, the applications will definitively benefit from the optimal configuration of the HPC system they are deployed on.

It is also worth mentioning that MPI is not unique in its goal of enabling HPC for Java applications. The newly emerging parallelisation paradigms, such as Hadoop realization of MapReduce[7], offer a promising view at developing efficient parallel applications as well. Our next step will be in-depth investigation and comparison of different parallelisation technologies and elaborating an optimal strategy to be followed by the developer willing to start parallelising his/her sequential application from scratch.

Acknowledgment

We thank Open MPI consortium for the support with integrating Java bindings as well as the LarKC project (http://www.larkc.eu) for the proposed test use case.

References

[1] M. Sahlgren. An introduction to Random Indexing, in Methods and Applications of Semantic Indexing Workshop at the 7th International Conference on Termonology and Knowledge Engineering, TKE 2005, Citeseer, 2005.

[2] Efthimis N. Efthimiadis. Query Expansion. In: Martha E. Williams (ed.), Annual Review of Information Systems and Technology (ARIST), v31, pp 121–187, 1996 - An introduction for less-technical viewers.

[3] Bastian Quilitz, Ulf Leser. Querying Distributed RDF Data Sources with SPARQL. 5th European Semantic Web Conference (ESWC2008)

[7] http://en.wikipedia.org/wiki/MapReduce

[4] Jurgens and Stevens, (2010). The S-Space Package: An Open Source Package for Word Space Models. In System Papers of the Association of Computational Linguistics.
[5] The MPI standard http://www.mcs.anl.gov/research/projects/mpi/standard.html
[6] The Java API for MPI http://www.hpjava.org/theses/shko/thesis_paper/node33.html
[7] Bryan Carpenter, Vladimir Getov, Glenn Judd, Tony Skjellum and Geoffrey Fox. MPJ: MPI-like Message Passing for Java. Concurrency: Practice and Experience, Volume 12, Number 11. September 2000
[8] Mark Baker, Bryan Carpenter, and Aamir Shafi. MPJ Express: Towards Thread Safe Java HPC. IEEE International Conference on Cluster Computing (Cluster 2006), Barcelona, Spain, 25-28 September, 2006