I.J. Information Technology and Computer Science, **2015, 09, 8-14**
Published Online August 2015 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijitcs.2015.09.02

Modern Platform for Parallel Algorithms Testing: Java on Intel Xeon Phi

Artur Malinowski
Gdansk University of Technology, Faculty of Electronics, Telecommunications and Informatics, Gdansk, Poland
Email: artur.malinowski@pg.gda.pl

Abstract**—**Parallel algorithms are popular method of increasing system performance. Apart from showing their properties using asymptotic analysis, proof-of-concept implementation and practical experiments are often required. In order to speed up the development and provide simple and easily accessible testing environment that enables execution of reliable experiments, the paper proposes a platform with multi-core computational accelerator: Intel Xeon Phi, and modern programming language: Java. The article includes the description of integration Java with Xeon Phi, as well as detailed information about all of the software components. Finally, the set of tests proves, that proposed platform is able to prepare reliable experiments of parallel algorithms implemented in modern programming language.

Index Terms**—**Parallel programming, parallel algorithms, testing, Java, Xeon Phi.

I. Introduction

Multi-core processors – a solution for problem with maintenance of the trend of Moore's law [1] – have become a standard. Massively parallel architecture devices (e.g. CUDA or OpenCL compatible) are more and more popular. The TOP 500 list presented in November shows, that 15% of the systems are using accelerators or co-processors, and 96% of the systems are composed with processor that include six or more cores [2]. It is clear, that modern, high performance software must be able to run concurrently, and it is crucial to focus on designing algorithms that parallelise efficiently.

Designing an algorithm is a process that consist of a several phases. At the beginning, the prediction of the running time is based on asymptotic analysis. According to M. T. Goodrich and R. Tamassia [3], such approach has following limitations: underestimates influence of constants, focuses on worse-case scenario, and appear not to be very accurate with more complicated algorithms. In order to verify deductive assumptions, implementation and a set of experiments is required.

For theoretical computer scientists and mathematicians, experiments are supposed to be a proof of concept that presents key features of their solutions, e.g. confirm asymptotic analysis results or show speed-up of parallel approach. A modern programming platform could be useful in order to ease the implementation and speed up development. Moreover, with parallel algorithms, the platform should provide mechanisms to cope with concurrency.

Testing parallel applications require also a hardware with multi-threaded execution support. Although modern PCs are equipped with multi-core processors, typical configuration contains two or four cores, which could be insufficient to show required properties, e.g. scalability. Another possibility is to use accelerators, but it is often connected with low level programming platform (e.g. CUDA/OpenCL on GPGPU or C/C++ on Intel Xeon Phi). Distributed platforms (e.g clusters, grids) are not considered due to higher complexity of software implementation and less accessibility.

The paper aims to provide both a software and a hardware setup, that allows to implement parallel algorithms easily with high level, modern programming platform, and execute tests in environment, that will be sufficient to show benefits of running algorithm in parallel. Java, very popular platform nowadays, is proposed as a programming language, while Intel Xeon Phi is a device that has been chosen as a parallel processor.

The paper continues as follows: Section 2 provides motivations that resulted in the idea of this research. Section 3 contains related work that justifies selection of the platform components. Section 4 describes the steps of making Java and Xeon Phi work together. Section 5 presents experiments that proves correctness of environment configuration and ability of running implementation on many threads in parallel. Finally, Section 6 summarizes the research and proposes future work.

II. Motivations

The need for software and hardware platform for parallel algorithms testing comes from author's research that involves designing sophisticated caching solutions. The whole problem is split into independent parts, each part requires selection of best approach. While it is often difficult to compare ideas using only mathematical analysis, the project team decided, that it would be useful to prepare proof of concept implementations and verify it under conditions similar to target platform.

III. Related Work

A. Parallel Algorithms Testing

To the best knowledge of the author, parallel algorithms testing itself is not the main issue of any research. However, many works connected to specific algorithms include a topic of testing.

The simplest execution environment is based on a single computing node. A.J. Umbarkara, M.S. Joshib, and Wei-Chiang Hongd proposed multithreaded implementation of Dual Population Genetic Algorithm in Java [4]. Efficient speedup of the solution was claimed, but the algorithm was tested only on single, dual core CPU. Another example, performance of graph algorithms proposed by G.M. Slota, was verified with 16 cores CPU [5]. It is clear, that with the increasing number of concurrent threads, the speedup tests become much more reliable. Moreover, tests executed on single CPU cannot easily exceed limit of about 16 concurrent threads nowadays.

Another popular, but much more complicated testbed environment is a cluster of nodes. Many parallel algorithms were tested on a cluster, e.g. tsunami wave modeling (Supercomputer K with more than 80,000 processors) [6], BLAST – Basic Local Alignment Search Tool (40 nodes cluster) [7], or RainbowCrack (supercomputer SHARCNET, 90 nodes) [8]. However, usage for individual is limited because of the high price– usually only institutions, e.g. universities or big research and development departments can afford such equipment. What is more, programming on a cluster often requires additional effort to cope with distant communication, data distribution, shared memory etc.

The third idea of proving advantages of parallel approach is implementation on the computation accelerator. GPGPU (General-Purpose computing on Graphics Processing Unit) is a typical instance. Example implementations: two-list algorithm for the subset-sum problem [9] or protein structure similarity search engine [10] illustrate the approach– parallel algorithms execution on GPU requires adjusting to the specific architecture.

B. Modern Programming Platform

Popularity of a programming language is an important factor – it is usually connected with an ease of learn and large number of libraries developer can use. According to TIOBE Index for March 2015 [11], Java is the second most popular programming language. Considering C, Objective-C and C++ as not modern, the closest competitor, C#, is about 3 times less popular. Moreover, Java is attractive for parallel processing. Programmers can use FastMPJ – message-passing library similar to well-established MPI [12] or OpenMP-like directives [13]. An example of parallel, multithreaded Java library is Parallel Colt proposed by P. Wendykier and J.G. Nagy [14]. The aim of the library is to speed up scientific computing and image processing.

Nowadays, programming using computational accelerators is not limited to C/C++ based languages like Nvidia CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language). J. Docampo et. al. evaluated several libraries that enable the use of Java in GPGPU [15]. One of the tested solutions, Aparapi framework, was reported to keep high productivity, some additional effort of knowing the tool is required, moreover, GPU computations are enclosed in special routine with limited operations allowed (what limits usage of external libraries).

Another accelerator, which is based on many x86_64 cores, Intel Xeon Phi, is reported not to have Java support [16].

C. Cost Of The Platform

One of the main factor that decides about availability of the platform for individuals is the price of hardware components. At the time of writing the paper, there is a possibility to obtain a processor with maximum of 16 cores. Motherboards with dual or quad sockets allow to install more processors in a single node, however, it increases the price significantly. According to Table 1, building a platform based on GPU seems to be the best option, however, GPU architecture requires specific programming techniques that makes development more difficult. Xeon Phi accelerator has relatively small price for a single multi-purpose core. Cluster environments are not included because of much higher price compared to a single node.

Table 1. Cost of example components included in testing platforms

Type	Example	Price
CPU	Intel Xeon E3-1220 (4 cores)	193 USD [17]
CPU	AMD Opteron 6274 (16 cores)	746.99 USD [18]
GPU	Nvidia GeForce GTX 750 (512 CUDA cores)	150.00 USD [19]
Xeon Phi	Intel Xeon Phi Coprocessor 3120A (57 cores)	1695 USD [20]

IV. Contribution

A proposal of parallel algorithms testing platform consists of a Java as a software part, and Intel Xeon Phi as a hardware. Java language is chosen because of popularity, productivity, and the amount of additional libraries programmer can use. The main advantage of Xeon Phi accelerator is providing multi-threaded execution environment with reasonable price for a single processing core. As stated before, Java is not an officially supported programming language on the accelerator, however, the architecture based on x86_64 processors seems to be more flexible than GPU architecture.

Typically, Java applications are compiled to a bytecode, and then executed on Java virtual machine (JVM). Compilation could be done with standard tools on host system, while the JVM is needed to be ported on Xeon Phi. JVM Specification is publicly available [21] and many implementations were proposed, however, only a few became popular. Two of them were considered as a part of a solution: HotSpot (primary reference implementation, currently managed by Oracle [22]) and JamVM (lightweight and small implementation, that used to be the default Java environemt on Ubuntu ARM [23][24]). Both implementations are platform dependable, on the other hand, both are also well prepared to be built without assembler using libffi library. Finally, JamVM was chosen because of lower number of compilation problems.

JVM execution includes native calling (e.g. related with memory management or thread support), which is often related to hardware architecture. In order to provide an interface between native call and a hardware layer, libffi library is commonly used. Libffi [25] provides high level programming interface that allows to build application on the top of it without architecture specific code; unfortunately, the library does not support Xeon Phi.

Proposed platform includes libffi support for Xeon Phi. Single accelerator core is compatible with x86_64 architecture, but it does not enable SSE extension, widely used in libffi x86_64 routine implementation. The solution is based on replacement of SSE instructions and XMM registers using AVX-512 instructions and ZMM registers, that are available on Xeon Phi [26].

The third element of Xeon Phi JVM compilation is Java Class Library – set of libraries that could be called by application on JVM [27]. Although JamVM supports both most popular implementations, GNU Classpath and OpenJDK, GNU Classpath was used due to better compatibility with JamVM.

All of the software used to prepare a platform, i.e. JamVM, Libffi, Gnu Classpath, is open source.

It should be also noted, that the only purpose of proposed technology stack is testing of algorithms properties. The platform does not offer fast or efficient environment that can be useful in performing complex computations. One reason is Java compilers, that generate bytecode that is very efficient on modern x64 SSE processors, however, it often makes impossible to use Xeon Phi optimization techniques like loop unrolling. Another issue is connected to lack of JVMs optimization for vector instructions commonly used in accelerators. Although it is not in the main scope of this research, simple tests confirm, that Java applications running on proposed platform are slower than executed on any modern PC.

V. Experiments

Experiments consist of the execution of selected parallel algorithms with various number of threads: quicksort, calculation of PI using Monte Carlo method, Fast Fourier transform, discrete cosine transform, and simulation of distributed cache algorithm that was the motivation for this paper described in section II. For each execution, time and speedup are presented. Speedup is defined by the following formula:

$$S = \frac{T_{sequential}}{T_{\text{parallel}}}$$

where T is a time of execution. The aim of the experiments is to prove, that proposed platform is capable of executing an algorithm in parallel in order to provide an information of the speedup according to number of concurrent threads.

A. Testbed Environment

All the tests are performed on single Intel Xeon Phi Coprocessor 5100 accelerator, parameters of testing platform are presented in Table 2.

Table 2. Intel Xeon Phi Coprocessor 5100 specification

Memory size	8GB DDR5
L2 cache size	30MB
Number of cores	60
Base processor frequency	1.1GHz

It is expected, that with 60 independent cores a parallel algorithm execution should scale smoothly up to tens of simultaneously executed threads.

B. Quicksort

First test is based on parallel version of quicksort implementation. The input was an array of 1500 randomly selected complex numbers, result time includes the time of data generation. Table 3 and Figure 1 illustrates near-linear speedup at a reasonable level up to 30 threads. After exceeding 30 threads, speedup stops at the level of 23, what is probably caused by increasing memory usage. Although performance drop was expected after exceeding 60 threads, it is not observed - further analysis confirmed, that the overhead for management of threads is not significant in uncomplicated implementations.

Table 3. Quicksort: Average execution times and speedup

Number of threads	Execution time	Speedup
1	43.04s	1.00
2	22.23s	1.94
4	11.41s	3.77
8	6.09s	7.07
15	3.4s	12.66
20	2.58s	16.68
30	2.00s	21.52
40	1.95s	22.07
50	1.90s	22.65
60	1.91s	22.53
70	1.90s	22.65
80	1.90s	22.65

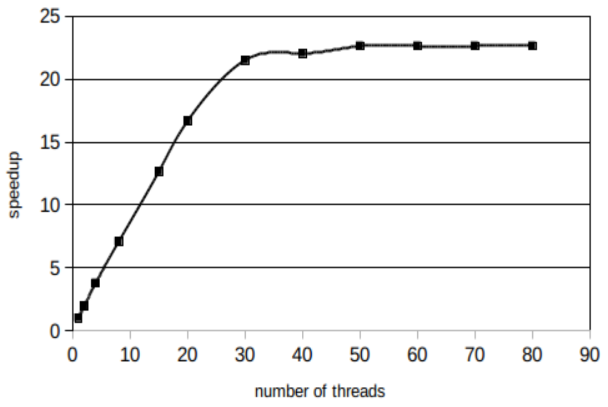

Fig. 1. Speedup of quicksort implementation by number of threads

C. Calculation of PI using Monte Carlo Method

Second experiment is based on an uncomplicated algorithm, that can be easily implement in parallel – calculation of PI using Monte Carlo method. The implementation includes not only a calculation part, but also a simple random number generator (separate instance is created for each thread). Monte Carlo methods are iterative, and the test is based on 10^7 iterations. Almost linear speedup up to 60 threads shown in Table 4 and Figure 2 is a result of simplicity of implementation and relatively high independence of calculation parts. Similarly to the previous experiment, uncomplicated algorithm and straightforward implementation caused no significant performance drop observed in scenarios that included more than 60 threads.

Table 4. Calculation of PI using Monte Carlo Method: Average execution times and speedup

Number of threads	Execution time	Speedup
1	30.49s	1.00
2	15.29s	1.99
4	7.66s	3.98
8	4.24s	7.19
15	2.83s	10.77
20	2.20s	13.86
30	1.50s	20.33
40	1.29s	23.64
50	1.10s	27.72
60	0.99s	30.80
70	1.01s	30.19
80	0.98s	31.11

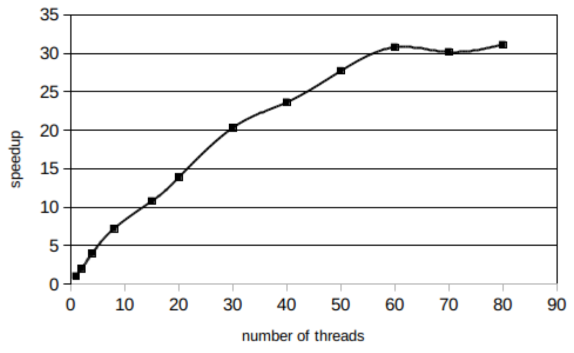

Fig. 2. Speedup of PI calculation with Monte Carlo by number of threads

D. Fast Fourier Transform

Parallel implementation of fast Fourier transform utilized in this test comes from Parallel Colt library [14]. Library was not modified in order to show, that applications can be easily build with any existing Java library. Moreover, previous Parallel Colt performance tests was performed using maximum of 8 concurrent threads; proposed platform allows for testing it with more threads.

The input data was a matrix of 2000x2000 size filled with randomly generated complex numbers. Time of memory allocation and preparing the data is not included into results.

The experiment differs from two previous because of higher complexity of implementation. Table 5 and Figure 3 presents almost linear speedup up to 40 threads. Speedup between 40 and 60 threads is constant – the gain from greater number of concurrent threads is balanced with an overhead for thread management (i.e. creation and termination of threads, communication, shared resources access). When exceeding 60 – the number of cores in accelerator – the overhead causes significant drop in performance.

Table 5. Fast Fourier transform: Average execution times and speedup

Number of threads	Execution time	Speedup
1	63.04s	1.00
2	32.56s	1.94
4	17.69s	3.56
8	9.46s	6.66
15	4.99s	12.63
20	3.61s	17.46
30	2.47s	25.52
40	1.72s	36.65
50	1.72s	36.65
60	1.76s	35.82
70	3.54s	17.81
80	3.94s	16.00

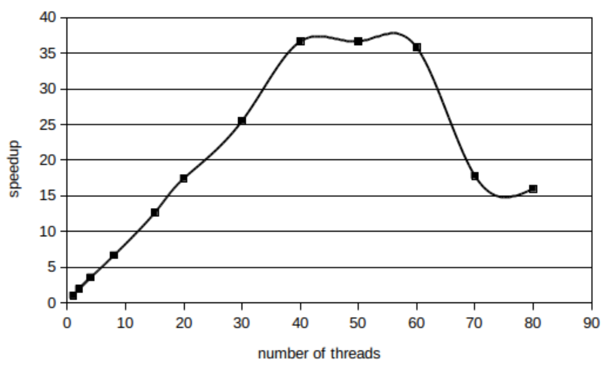

Fig. 3. Speedup of fast Fourier transform by number of threads

E. Discrete Cosine Transform

The test is based on discrete cosine transform, algorithm similar to fast Fourier transform. The implementation also comes from Parallel Colt library. The input was a randomly generated 2000x2000 matrix of double numbers. Time of memory allocation and preparing the data is not included into results.

Results presented in Table 6 and Figure 4 are similar to the results of previous test: near linear speedup can be observed up to 40 threads, constant speedup between 40 and 60 threads. and performance drop When the number of threads exceeds the number of physical cores in the processor.

Table 6. Discrete cosine transform: Average execution times and speedup

Number of threads	Execution time	Speedup
1	85.02s	1.00
2	50.14s	1.70
4	26.01s	3.27
8	13.13s	6.48
15	6.84s	12.43
20	4.71s	18.05
30	3.45s	24.64
40	2.31s	36.81
50	2.33s	36.49
60	2.33s	36.49
70	4.95s	17.18
80	5.23s	16.26

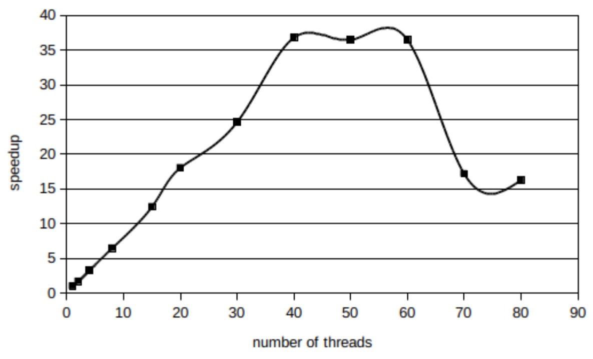

Fig. 4. Speedup of discrete cosine transform by number of threads

F. Distributed Cache

The last experiment is a use case based on a distributed cache solution mentioned in section II. Because the whole architecture is composed of several entities with complex dependencies, it is difficult to predict scalability using theoretical analysis. In order to verify whether solution design is expected to fulfills requirements, a simplified Java implementation was created (to provide high performance of final implementation, C was selected as a programming language, and Message Passing Interface library as a standard of distributed processing). Although the first solution performed well in most of test cases, it turned out to have performance problems under heavy load – result presented in Table 7 and Figure 5 show no speedup with increasing number of threads. With proposed testing platform, defects of the algorithms was easily detectable at the early stage of the project.

Table 7. Distributed cache, first solution: Average execution times and speedup

Number of threads	Execution time	Speedup
1	6.28s	1.00
2	6.28s	1.00
4	6.35s	0.99
8	6.59s	0.95
15	6.84s	0.92
20	6.96s	0.90
30	7.46s	0.84
40	7.73s	0.81
50	7.71s	0.81
60	8.38s	0.75
70	9.06s	0.69
80	9.34s	0.67

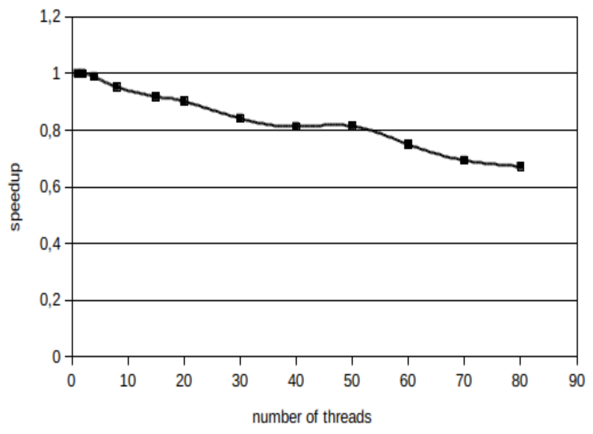

Fig. 5. Speedup of distributed cache (first solution) by number of threads

The problems with first distributed cache algorithm was caused by a bottleneck – entity responsible for thread management was prone to overloading. After redesigning of the algorithm and changes in implementation, new algorithm version fulfilled the scalability requirements. Modern programming language Java properties such as modularity and productivity allowed to prepare new implementation fast and easily. Table 8 and Figure 6 show the result of the test under heavy load: algorithm scales up linearly up to 40 threads, performance does not decrease rapidly when the number of threads exceeds 60 (number of physical cores).

Table 8. Distributed cache, second solution: Average execution times and speedup

Number of threads	Execution time	Speedup
1	310.19s	1.00
2	155.28s	2.00
4	78.47s	3.95
8	39.15s	7.92
15	21.53s	14.41
20	16.45s	18.86
30	10.90s	28.46
40	8.34s	37.19
50	7.77s	39.92
60	7.42s	41.80
70	7.35s	42.20
80	7.46s	41.58

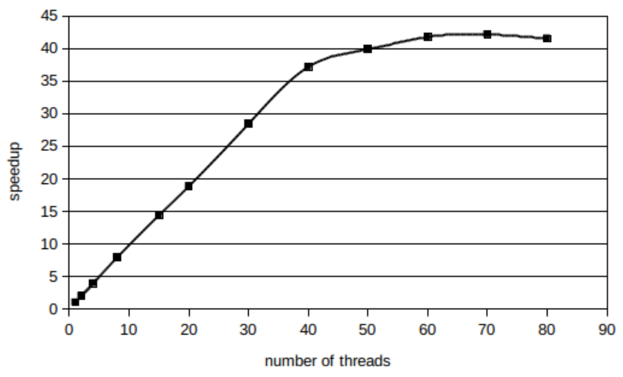

Fig. 6. Speedup of distributed cache (second solution) by number of threads

VI. Conclusion And Future Work

A. Conclusion

The paper proposes software and hardware platform for parallel algorithm testing. The platform gives a possibility to run modern programming language implementation on about 40-60 concurrent threads on a single node machine. Software technology stack is based on Java – well established, productive, easy to learn and considered as quite modern programming environment. Selection of Intel Xeon Phi Accelerator as a hardware provides concurrent, multithreaded execution platform. Although Java is not officially supported on the accelerator, the paper provides a detailed description of extending Xeon Phi with Java support.

Experiments confirmed, that the proposed environment is useful to test parallel algorithms. Examples show, that testing of speedup is reliable up to tens of concurrent threads, depending on the type of algorithm.

B. Future Work

First future task is connected with publishing a manual and set of scripts that will allow to build the whole software platform easily.

Moreover, an additional effort will be put in order to verify whether Java on Intel Xeon Phi could be treated not only as a tool to obtain performance tests result, but also as a fast and efficient computation platform. The research will probably include major changes in Java compiler optimizations and particular JVM implementation.

References

[1] J. Chen, "Analysis of Moore's Law on Intel Processors". Proceedings of the 2013 International Conference on Electrical and Information Technologies for Rail Transportation (EITRT2013), vol II. Springer, 2014, 391-400.

[2] Top500. November 2014, http://www.top500.org/lists/2014/11/, accessed: Mar. 16, 2015.

[3] M. T. Goodrich, R. Tamassia, "Algorithm Design: Foundations, Analysis and Internet Examples", John Wiley & Sons, Inc., 2009.
[4] A. J. Umbarkar, M. S. Joshi, Wei-Chiang Hong, "Multithreaded Parallel Dual Population Genetic Algorithm (MPDPGA) for unconstrained function optimizations on multi-core system", Applied Mathematics and Computation, vol. 243, 2014, 936-949.
[5] G. M. Slota, "BFS and Coloring-Based Parallel Algorithms for Strongly Connected Components and Related Problems", Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, 2014, 550-559.
[6] T. Baba, N. Takahashi, Y. Kaneda, K. Ando, D. Matsuoka, T. Kato, "Parallel Implementation of Dispersive Tsunami Wave Modeling with a Nesting Algorithm for the 2011 Tohoku Tsunami", Pure and Applied Geophysics, Springer, 2015.
[7] N. Dhankher, O. P. Gupta, "Parallel Implementation & Performance Evaluation of Blast Algorithm on Linux Cluster", International Journal of Computer Science and Information Technologies, vol. 5 (3), 2014, 4818-4820.
[8] E. R. Sykes, W. Skoczen, "An improved parallel implementation of RainbowCrack using MPI", Journal of Computational Science, vol 5, 2014, 536-541.
[9] Lanjun Wan, Kenli Li, Jing Liu, Keqin Li, "GPU implementation of a parallel two-list algorithm for the subset-sum problem", Concurrency and Computation: Practice and Experience, vol. 27, 2015, 119-145.
[10] D. Mrozek, M. Brożek, B. Małysiak-Mrozek, "Parallel implementation of 3D protein structure similarity searches using a GPU and the CUDA", Journal of Molecular Modeling, Springer, 2014.
[11] TIOBE Software Corporation, "TIOBE Index for March 2015",http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, accessed: Apr. 2, 2015.
[12] R. R. Exposito, S. Ramos, G. L. Taboada, J. Tourino, R. Doallo, "FastMPJ: a scalable and efficient Java message-passing library", Cluster Computing – The Journal of Networks, Software Tools and Applications, vol. 17, Springer, 2014, 31-1050.
[13] M. E. Kambites, J. Obdrzalek, J. M. Bull, "An OpenMP-like interface for parallel programming in Java", Concurrency and Computation: Practice and Experience, vol. 13, 2001, 793-814.
[14] P. Wendykier, J. G. Nagy, "Parallel Colt: A High-Performance Java Library for Scientific Computing and Image Processing", ACM Transactions on Mathematical Software (TOMS), vol. 37, 2010.
[15] J. Docampo, S. Ramos, G. L. Taboada, R. R. Exposito, J. Tourino, R. Doallo, "Evaluation of Java for General Purpose GPU Computing", Advanced Information Networking and Applications Workshops (WAINA), 2013 27th International Conference on, 2013, 1398-1404.
[16] Intel Corporation, "Intel® Xeon Phi™ Coprocessor February Developer Webinar Q&A Responses", https://software.intel.com/en-us/articles/intelr-xeon-phitm-coprocessor-february-developer-webinar-qa-responses, accessed: Oct. 30, 2014.
[17] Intel Corporation, "Intel® Xeon® Processor E3-1220 v3", http://ark.intel.com/products/75052/Intel-Xeon-Processor-E3-1220-v3-8M-Cache-3_10-GHz, accessed: Apr. 6, 2015.
[18] AMD Corporation, "AMD Opteron 6200 Series Processor 6274", http://shop.amd.com/en-us/business/processors/ecxOffUS798864, accessed: Apr. 6, 2015.
[19] Nvidia Corporation, "GeForce GTX 750 Specyfication", http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-750/specifications, accessed: Apr. 6, 2015.
[20] Intel Corporation, "Intel® Xeon® Coprocessor 3120A" http://ark.intel.com/products/75797/Intel-Xeon-Phi-Coprocessor-3120A-6GB-1_100-GHz-57-core, accessed: Apr. 6, 2015.
[21] T. Lindholm, F. Yellin, "Java Virtual Machine Specification", Addison-Wesley Longman Publishing Co., Inc., 1999.
[22] Oracle Corporation, "Java SE HotSpot at a Glance", http://www.oracle.com/technetwork/articles/javase/index-jsp-136373.html, accessed Mar. 19, 2015.
[23] R. Lougher, "JamVM – A compact Java Virtual Machine", http://jamvm.sourceforge.net/, accessed Mar. 19, 2015.
[24] Open Source Community, "Openjdk-6 package", https://launchpad.net/ubuntu/+source/openjdk-6/6b23~pre4-0ubuntu1, accessed Mar. 19, 2015.
[25] Open Source Community, "Libffi. A Portable Foreign Function Interface Library", https://sourceware.org/libffi/, accessed Mar. 19, 2015.
[26] Intel Corporation, "Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual", https://software.intel.com/sites/default/files/forum/278102/327364001en.pdf, accessed Mar. 19, 2015.
[27] Open Source Community, "GNU Classpath", http://www.gnu.org/software/classpath/, accessed Apr. 6, 2015.

Author's Profile

Artur Malinowski: PhD student at the Gdansk University of Technology, Department of Computer Architecture, Faculty of Electronics, Telecommunications and Informatics; received engineering degree in 2013 and master's degree in 2014. Research interests: high performance computing and modern programming platforms.