PCJ - a Java library for heterogenous parallel computing

MAREK NOWICKI
N. Copernicus University
Faculty of Mathematics
and Computer Sciene
Chopina 12, 87-100 Toruń
POLAND
faramir@mat.umk.pl

MAGDALENA RYCZKOWSKA
N. Copernicus University
Faculty of Mathematics
and Computer Sciene
Chopina 12, 87-100 Toruń
POLAND
gdama@mat.umk.pl

ŁUKASZ GÓRSKI
N. Copernicus University
Faculty of Mathematics
and Computer Sciene
Chopina 12, 87-100 Toruń
POLAND
lgorski@mat.umk.pl

MICHAŁ SZYNKIEWICZ
N. Copernicus University
Faculty of Mathematics
and Computer Sciene
Chopina 12, 87-100 Toruń
POLAND
szynkiewicz@mat.umk.pl

PIOTR BAŁA
University of Warsaw
Interdisciplinary Centre for Mathematical
and Computational Modeling
Pawińskiego 5a, 02-106 Warszawa
POLAND
bala@icm.edu.pl

Abstract: With the wide adoption of the multicore and multiprocessor systems the parallel programming became a very important element of the computer science. The programming of the multicore systems is still complicated and far to be easy. The difficulties are caused, amongst others, by the parallel tools, libraries and programming models which are not easy especially for a nonexperienced programmer. In this paper, we present PCJ - a Java library for parallel programming of heterogeneous multicore systems. The PCJ is adopting Partitioned Global Address Space paradigm which makes programming easy. We present basic functionality pf the PCJ library and its usage for parallelization of selected applications. The scalability of the genetic algorithm implementation is presented. The parallelization of the N-body algorithm implementation with PCJ is also described.

Key–Words: Parallel computing, Java, PGAS

1 Introduction

With the wide adoption of the multicore and multiprocessor systems the parallel programming is still not an easy task. The parallelization of the problem has to be performed on the algorithmic level, therefore the use of the automatic tools is not possible. The parallel algorithms are not easy to develop and require computer science knowledge in addition to the domain expertise. Once a parallel algorithm is developed it has to be implemented using suitable parallel programming tools. This task is also not trivial. The difficulties are caused, amongst others, by the parallel tools, libraries and programming models. The message passing model is difficult, the shared memory model is easier to learn but writing codes which scale well is not easy. Others, like Map-Reduce, are suitable for an only certain class of problems. Finally, the traditional languages such as FORTRAN and C/C++ are loosing popularity compared to the new ones such as Java, Scala, Python and many others.

There is also quite a potential in the PGAS languages [1] but they are not widely popularized. Most implementations are still based on the C or FORTRAN and there is a lack of widely adopted solutions for emerging languages such as Java. The PGAS programming model allows for efficient implementation of parallel algorithms.

2 PCJ Library

PCJ is a library [2, 3, 4, 5] for Java language that helps to perform parallel and distributed calculations. It is able to work on the multicore systems with the typical interconnect such as ethernet or infiniband providing users with the uniform view across nodes. The library is OpenSource (BSD license) and its source code is available at GitHub.

PCJ implements partitioned global address space model and was inspired by languages like Co-Array Fortran [6], Unified Parallel C [7] and Titanium [11].

ISBN: 978-1-61804-365-8

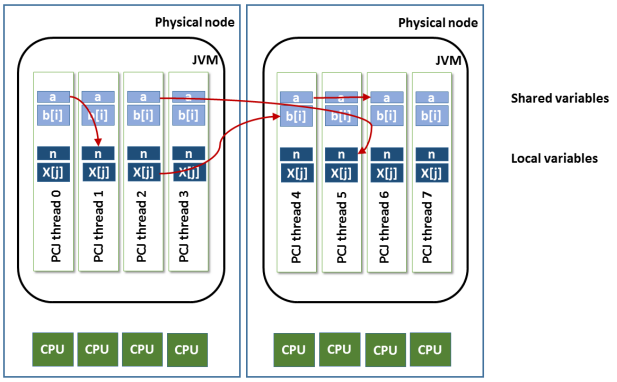

Figure 1: Schematic view of the PCJ computing model. Arrows denote possible communication using shared variables and `put()` and `get()` methods.

We put emphasis on compliance with Java standards. In contrast to the listed languages, the PCJ does not extend nor modify language syntax. The programmer does not have to use additional libraries, which are not part of the standard Java distribution.

In the PCJ, as presented in the Figure 1, each task (PCJ thread) has its own local memory and executes its own set of instructions. Variables and instructions are private to the task. Each task can access other tasks variables that have a special annotation `@Shared`. The library provides methods to perform basic operations like synchronization of tasks, get and put values in an asynchronous one-sided way.

The library offers methods for creating groups of tasks, broadcasting, and monitoring variables. The PCJ library fully complies with Java standards, therefore, the programmer does not have to use additional libraries, which are not part of the standard Java distribution. In particular, PCJ can use, implemented in Java SE 7, Sockets Direct Protocol (SDP), which increases network performance over infiniband connections.

The application using PCJ library is run as typical Java application using Java Virtual Machine (JVM). In the multinode environment one (or more) JVM has to be started on each node. PCJ library takes care on this process and allows a user to start execution on multiple nodes, running multiple threads on each node. The number of nodes and threads can be easily configured.

One instance of JVM is understood as PCJ node. In principle, it can run on a single (physical) multicore node. One PCJ node can hold many tasks (PCJ threads).This design is aligned with novel computer architectures containing hundreds or thousands of nodes, each of them built of several or even more cores.

Since PCJ application is not running within single JVM, the communication between different threads has to be realized in different manners. If communicating threads run within the same JVM, the Java concurrency mechanisms are used to synchronize and exchange information. If data exchange has to be realized between different JVM's the network communication using, for example, sockets have to be used.

3 PCJ details

The basic primitives of PGAS programming paradigm offered by the PCJ library are as follows and may be executed over all the threads of execution or only a subset forming a group:

get(int threadId, String name) - get allows to read a shared variable (tagged by name) published by another thread identified with *threadId*); both synchronous and asynchronous read with *FutureObject* is supported;

put(int threadId, String name, T newValue) - dual to get, put writes to a shared variable (tagged by name) owned by a thread identified with *threadId*; the operation is non-blocking and may return before target variable is updated;

barrier() - blocks the threads until all pass the synchronization point in the program; a two-point version of barrier that synchronizes only the selected two threads is also supported

broadcast (String name, T newValue) - broadcasts the *newValue* and writes it to each thread's shared variable tagged by name;

waitfor(String name) - due to the asynchronicity of communication primitives a measure that allows one thread to block until another changes one of its shared variables (tagged with a name) was introduced.

The presented PCJ methods allows to implement complicated parallel algorithms. The PCJ library does not provide constructs for automatic data distribution and this task has to be performed by the programmer. This allows to design data and work distribution aligned with the parallel algorithm necessary to obtain efficient and scalable implementation.

Below we present the most important implementation details of the basic PCJ functionality.

3.1 Node numbering

In the PCJ, there is one node called *Manager*. It is responsible for setting unique identifiers to the tasks, sending messages to other tasks to start calculations,

ISBN: 978-1-61804-365-8

creating groups and synchronizing all tasks in calculations. The *Manager* node has its own tasks and can execute parallel programs.

The *Manager* is the *Master* of a group of all tasks and has group identifier equals to 0. Each node has its own, unique for whole calculations, identifier. That node is called *physical id* or *node id* in short. All nodes are connected to each other and that connection is accomplished before starting a calculation. At this stage, nodes are exchanging their *physical node ids*.

At the beginning, user who wants to start using PCJ for parallel execution has to execute static method *PCJ.start()* providing information about requested StartPoint and Storage classes and list of nodes. The list of nodes is used to number PCJ nodes and PCJ threads. Every PCJ node is processing the list to localize items that contain its hostname data – items number will be used to number PCJ threads.

There is a special node, called *node0*, that is coordinating other nodes in a startup. *Node0* is a node located as the first item on the list. After processing the list, each node connects to *node0* and tells the items numbers from the list, that contains its hostname. When *node0* receives information about every node from the list, it number nodes with numbers starting from 0, increasing the number by one on each distinguished node – the number is called *physicalId*. *Node0* responses to all other nodes with their *physicalId*.

At this point every node is connected with *node0* and knows its *physicalId*. Next step is to exchange information between nodes and to connect every node with each other. To do that, *node0* is broadcasting information about each node. The broadcast is made using a balanced tree structure, where each node contains at most two children. At the beginning of the operation, the tree has only one vertex, which is *node0* – root. Broadcasted message contains information about new node in the tree: *physicalId*, parent *physicalId*, *threadsIds* and *hostname*.

When the node receives that data, it sends it down the tree, save information about a new node, and when a node is the parent of the new node, it adds it as own children. After that, the node connects to new node and sends information about itself (*physicalId* and *threadIds*). At the end, when the new node receives information from all nodes with the physical id less physical id of the new node, it sends information to *node0*, which completes initialization step.

When all nodes send information about completion of the initialization step, *node0* sends a message to start user application. Each node starts adequate number of PCJ threads using provided *StartPoint* class.

3.2 Communication

The communication between different PCJ threads has to be realized in different manners. If communicating threads run within the same JVM, the Java concurrency mechanisms can be used to synchronize and exchange information. If data exchange has to be realized between different JVM's the network communication using, for example, sockets have to be used.

The PCJ library handles both situations hiding details from the user. It distinguishes between inter- and intranode communication and pick up proper data exchange mechanism. Moreover, nodes are organized in the graph which allows to optimize global communication.

The communication between tasks running on the same JVM is performed using Java methods for thread synchronization. One should note that from the PCJ user point of view both mechanisms are transparent. The particular mechanism is used depends on the task ids involved in the communication.

PCJ uses TCP/IP protocol for the connection. The TCP protocol was chosen because of its features: it gives a reliable and ordered way of transmitting data with and error-checking mechanism over an IP network. Of course, it has some drawbacks, especially associated with performance because TCP is optimized for accurate rather than timely delivery. Usage of other protocols, like UDP, would require additional work for implementing required features: ordering out-of-order messages and retransmissions of lost or incorrect messages.

The network communication takes place between nodes and is performed using Java New IO classes (`java.nio.*`). There is one thread per node for receiving incoming data and another one for processing messages. The communication is nonblocking and uses 256 KB buffer by default [3]. The buffer size can be changed using dedicated JVM parameter.

PCJ threads can exchange data in an asynchronous way. Sending a value to another task storage is performed using the *put* method as presented in the listing 1. Since the data transfer is asynchronous the put method is accompanying with the *waitFor* statement executed by the PCJ thread receiving data. The *get* method is used for the getting value from other task storage. In these two methods, the other task is nonblocking when process puts or gets a message, but the task which initiated exchange process, blocks. There is also the *getFutureObject* method that works in fully nonblocking manner – the initializing task can check if the response is received and in the meantime do other calculations.

```
1 @Shared
2 double a;
```

ISBN: 978-1-61804-365-8

```
3
4  double c = 10.0;
5  if (PCJ.myId() == i ) {
6       PCJ.put(j, "a", c);
7  }
8  if (PCJ.myId() == j ) {
9       PCJ.waitFor"a");
10 }
```

Listing 1: Example use of the PCJ put method. The value of the variable c from PCJ thread i is broadcasted to the thread j and is stored in the shared variable a

3.3 Broadcast

Broadcasting is very similar to the *put* operation. Source PCJ thread serializes value to broadcast and sends to *node0*. *Node0* uses a tree structure to broadcast that message to all nodes. After receiving the message, it is sent down the tree, deserialized and stored into specified variable of all PCJ thread storages. An example use of the broadcast is presented in the listing 2. Please note that broadcast is asynchronous.

```
10 @Shared
11 double a;
12
13 double c = 10.0;
14 if (PCJ.myId() == 0 ) {
15      PCJ.broadcast("a", c);
16 }
```

Listing 2: Example use of the PCJ broadcast. The value of the variable c from PCJ thread 0 is broadcasted to all nodes and stored in the shared variable a

3.4 Synchronization

Synchronization is organized as follows: one task sends a proper message to the group master. When every task sends synchronization message, the group master sends an adequate message to all tasks, using the binary tree structure.

```
20      PCJ.barrier();
```

Listing 3: Example use of the PCJ synchronization of the execution performed by all PCJ threads.

The synchronization of two PCJ thread is a little more advanced functionality. Two threads, on the same node or on different nodes, can synchronize their execution as follows: one PCJ thread sends a message to another and waits for the same message to come. When the message comes before even started to wait, the execution is not suspended at all.

```
25 if (PCJ.myId() == 0 ) {
26      PCJ.barrier(5);
27 }
```

Listing 4: Example use of the PCJ synchronization. The synchronization of the execution performed by PCJ threads 0 and 5 is performed.

3.5 Fault tolerance

PCJ library provides also basic resilience mechanisms. The resilience extensions provide the programmer with the basic functionality which allows to detect node failure. For this purposes, the Java exception mechanism is used. It allows to detect execution problems associated with all intranode communication and present it to the programmer. The programmer can uptake proper actions to continue program execution. The detailed solution (algorithm) how to recover from the failure has to be decided and implemented by the programmer.

The fault-tolerance implementation relies on the assumption that node 0 never dies which is a reasonable compromise since node 0 is the place where execution control is performed. The probability of its failure is much smaller than the probability of failure of one of other nodes and can be neglected here.

The support for fault tolerance introduces up to 10% overhead when threads are communicating heavily. When a node fails, node 0 is waiting for the hearthbeat message from that node, and if it does not get it, it assumes that the node is dead.

4 Related work

There are some projects that aim to enhance Java's parallel processing capabilities. Those include Parallel Java [8] or Java Grande project [9, 10] (though they have not gained wider adoption), Titanium [11] or ProActive [12]. New developments include parallel stream implementation included in the new version of Java distribution [13]. Most of the mentioned solutions introduces extensions to the language. This requires preprocessing of the code which causes delays with the adoption to the changes in the Java. Moreover, the solutions are restricted to single JVM, therefore they can run only on the single physical node and do not scale to a large number of cores. ProActive, which allows to run an application on the relatively large number of cores suffers form performance deficiencies due to inefficient serialization mechanisms.

ISBN: 978-1-61804-365-8

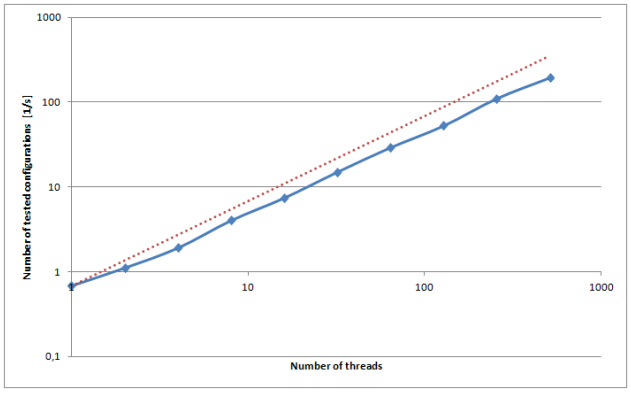

Figure 2: The performance of the differential evolution code implemented using PCJ library. The ideal scalling is presented as the doted line.

An extensive description of the related solutions together with some performance comparison can be found elsewhere [14].

5 PCJ examples

The PCJ library has been successfully used to parallelize a number of applications including typical HPC benchmarks [15] receiving HPC Challenge Award at recent Supercomputing Conference (SC 2014). Some examples can be viewed on the [3].

Recently PCJ has been used to parallelize the problem of a large graph traversing. In particular, we have implemented Graph500 benchmark and evaluated its performance. The obtained results are compared to the standard MPI implementation of the Graph500 showing similar scalability [16].

Another example is parallelization of the differential evolution on example mathematical function as well as was to fine-tune the parameters of nematode's *C. Elegans* connectome model. The results have shown that a good scalability and performance was achieved with relatively simple and easy to develop code. The simple parallelization based on the equal job distribution amongst PCJ thread was not enough since execution time of iterations performed by different threads varies. Therefore the code has been extended by the work load equalization implemented using PCJ library. In result, a scaling close to the ideal up to thousand of cores was achieved reducing simulation time from days to minutes [17] (see Fig. 2).

In this paper, we present also the performance of the MolDyn benchmark from the Java Grande Benchmark Suite implemented using PCJ library. It performs a simple N-body calculation which involve computing the motion of a number of particles (defined by a position, velocity, mass and possibly a shape). These particles move according to Newtons Laws of Motion and attract/repluse each other according to a potential function.

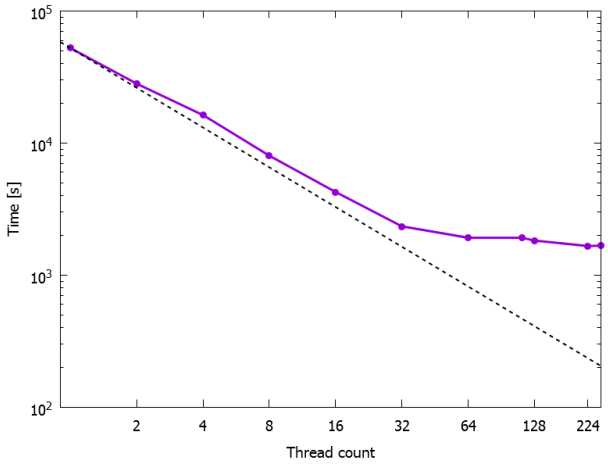

Figure 3: The performance of the MolDyn benchmark implemented using PCJ library. The ideal scalling is presented as the doted line.

The force acting on each particle is calculated from the sum of each of the forces the other particles impart on it. The total force on each particle and then apply a suitable numerical integration method to calculate the change in velocity and position of each particle over a discrete time-step.

The All-Pairs method is the simplest algorithm for calculating the force. This is an $O(N^2)$ algorithm as for N particles, the total acceleration on each particle requires $O(N)$ calculations. This method is simple to implement but it is limited by the exponential computational complexity of the algorithm.

```
30 /* move the particles and update velocities
       */
31   for (i = 0; i < mdsize; i++) {
32           one[i].domove(side);
33   }
34
35 /* compute forces */
36   rank = PCJ.myID();
37   nprocess = PCJ.thredCount();
38
39   for (i = rank; i < mdsize; i += nprocess)
       {
40             one[i].force(side, rcoff, mdsize
                   , i);
41 }
```

Listing 5: PCJ Java implementation of the MolDyn benchmark. The code for the movement of the particles and forces computation.

ISBN: 978-1-61804-365-8

In the Java Grande Benchmark implementation, atom's information is replicated on all threads and almost all operations are performed by every thread. The only parallelized part of the code is force calculation as presented in the listing 5. Each PCJ thread computes forces on the subset of particles (every *PCJ.threadCount()* atom).

The calculated partial forces have to be sum up over all threads. This task is performed by sending calculated forces to the PCJ thread 0 and than summing them up. The communication is performed in the asynchronous way and is overlapped with the calculation of the forces. Than the result is broadcasted to all PCJ threads (see listing 6) and used to calculate new positions. The broadcast statement is executed when all forces are gathered at PCJ thread 0, therefore synchronization statement can be omitted.

```
50 if (PCJ.myId() != 0) {
51       PCJ.put(0, "r_xforce", tmp_xforce,
             PCJ.myId());
52   } else {
53       PCJ.waitFor("r_xforce", PCJ.
             threadCount() - 1);
54
55       double[][] r_xforce = PCJ.getLocal("
             r_xforce");
56
57       for (int node = 1; node < PCJ.
             threadCount(); ++node) {
58            for (i = 0; i < mdsize; ++i) {
59                 tmp_xforce[i] += r_xforce[
                       node][i];
60            }
61       }
62       PCJ.broadcast("tmp_xforce",
             tmp_xforce);
63 }
```

Listing 6: The code to gather forces calculated on the different PCJ threads sum them up and distribute to the all PCJ threads. All instructions are repeated for all dimensions x y z (not shown here).

The simulation has been performed for N = 442 368 particles interacting with the Lenard-Jones potential. The periodic boundary conditions were applied and no cut-off was used. The experiments were run on the PC cluster consisting of 64 computing nodes based on the Intel Xeon E5-2697 v3 CPU (28 core each) with Infiniband interconnection. Each processor is clocked at 2.6 GHz. Every processing node has at least 64 GB of memory. Nodes are connected with Infiniband FDR and with 1Gb Ethernet. PCJ was run using Oracle's JVM v. 1.8.0. The calculations were performed using the double precision floating point arithmetic.

As presented in the Fig.3 the PCJ implementation scales well up to the 32 cores, for the higher number of cores the communication cost starts to dominate. For the larger number of cores, the calculation of the forces takes less time as it is proportional to the number of atoms allocated to the particular PCJ thread. One should note that the scalability of the PCJ implementation is similar to the original code using MPI for the communication. The resulting code is simple and contains fewer lines of parallel primitives.

6 Conclusions and future work

The obtained results show good performance and good scalability of the benchmarks and applications implemented in Java with the PCJ library. The resulting code is simple, usually contains fewer lines of code than other solutions. This is obtained thanks to the PGAS programming model and one-sided asynchronous communication implemented in the PCJ. Therefore, the parallelization is easier than in the other programming models. It allows also for easy and fast parallelization of ant data intensive processing. In this case the parallelization can be obtained by the development of simple code responsible for the data distribution. The data intensive part can be performed using existing code or even existing applications.

The communication and synchronization cost is comparable to other implementations such as MPI resulting in good performance and scalability.

The PCJ library provides additional features as support for resilience. The support for GPU through JCuda [18] is currently under tests and will be available soon.

All these features make PCJ very promising tool for parallelization large scale applications on the multicore heterogeneous systems.

Acknowledgments

This work has been performed using the PL-Grid infrastructure. Partial support from CHIST-ERA consortium and OCEAN project is acknowledged.

References:

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

[2] http://pcj.icm.edu.pl Accessed: 20.11.2015.

[3] M. Nowicki, P. Bała. Parallel computations in Java with PCJ library In: W. W. Smari and V. Zeljkovic (Eds.) *2012 International Conference on High Performance Computing and Simulation (HPCS)*, IEEE 2012 pp. 381-387

[4] M. Nowicki, P. Bała. PCJ-new approach for parallel computations in java In: P. Manninen, P. Oster (Eds.) *Applied Parallel and Scientific Computing, (LNCS 7782)*, Springer, Heidelberg (2013) pp. 115-125

[5] M. Nowicki, Ł. Górski, P. Grabarczyk, P. Bała. PCJ - Java library for high performance computing in PGAS model In: W. W. Smari and V. Zeljkovic (Eds.) *2014 International Conference on High Performance Computing and Simulation (HPCS)*, IEEE 2014 pp. 202-209

[6] R. W. Numrich, J. Reid. Co-array Fortran for parallel programming ACM SIGPLAN Fortran Forum Volume 17(2), pp. 1-31, 1998

[7] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, K. Warren. *Introduction to UPC and Language Specification* IDA Center for Computing 1999

[8] A. Kaminsky. Parallel java: A unified api for shared memory and cluster parallel programming in 100% java. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[9] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty and R. A. Davey. A Benchmark Suite for High Performance Java, *Concurrency: Practice and Experience*, 12, 375-388, 2000.

[10] Java Grande Project: benchmark suite. https://www.epcc.ed.ac.uk/research/computing/performance-characterisation/-and-benchmarking/java-grande-benchmark-suite. Accessed: 19.11.2015.

[11] P. Hilfinger, D. Bonachea, K. Datta, D. Gay, S. Graham, B. Liblit, G. Pike, J. Su and K. Yelick. *Titanium Language Reference Manual* U.C. Berkeley Tech Report, UCB/EECS-2005-15, 2005 http://titanium.cs.berkeley.edu/papers/EECS-2005-15.pdf Accessed: 3.11.2015

[12] D. Caromel, Ch. Delbé, Al. Di Costanzo, M. Leyton, et al. Proactive: an integrated platform for programming and running applications on grids and p2p systems. *Computational Methods in Science and Technology*, 12, 2006.

[13] http://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html Accessed: 2.11.2015.

[14] M. Nowicki. Opracowanie nowych metod programowania równoległego w Javie w oparciu o paradygmat PGAS (Partitioned Global Address Space). PhD thesis, University of Warsaw 2015 http://ssdnm.mimuw.edu.pl/pliki/prace-studentow/st/pliki/marek-nowicki-d.pdf Accessed: 20.11.2015.

[15] P. Luszczek, D. Bailey, J. Dongarra, J. Kepner, R. Lucas, R. Rabenseifner, D. Takahashi. The HPC Challenge (HPCC) Benchmark Suite, *SC06 Conference Tutorial*, IEEE, Tampa, Florida, November 12, 2006.

[16] M. Ryczkowska, M. Nowicki, P. Bała. The Performance Evaluation of the Java Implementation of Graph500 In: R. Wyrzzykowski (Ed.) *PPAM 2015 Lecture Notes in Computer Science* (in press)

[17] Ł. Górski, F. Rakowski, P. Bała. Parallel differential evolution in the PGAS programming model implemented with PCJ Java library In: R. Wyrzykowski (Ed.) *PPAM 2015 Lecture Notes in Computer Science* (in press)

[18] Y. Yan, M. Grossman, V. Sarkar JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA. In *Euro-Par 2009 Parallel Processing* . Springer Berlin Heidelberg 2009, pp. 887-899.