ssdnm
środowiskowe
studia doktoranckie
z nauk matematycznych

Marek Nowicki

Uniwersytet M. Kopernika w Toruniu

Developing of PCJ — a new library for distributed calculations in Java language

Praca semestralna nr 3

(semestr zimowy 2011/12)

Opiekun pracy: Joanna Trylska

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Developing of PCJ – a new library for distributed calculations in Java language

Marek Nowicki[1,3], Piotr Bała[1,2], and Joanna Trylska[3]

[1] Faculty of Mathematics and Computer Science,
Nicolaus Copernicus University,
Chopina 12/18, 87-100 Toruń, Poland
`faramir@mat.umk.pl`

[2] Interdisciplinary Centre for Mathematical and Computational Modelling,
University of Warsaw,
Pawińskiego 5a, 02-106 Warsaw, Poland
`bala@mat.umk.pl`

[3] Centre of New Technologies,
University of Warsaw
Żwirki i Wigury 93, 02-089 Warsaw, Poland
`joanna@icm.edu.pl`

Abstract In this paper we present PCJ - a new library for parallel computations in Java. We present the design details and evaluation of the performance of the PCJ communication on the state of art hardware such as cluster with gigabit interconnect. The results are compared with the native MPI implementation showing good performance and scalability.

1 Introduction

The changes in the hardware are associated with the interest in new programming languages not being considered by traditional high performance computing. A good example is Java with its increasing performance and parallelization tools such as Java Concurrency which has been introduced in Java SE 5 and improved in Java SE 6 [1]. The parallelization tools available for Java do not limit to threads and include solutions based on various implementations of the MPI library [2], distributed Java Virtual Machine [3], solutions based on Remote Procedure Call (RPC) [4] and Remote Method Invocation (RMI) [5].

In our work, we present a new approach motivated by the partitioned global address space approach represented by CoArray Fortran or UPC [6]. It has been designed and implemented as the Java library called PCJ (Parallel Computations in Java). PCJ offers methods for partitioning work, synchronizing nodes, getting and putting values in means of asynchronous one-sided communication. The library provides methods for broadcasting, creating groups of nodes, and monitoring changes of variables. The PCJ library is created to help develop parallel applications which have large requirements for memory, bandwidth or processing power. A good example are scientific and engineering applications or applications used in financial simulations. PCJ can be considered as a necessary set of methods to program distributed applications.

In this paper we evaluate the performance of PCJ using a relevant subset of Java Grande Forum Benchmark Suite tests [7] executed on the cluster with gigabit interconnection. The results are compared with analogous tests using the MPI library written in C.

2 Library Description

PCJ has been developed from scratch using the newest version of Java SE 7 with its advantages such as new IO library [8]. Using the newest version of Java increases the performance, prolongs the library life and, in the future, helps to move it to more recent versions of Java. Java SE 7 implements Sockets Direct Protocol (SDP) which can increase network performance over Infiniband connections.

The library developed by us offers elementary methods required to develop distributed parallel applications, such as: getting a value from a node, putting a value to a node, synchronizing all nodes. It also contains more advanced methods like joining groups, broadcasting messages, monitoring and waiting for a variable change.

The PCJ library is built based on some fundamental assumptions presented below.

2.1 PCJ Fundamentals

In the PCJ library each node runs its own calculations and has its own local memory. Therefore, by default the variables are stored and accessed locally. Some variables can be shared between nodes, so in PCJ they are called shared variables.

One node is intended to be the *Manager* which starts calculations on other nodes and takes responsibility for setting unique identification to nodes, creating and assigning nodes into groups and synchronizing nodes within a group. *Manager* is running on the main JVM – one which starts PCJ. The remaining nodes are used for calculations. So if one wants to use sixteen nodes in the calculations, one additional node is needed to serve as the manager. Since the manager is not CPU intensive, it can be run on the same physical node as one of the nodes used in the calculations.

All variables, which are shareable, are stored in a special *Storage* class. Each node has one and only one *Storage* instance. Each *shared variable* should have a special annotation `@Shared` with *share-name* of that variable. The compiler checks if the *share-name* is ambiguous, if so, displays an appropriate error note. The class can become the *Storage* by extending `pl.umk.mat.pcj.storage.StorageAbstract` class. An example of the *Storage* class definition is available in the Listing 1.1.

```java
public class BcastStorage extends StorageAbstract {

    @Shared("array")     // variable identifier = share-name
    private double[] array;
}
```

Listing 1.1. *Storage* class

Next, the type of each *shared variable* has to implement `java.io.Serializable`. When sending or receiving the variable, the content is converted to byte array using the serialization mechanism. If the variable type does not implement `java.io.Serializable` interface, the compiler indicates an error.

The last assumption is that there is a *start point* class. This class should implement the `pl.umk.mat.pcj.StartPoint` interface, what indicates, that it should contain the `public void main()` method. This method is executed after initializing PCJ, as a starting point like `public static void main(String[] args)` method in the normal execution. The example of the *start point* class can be viewed in the Listing 1.2. In this example the current node *global id* is also determined and printed out.

```java
public class Bcast implements StartPoint {

    @Override
    public void main() {
        System.out.println("My node id is: " + PCJ.myNode());
    }
}
```

Listing 1.2. *Start point* class

Any class can be both *Storage* and *Start point* by extending and implementing appropriate class and interface.

2.2 Protocol

During initialization, the newly created node connects to the *Manager* to inform about the successful start and to receive its unique *global node id*. Other, already connected nodes get information about the new one. The node that in the calculations receives information about a new node *welcomes* the new node by connecting to its listening address and obtaining its *global node id*. The *Manager* sends a message to the first node (with $nodeId = global\ node\ id = 0$). Then this node sends information to its two children ($nodeId * 2 + 1$ and $nodeId * 2 + 2$), therefore, the information is sent to all nodes by using the binary tree with communication complexity $O(n \log n)$.

Every node that has finished initialization waits for all other nodes to connect to the *Manager*. When all nodes are connected, the *Manager* sends the signal to start the calculations. This is performed by broadcasting a dedicated message over the tree structure of nodes. The node that receives the message to start the calculations, runs the `public void main()` method from the *start point* class.

The nodes can be grouped to simplify the code and optimize data exchange. Joining the group works similarly to initialization. The node sends the message to join a specified group (groups are distinguished by names) to the *Manager*. The *Manager* checks if the group already exists and then the node receives its *group node id*. All nodes in that group are notified, using the tree structure of nodes, about a new node in the group. Then the group members *welcome* the new node by sending their *global node id* and associated *group node id*. One node can be a member of many groups. If a node sends request to join a non-existing group the *Manager* simply creates it.

Working with all nodes and with a group of nodes is identical. The group which gathers all nodes is called *global group*. There are some small differences in implementation, but all the descriptions presented below are true for a global group and subset of nodes affiliated to groups.

Each node can give a value to any other node. The value is put in an asynchronous way. The receiving node does not interrupt when it is assigned a value. The receiver can monitor attempts of modification of a variable using the *monitor* and *waitFor* methods. A new value is put to the receiving node to its *Storage* space.

The get method is analogous to the put. The node gets the value from another node's *Storage* space in an asynchronous way – the sender does not interrupt its own calculations when sending the value from its *Storage* space.

Synchronization, also known as barrier, works in a similar way to the procedure used to start calculations. Each node in the group is supposed to call the *sync* method. Upon calling this method the node sends an appropriate message to the *Manager* and pauses the current thread until the *Manager* receives messages from all nodes. Then the *Manager* sends, using the previously described tree structure of the nodes, a message to continue calculations. There are methods for synchronizing all nodes, nodes in created groups or to synchronize nodes not associated to the groups. The synchronization of nodes, even without creating a group, is a way to get synchronous put and get methods.

Broadcast is performed by putting a value to all nodes in a group using the tree structure of nodes. Broadcast works in an asynchronous manner.

2.3 Examples

Starting up the calculations using PCJ is displayed in Listing 1.3.

```
    public static void main(String[] args) throws Throwable {
        /* read configuration file */
        Configuration conf = Configuration.parse(
                                    new File(args[1]));

        /* get information about nodes from configuration */
        NodeInfo[] nodes = conf.getNodes().toArray(
                                    new NodeInfo[0]);

        /* get manager information from configuration */
        ManagerInfo[] managers = conf.getManagers().toArray(
                                    new ManagerInfo[0]);

        PCJ.deploy(Bcast.class,     // StartPoint
                  BcastStorage.class, // Storage
                  managers,           // managers info
                  nodes);             // nodes info
      }
```

Listing 1.3. Deploying PCJ

The example for synchronizing, broadcasting a value from node 0 and monitoring a variable is available in Listing 1.4.

```
        PCJ.monitor("array");         // tell which variable
                                      //  to monitor

        PCJ.sync();                   // synchronize all nodes

        if (PCJ.myNode() == 0) {      // if node id equals 0
            PCJ.broadcast("array",    // then broadcast
                new double[]{         // new value of variable
                0.57721566, 1.618034, 2.7182818, 3.14159});
        }

        PCJ.waitFor("array");         // wait for modification
                                      // of variable
```

Listing 1.4. Synchronizing broadcasting and monitoring a variable

3 Scaling and Performance

In order to evaluate PCJ we have run selected Java Grande Forum Benchmark Suite tests [7] which address communication efficiency: *PingPong*, *Bcast*, *Barrier*. Tests can be run in the time limit (10 seconds) or the limit of main loop repetitions (1000000 repetitions). We have compared the results for PCJ (running on 64-bit Java Virtual Machine, Oracle version 1.7.0_01) with the results collected with MPICH2 (version: 1.4.1p1), OpenMPI (version: 1.4.2) and PGI (version: 11.3). All tests have been run on the cluster built of 64 bit Intel Xeon Processors X5660. Each processor has six cores at 2800 MHz, 24 GB RAM. The nodes are connected with the Gigabit Ethernet.

The first performed test was *PingPong*. It is based on sending an array of doubles between two nodes many times, counting all the sent data. The results are shown in Figure 1. There were differences in the buffer size (1 MB and 32 MB) and algorithm for processing of incoming data in PCJ which resulted in performance improvements (see lines *v1*, *v2* and *v3* in the graph). With the recent version of PCJ we can compete with C/C++ solutions that have been optimized for many years. The described here performance tests have been carried out using the newest version of data processing unit (*v3*) and 32 MB of buffer size.

In Figure 2 we present the result for the *Barrier* test which counts the number of barrier (synchronization) operations between all nodes in calculations. The

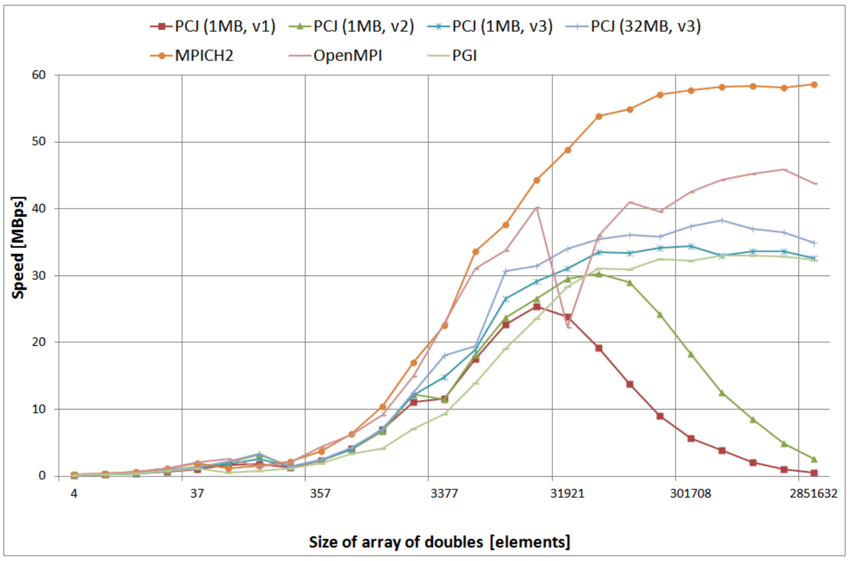


Fig. 1. Speed of PingPong with various size of array of double.

PCJ speed compared to the OpenMPI results is low and it should be improved. However, the scaling of PCJ is good which is promising and shows room for improvements.

The third performed test is called *Bcast*. It relays on broadcasting messages that consist of array of doubles of the specified size to all nodes in calculations, counting data sent by the first node. We have performed tests for the different array size. In Figure 3a there are results for the array of 21 double elements, Figure 3b presents the results for 3377 double elements and Figure 3c presents the results for 172072 double elements. There is a high correlation between the array size and maximum speed. The results for larger array sizes are very competitive in comparison to the MPI results. For small data – the PCJ speed oscillates around the 4600 B/s. and this part of PCJ should be improved. This effect has the same origin as low barrier efficiency.

4 Conclusions and future work

The PCJ library offers a new approach for the development of parallel, distributed application in Java language. It uses the newest advantages of Java and therefore can be a good base for new parallel applications. In contrast to other available solutions, it does not require any additional libraries, applications or modified version of JVM. It is noteworthy that the PCJ library has great promise to be successful in Java scientific applications.

However, the presented tests show that there are still some areas for improvements. The efficiency of sending small data can be increased. The barrier

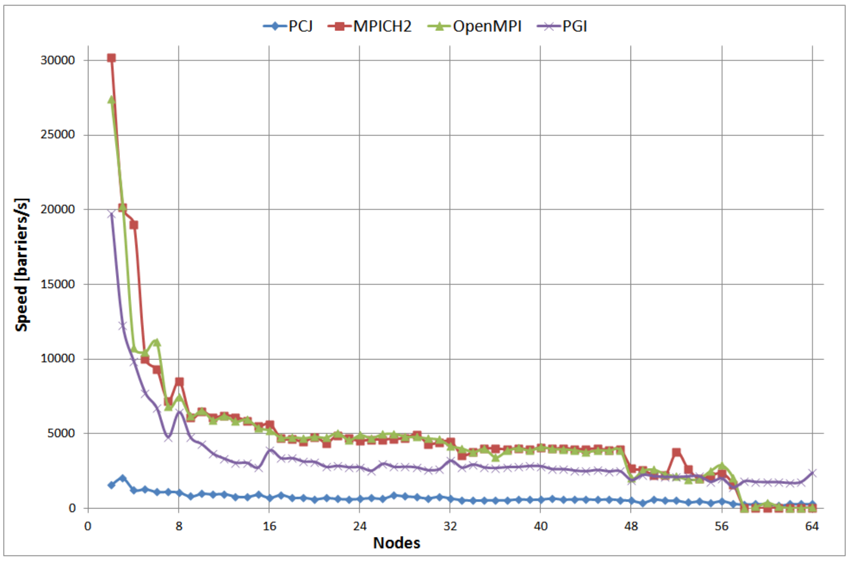


Fig. 2. Speed of performing barrier operation depending on the number of nodes.

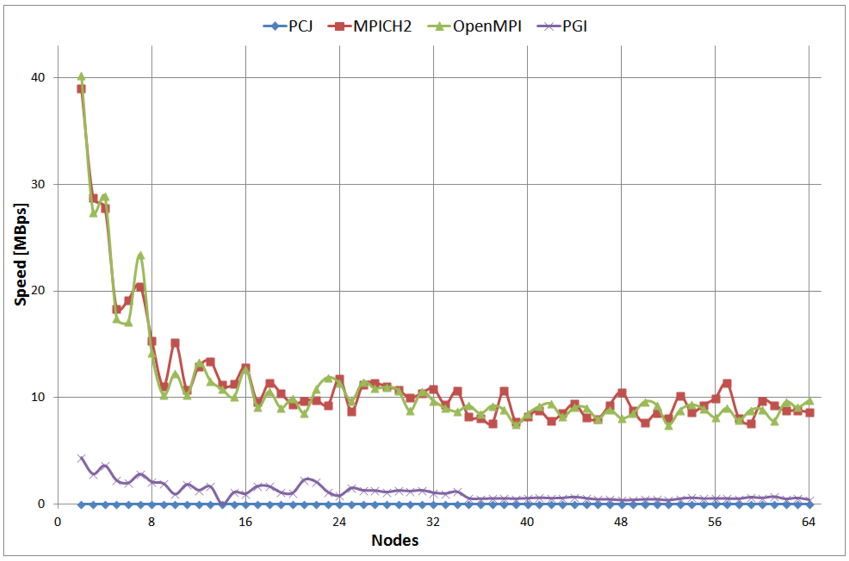
(node synchronization) speed also needs improvements. Additionally, there are no advanced techniques for the breakdown recovery and node failure handling. Such mechanisms should be also implemented in order to make PCJ a real world library for distributed and parallel application for Java language.

Acknowledgements

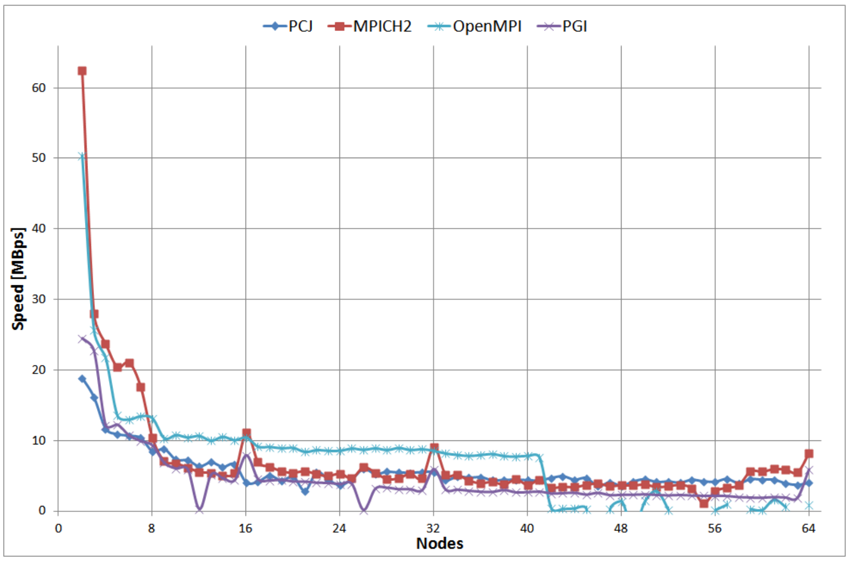
This work has been performed using PL-Grid infrastructure.

References

1. Java Platform, Standard Edition 6, Features and Enhancements http://www.oracle.com/technetwork/java/javase/features-141434.html
2. Carpenter B., Getov V., Judd G., Tony Skjellum and Geoffrey Fox. MPJ: MPI-like Message Passing for Java. Concurrency: Practice and Experience, Volume 12, Number 11. September 2000
3. Boner J. and E. Kuleshov E.: Clustering the Java virtual machine using aspect-oriented programming. In AOSD '07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development, 2007.
4. Nelson B. J.: Remote Procedure Call, Xerox Palo Alto Research Center, 1981
5. Nester Ch., Philippsen M., and Haumacher B.: A more efficient RMI for Java. In Proceedings of the ACM 1999 conference on Java Grande (JAVA '99). ACM, New York, NY, USA, 152-159. 1999



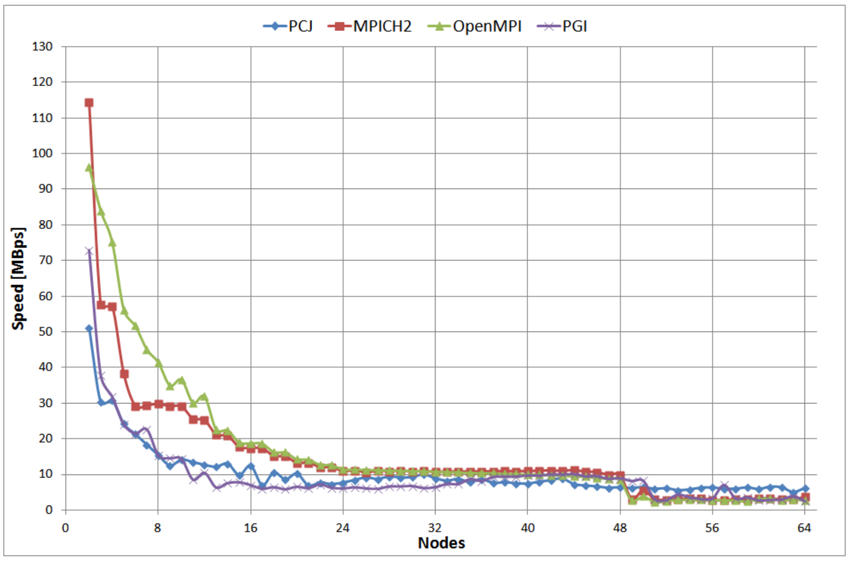
(a) 21 elements



(b) 3377 elements

Fig. 3. Speed of broadcasting array of double of various size

6. D. Mallón, G. Taboada, C. Teijeiro, J.Tourino, B. Fraguela, A. Gómez, R. Doallo, J. Mourino. Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures In: M. Ropo, J. Westerholm, J. Dongarra (Eds.) Recent Advances in



(c) 172072 elements

Fig. 3. Speed of broadcasting array of double of various size

Parallel Virtual Machine and Message Passing Interface (*Lecture Notes in Computer Science 5759)* Springer Berlin / Heidelberg 2009, pp. 174-184

7. Java Grande Project: benchmark suite http://www.epcc.ed.ac.uk/research/java-grande/
8. Java SE 7 Features and Enhancements http://www.oracle.com/technetwork/java/javase/jdk7-relnotes-418459.html