Performance Evaluation of Java/PCJ Implementation of Parallel Algorithms on the Cloud

Marek Nowicki\(^1\), Łukasz Górski\(^2\), and Piotr Bała\(^2\)

\(^1\) Faculty of Mathematics and Computer Science,
Nicolaus Copernicus University in Toruń, Torun, Poland
faramir@mat.umk.pl

\(^2\) Interdisciplinary Centre for Mathematical and Computational Modeling,
University of Warsaw, Warsaw, Poland
\{lgorski, bala\}@icm.edu.pl

Abstract. Cloud resources are more often used for large scale computing and data processing. However, the usage of the cloud is different than traditional High-Performance Computing (HPC) systems and both algorithms and codes have to be adjusted. This work is often time-consuming and performance is not guaranteed. To address this problem we have developed the PCJ library (Parallel Computing in Java), a novel tool for scalable high-performance computing and big data processing in Java. In this paper, we present a performance evaluation of parallel applications implemented in Java using the PCJ library. The performance evaluation is based on the examples of highly scalable applications that run on the traditional HPC system and Amazon AWS Cloud. For the cloud, we have used Intel x86 and ARM processors running Java codes without changing any line of the program code and without the need for time-consuming recompilation. Presented applications have been parallelized using the PGAS programming model and its realization in the PCJ library. Our results prove that the PCJ library, due to its performance and ability to create simple portable code, has great promise to be successful for the parallelization of various applications and run them on the cloud with a similar performance as for HPC systems.

Keywords: Cloud computing · Parallel computing · Performance evaluation · Java · PCJ · HPC · Cloud

1 Introduction

Cloud computing, despite its quite long presence and stable position on the market differs from High Performance Computing (HPC). While both target large scale workloads with scalability being the main factor, this target is achieved differently.
HPC is concerned with the vertical scalability. Typical workflow may be conceived as a single application processing extremely large datasets in parallel, with data being shared by the compute nodes. Therefore network performance becomes a limiting factor, with advancements in the networking technology being crucial for the successful parallelization of workloads [38]. Cloud computing can be connected with horizontal scalability. Here a problem can be effortlessly divided into a number of independent sub-problems, each susceptible to being solved in a self-contained manner. Multiple instances of a given application run in concert, with their number being dynamically adjusted to account for the current system’s load [38].

HPC and cloud computing, therefore, try to achieve a different type of scalability. To achieve their aim, both techniques use their types of optimized hardware. The feasibility of choosing one or the other depends on the particular application’s requirements. However, there is increasing interest in finding solutions that bridge HPC and cloud computing. New solutions have to be open to new programming languages utilized, *inter alia*, by Big Data and Artificial Intelligence communities as well. To address this problem, we have focused on the usability of Java for HPC applications. Usage of well-established industrial language opens the field for easy integration of both workload types. The PCJ library (Parallel Computing in Java) [25] fully complies with core Java standards, therefore, the programmer does not have to use additional libraries, which are not part of the standard Java distribution. Thus user is free from the burden of installing (properly versioned) dependencies, as the library is a single self-contained *jar* file, which can be easily dropped into the *classpath*.

In the previous works, we have shown that the PCJ library allows for the easy and feasible development of computational applications as well as Big Data and AI processing running on supercomputers or clusters. The performance comparison with the C/MPI based codes has been presented in previous papers [22,26]. The extensive comparison with Java-based solutions including APGAS (Java implementation of X10 language) has been also performed [27,31].

In this paper, we contribute by focusing on the performance of selected applications run on the Amazon Web Services (AWS) Elastic Compute Cloud (EC2) using both Intel and ARM architectures. The ultimate aim is to show the feasibility of using Java and the PCJ library for the development of parallel applications run on the cloud. The results have been compared to the state of the art HPC system, namely Cray XC40 equipped with the Intel processors.

Intel architecture is available at AWS since the beginning, the ARM architecture has been added at the end of 2018 [2]. In this case dedicated processors designed for AWS are used, called AWS Graviton, that are modified ARM Cortex-A72 processor [42]. AWS continues to develop its own ARM processors and recently announced AWS Graviton2 processors to be available in new instance types [3].
2 Related Work

The interest in joining the traditional HPC computing and exploiting the potential of the cloud has already been growing for some time. It has been noted that while the HPC paradigm offers great computing capabilities, the cloud offers elasticity and dynamic allocation of resources on a scale unseen before. There is a growing influx of competent engineers that specialize in DevOps and are well-acquainted with microservices, virtualization, and the containerization of software (with tools like Kubernetes or Docker). On the other hand, traditional HPC was concerned mainly with languages like C or Fortran, which are now decreasing in popularity. Moreover, the divergence between traditional HPC workloads and emerging new ones could already have been observed in the case of Big Data processing or Artificial Intelligence applications. They are implemented in languages like Java or Scala which, up to now, were out of interest of the HPC community.

MPI, which is the basic parallelization library, is also criticized because of the complicated API and difficulty in programming. Users are looking for easy to learn, yet feasible and scalable tools more aligned with popular programming languages such as Java or Python. They would like to develop applications using workstations or laptops and then easily move them to large systems including HPC-based peta- and exascale ones or to deploy them on a cloud.

Cloud has been already recognized as a way for provisioning tailored-to-fit computational resources for medium-sized HPC workloads [15]. It was found that it is especially well-suited for communication-intensive applications (up to low processor count) and embarrassingly parallel ones (up to high processor count) [14].

PCJ library deviates from the standard scientific message-passing paradigm and instead opts to use the PGAS model. The model aims to present the programmer with an abstraction of unified memory view, even when in fact it is distributed among the computer nodes. The PGAS model is supported by libraries such as SHMEM [10], or Global Arrays [20] as well as by specialized languages or dialects, such as UPC [6] (C-based), Fortran [33], X10 [9] or Chapel [8]. PGAS systems differ in the way the global namespace is organized.

The perspective language is, amongst others, Java due to its popularity and portability. Java has good support of threads since the beginning. The parallelization tools available for Java include threads and Java Concurrency which have been introduced in Java SE 5 and improved in Java SE 6 [29]. There are also solutions based on various implementations of the MPI library [7, 40], distributed Java Virtual Machine [4] and solutions based on Remote Method Invocation (RMI) [19]. We should also mention solutions motivated by the PGAS approach represented by Titanium [41]. Titanium defines new language constructs and has to use a dedicated compiler which makes it difficult to follow recent changes in Java language.

The APGAS library for Java [37] adds asynchronism to the PGAS model by adopting a task-based approach. The parallelization and distribution concepts are the same as those of IBM’s parallel language X10. Program execution starts
with a single task. Later, any task can spawn any number of child tasks dynamically synchronously or asynchronously. In either case, the programmer must specify an execution place for each task. Inside each place, tasks are automatically scheduled to workers. The place-internal scheduler is implemented with Java’s Fork/Join-Pool [30].

Yet another solution for writing high-performance HPC applications, that incorporates the PGAS programming paradigm is PCJ library described in next section.

3 The PCJ Library

PCJ [1,25] is an open-source Java library that does not require any language extensions or special compiler. The user has to download the single jar file (or use build automation tool with dependency resolvers like Maven or Gradle) on any system with Java installed.

The PCJ library is designed to support the application running on the multicore, multinode systems. The programmers are provided with the PCJ class with a set of methods to implement necessary parallel constructs. All technical details like threads administration, communication, and network programming are hidden from the programmers. The intranode communication is implemented using the Java Concurrency mechanism. The object sent from one thread to another has to be serialized, and then deserialized on the other thread and stored in memory. This way of cloning data is safe, as the data is deeply copied – the other thread has its copy of data and can use it independently. The communication between nodes uses standard network communication with sockets. The network communication is performed using Java New IO classes (i.e. java.nio.*). The details of the algorithms used to implement PCJ communication are described in the [25].

The PCJ library provides necessary tools for PGAS programming including threads numbering, data transfer and threads synchronization. The communication is one-sided and asynchronous which makes programming easy and less error-prone. With a relatively simple set of methods, programmers can easily implement data and work partitioning best suited to the problem they are solving. Instead of modifying the problem to fit the given programming model, the user optimally implements his algorithm using the PCJ library as a tool to expose parallelism.

The PCJ library has won the HPC Challenge award in 2014 and has already been used for the parallelization of multiple applications. A good example is a communication-intensive graph search from the Graph500 test suite. The PCJ implementation scales well and outperforms Hadoop implementation by the factor of 100 [28,34]. PCJ library was also used to develop code for the evolutionary algorithm which has been used to find a minimum of a simple function as defined in the CEC’14 Benchmark Suite [17]. Recent examples of PCJ usage include parallelization of the sequence alignment [23,24]. PCJ library allowed for the easy implementation of the dynamic load balancing for multiple NCBI-BLAST instances spanned over multiple nodes. The obtained performance was
at least 2 times higher than for the implementations based on the static work
distribution. Another example that uses the PCJ library is an application for
calculating the parameters of *C. Elegans* connectome model [13,32] that uses
differential evolution algorithm.

4 Experimental Setup

The cloud used is the Amazon AWS EC2. We have used the instances in the
Europe (Frankfurt) region (*eu-central-1*).

We have used *t2.xlarge* instances equipped with Intel(R) Xeon(R) CPU E5-
2686 v4 processors (2.3 GHz). Each instance consists of 4 vCPUs, 16 GB RAM.
The network performance is described as *moderate*, what translates into about
1 Gbps network bandwidth. The instances’ operating system is Amazon Linux 2
AMI 2.0.20200207.1 x86.64 HVM gp2 with additionally installed OpenJDK Run-
time Environment Corretto-11.0.6.10.1 implementation of JVM.

Due to the increasing popularity of ARM processors, we have also used
*a1.xlarge* instances equipped with AWS Graviton Processors (2.3 GHz) featuring
64-bit ARM Neoverse cores designed by AWS and optimized for performance and
cost. Each instance consists of 4 vCPUs, 8 GB RAM with EBS disk storage. The
network bandwidth is up to 10 Gbps. The instances’ operating system is Amazon Linux 2
LTS Arm64 AMI 2.0.20200207.1 arm64 HVM gp2 with OpenJDK Runtime
Environment Corretto-11.0.6.10.1 implementation of JVM.

The next two AWS EC2 instance types are: *a1.4xlarge* and *a1.metal*. Both
instances are almost identical to *a1.xlarge* instance types, but each instance
consists of 16 vCPUs (or 16 CPUs in *bare metal* type, where the physical cores
are directly used) and 32 GB RAM. All other setups have been the same as for
*a1.xlarge* instances.

For the reference, we have used the performance results obtained using a typ-
ical HPC system such as the Cray XC40 system at ICM (University of Warsaw,
Poland). The computing nodes are homogenous. Each node is equipped with
the two 12-core Intel Xeon E5-2690 v3 processors (2.60 GHz) with hyperthread-
ing available (2 threads per core) and 128 GB RAM. Cray Aries interconnect
is used for communication. The operating system on nodes of Cray XC40 was
SUSE Linux Enterprise Server 12. The Oracle Java 1.8.0_51 implementation of
Java Virtual Machine was used. The choice of Cray XC40 systems is motivated
by the recent announcement of the first exascale systems which will be a con-
tinuation of such architecture [39].

All systems use the PCJ library in version 5.0.8 for parallel execution. The
source code was compiled into *bytecode* on the local machine, and the compiled
version was transferred to computing systems. The applications were run on the
systems without any modification.

5 Performance Results

This section describes three applications with performance evaluation made on
different hardware systems. Selected applications have various execution pro-
file: CPU bound (DES decryption, Subsect. 5.1), communication bound (FFT, Subsect. 5.2) and Big Data type (WordCount, Subsect. 5.3). Such selection allows us to test different aspects of application performance including computation, communication, and I/O.

5.1 DES Decryption

Hidden text decryption is an example of the CPU intensive, trivially parallel problem which can be solved in different ways. The DES encryption can be decrypted using a brute force algorithm, therefore we have used implementation based on the analysis of all possible encryption keys. In our example, the hidden text is decoded with usage of the various keys using the DES algorithm. If the decoded text contains a given pattern, the text is treated as properly decoded. The keys are generated to cover all possible combinations of a given key length and the workload is divided equally between available processors. For the encryption key of length $n$ there are $2^n$ possible combinations. Usually, the search is stopped while matching key is found, but for our performance tests, we have tested a whole range of keys.

The decryption algorithm has been implemented in Java using different parallelization tools and methods. In all cases `SealedObject` from `javax.crypto` package has been used to store keys and to perform decryption of the hidden text.

Different implementations have been tested:

SealedDES – Java threads implementation. The work is distributed among vanilla Java threads and each thread is processing a subset of keys. At each
loop iteration new SealedObject is created. This implementation is limited to the single node (i.e. single JVM). Full source code is available on GitHub at [5].

**PCJ SealedDES** – PCJ implementation developed based on the Java threads implementation. The work is distributed among PCJ threads in the same manner as for the SealedDES. Full source code is available on GitHub at [16].

**PCJ DesDecryptor** – modified PCJ implementation. The single SealedObject is created for each PCJ thread and is reused for all iterations of the main loop. Full source code is available on GitHub at [21].

The performance results are presented in Fig. 1. The limited scalability of Java threads implementation is visible. The PCJ SealedDES implementation scales much better, up to thousands of cores, depending on the key size. The improved PCJ implementation removes multiple SealedObject creation performed in each iteration of the main loop. This significantly reduces memory operations as well as Garbage Collector invocations. As a result, code performs about 3.5 times faster and scales to the larger number of cores.

The best performing implementation (i.a. PCJ DesDecryptor) has been run on the AWS cloud. Scalability is very good, but, as presented in Fig. 1, overall performance for the AWS cloud is lower than for Cray XC40. The difference is higher for a1 instances which reflects performance difference between x86 and ARM processors.

### 5.2 Fast Fourier Transform

Fast Fourier Transform (FFT) is used as a benchmark for communication-intensive parallel algorithms. The most widely used implementation is based on the algorithm published by Takahashi and Kanada [36]. The PCJ implementation is based on the Coarray Fortran 2.0 [18]. It uses a radix 2 binary exchange algorithm that aims to reduce interprocess communication: firstly, a local FFT calculation is performed based on the bit-reversing permutation of input data; after this step all threads perform data transposition from block to cyclic layout, thus allowing for subsequent local FFT computations; finally, a reverse transposition restores data to its original block layout [18]. The communication is localized in the all-to-all routine that is used for a global conversion of data layout, from block to cyclic and vice versa. The implementation details are described in [26]. Full source code is available on GitHub at [11].

The PCJ implementation of FFT has been run on a Cray XC40 system and AWS cloud using a different number of threads. Because of the exchange algorithm, the number of threads was a power of 2 which, in the case of Cray XC40, resulted in partial utilization of the computing nodes. In the Fig. 2 we have presented application speed (i.e. number of floating-point operations per second) which is a standard performance measure used for FFT. The FFT code runs faster on Cray XC40 and scales up to several PCJ threads. The scalability is better for larger arrays as it was presented in the [26]. The AWS cloud shows also good scalability. The results obtained with 16 threads running on each processor...
are similar for ARM (Graviton) and x86 processors. For a smaller number of threads (4 per processor) the code runs faster on x86 architecture. However, with the increase of the number of nodes, the performance becomes similar for both ARM and x86 processors. One should note, that performance of the AWS cloud (x86 processors) and Cray XC40 are almost the same for the small number of threads (up to 4). In this case, code is running on a single node and is not using interconnect. This reflects the nature of the FFT application which is communication bound and therefore performance is limited by the bandwidth of communication channel rather than by CPU speed. Cray XC40 is equipped with faster interconnect which results in better performance of communication bound applications.

![Graph of FFT performance on Cray XC40 and Amazon EC2 cloud for both x86 and arm64 (AWS Graviton) processors](image)

**Fig. 2.** The performance of the parallel FFT run on Cray XC40 and Amazon EC2 cloud for both x86 and arm64 (AWS Graviton) processors. The length of the array used for FFT is $2^{20}$. The performance is plotted in GFlops (higher value means better performance).

### 5.3 WordCount

*WordCount* is traditionally used to showcase the basics of the map-reduce programming paradigm. It works by reading an input file line-by-line and counting individual word occurrences (map phase). The reduction is performed by summing the partial results calculated by worker threads.

In the PCJ code, each line of input text is divided into words and each thread saves partial results to its shareable global variable. For simplicity, the code does not perform any further preprocessing (like stemming or lemmatization). After this phase, a reduction occurs with thread 0 as root. No overlap between the two phases is facilitated. The results presented hereinafter use a simple serial reduction scheme. For better results, a hypercube-based reduction can be used, as presented in our other works [28]. Full source code and sample input data are available on GitHub at [12].
Fig. 3. The performance (execution time) of the WordCount benchmark run on AWS cloud and Cray XC40 in the weak scalability mode.

For the performance tests, we have used ISO 8859-1 encoded text of the original French version of Georges de Scudéry’s *Artamène ou le Grand Cyrus* [35]. The tests were performed in a weak scalability regime resulting in the processing 10 MB file by each thread.

Results presented in Fig. 3 show similar scalability for the code run on both Intel and ARM processors. Once more, the code executed on the Cray XC40 runs 2 times faster compared to the AWS cloud. Good performance of ARM instances compare to the x86 comes mainly from EBS disk storage.

6 Conclusions

Results presented here show the feasibility of Java language to implement parallel applications running on the AWS cloud. The performance results, compare to state of the art HPC system (Cray XC40) are good, especially that code used is, due to the portability of Java, exactly the same. The PGAS programming model implemented by PCJ library allowed for easy implementation of a various parallel schema to run both on the HPC and cloud resources.

This was possible due to the PCJ library which brings parallel capabilities of the PGAS programming model to Java. The PGAS model, available mainly for C and Fortran, has been successfully used to implement many HPC applications, but it has not been widely used for Java. As presented in the paper, the PCJ library fills this gap and allows for easy development of scalable parallel applications for both Cloud and HPC architectures without the need for recompilation.

Moreover, the presented performance of the ARM processors, shows that they become a feasible alternative to x86 architecture.
Acknowledgment. This research was carried out with the support of the Interdisciplinary Centre for Mathematical and Computational Modelling (ICM) the University of Warsaw providing computational resources under grants no GB65-15, GA69-19. The authors would like to thank CHIST-ERA consortium for financial support under HPDCJ project.

References

1. PCJ homepage. http://pcj.icm.edu.pl. Accessed 12 Feb 2020
2. Barr, J.: New – EC2 Instances (A1) Powered by Arm-Based AWS Graviton Processors. AWS News Blog, 26 November 2018. https://aws.amazon.com/blogs/aws/new-ec2-instances-a1-powered-by-arm-based-aws-graviton-processors/. Accessed 18 Feb 2020
3. Barr, J.: Coming Soon – Graviton2-Powered General Purpose, Compute-Optimized and Memory-Optimized EC2 Instances. AWS News Blog, 3 December 2019. https://aws.amazon.com/blogs/aws/coming-soon-graviton2-powered-general-purpose-compute-optimized-memory-optimized-ec2-instances/. Accessed 18 Feb 2020
4. Boněr, J., Kuleshov, E.: Clustering the Java virtual machine using aspect-oriented programming. In: AOSD 2007 Proceedings of the 6th International Conference on Aspect-Oriented Software Development (2007)
5. Carey, N.: Parallel DES Key Cracker Benchmark: SealedDES implementation - source code. https://github.com/ncarey/Parallel-DES-Key-Cracker-Benchmark/blob/a760f6412496eb186a6dea53fa0aab1ed3546732/source/SealedDES.java. Accessed 12 Feb 2020
6. Carlson, W.W., Draper, J.M., Culler, D.E., Yelick, K., Brooks, E., Warren, K.: Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences (1999)
7. Carpenter, B., Getov, V., Judd, G., Skjellum, A., Fox, G.: MPJ: MPI-like message passing for Java. Concurrency: Pract. Experience 12(11), 1019–1038 (2000)
8. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the chapel language. Int. J. High Perform. Comput. Appl. 21(3), 291–312 (2007)
9. Charles, P., et al.: X10: an object-oriented approach to non-uniform cluster computing. In: ACM SIGPLAN Notices, vol. 40, pp. 519–538. ACM (2005)
10. Feind, K.: Shared memory access (SHMEM) routines. Cray Research (1995)
11. Górski, L.: PCJ Fast Fourier Transform - source code. https://github.com/hpdcej/hpc-challenge-fft/tree/ebd557e40ad50f614a869000321ee822b67d2623. Accessed 12 Feb 2020
12. Górski, L.: PCJ WordCount - source code. https://github.com/hpdcej/wordcount/tree/6a265bc92147a89c37176692ccae8dcf8d97df72. Accessed 12 Feb 2020
13. Górski, L., Rakowski, F., Bala, P.: Parallel differential evolution in the pgas programming model implemented with PCJ Java library. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K., Kitowski, J., Wiatr, K. (eds.) PPAM 2015. LNCS, vol. 9573, pp. 448–458. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32149-3_42
14. Gupta, A., Milojicic, D.: Evaluation of HPC applications on cloud. In: 2011 Sixth Open Cirrus Summit, pp. 22–26. IEEE (2011)
15. He, Q., Zhou, S., Kobler, B., Duffy, D., McGlynn, T.: Case study for running HPC applications in public clouds. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, pp. 395–401 (2010)
16. Korona, M.: PCJ SealedDES - source code. https://github.com/hpdcj/PCJ-tests/blob/96907dd4f0b165cdd1a807005294c5cb84012c2/src/org/pcj/tests/app/des/PCJSealedDES.java. Accessed 12 Feb 2020

17. Liang, J., Qu, B., Suganthan, P.: Problem definitions and evaluation criteria for the CEC 2014 special session and competition on single objective real-parameter numerical optimization. Zhengzhou University, Zhengzhou China and Technical Report, Nanyang Technological University, Singapore, Computational Intelligence Laboratory (2013)

18. Mellor-Crummey, J., et al.: Class ii submission to the hpc challenge award competition coarray fortran 2.0. http://www.hpcchallenge.org/presentations/sc2011/hpcc11_report_caf2_0.pdf

19. Nester, C., Philippsen, M., Haumacher, B.: A more efficient RMI for Java. Java Grande 99, 152–159 (1999)

20. Nieplocha, J., Harrison, R.J., Littlefield, R.J.: Global arrays: a non-uniform-memory-access programming model for high-performance computers. J. Supercomputing 10(2), 169–189 (1996)

21. Nowicki, M.: PCJ DesDecryptor - source code. https://github.com/hpdcj/PCJ-tests/blob/12c407d24c57442d8fac1395d044bd846d021cc/src/org/pcj/tests/app/des/DesDecryption.java. Accessed 12 Feb 2020

22. Nowicki, M., Bala, P.: Parallel computations in Java with PCJ library. In: 2012 International Conference on High Performance Computing and Simulation (HPCS), pp. 381–387. IEEE (2012)

23. Nowicki, M., Bzhalava, D., Bala, P.: Massively parallel sequence alignment with BLAST through work distribution implemented using PCJ library. In: Ibrahim, S., Choo, K.-K.R., Yan, Z., Pedrycz, W. (eds.) ICA3PP 2017. LNCS, vol. 10393, pp. 503–512. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65482-9_36

24. Nowicki, M., Bzhalava, D., Bala, P.: Massively parallel implementation of sequence alignment with basic local alignment search tool using parallel computing in Java. J. Comput. Biol. 25(8), 871–881 (2018)

25. Nowicki, M., Górski, L., Bala, P.: PCJ - Java library for highly scalable HPC and big data processing. In: 2018 International Conference on High Performance Computing and Simulation (HPCS), pp. 12–20. IEEE (2018)

26. Nowicki, M., Górski, L., Bala, P.: Performance evaluation of parallel computing and Big Data processing with Java and PCJ library. Cray Users Group (2018). https://cug.org/proceedings/cug2018_proceedings/includes/files/pap139s2-file1.pdf. Accessed 12 Feb 2020

27. Nowicki, M., Górski, L., Bala, P.: PCJ Java Library as a solution to integrate HPC. In: Big Data and Artificial Intelligence Workloads (in review)

28. Nowicki, M., Ryczkowska, M., Górski, L., Bala, P.: Big data analytics in Java with PCJ library: performance comparison with Hadoop. In: Wyzykowski, R., Dongarra, J., Deelman, E., Karczewski, K. (eds.) PPAM 2017. LNCS, vol. 10778, pp. 318–327. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78054-2_30

29. Oracle: Java Platform, Standard Edition 6, Features and Enhancements (2006). http://www.oracle.com/technetwork/java/javase/features-141434.html. Accessed 12 Feb 2020

30. Oracle: Class ForkJoinPool (2018). https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/ForkJoinPool.html

31. Posner, J., Reitz, L., Fohry, C.: Comparison of the HPC and big data Java libraries spark, PCJ and APGAS. In: 2018 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM), pp. 11–22. IEEE (2018)
32. Rakowski, F., Karbowski, J.: Optimal synaptic signaling connectome for locomotory behavior in Caenorhabditis elegans: design minimizing energy cost. PLoS Comput. Biol. 13(11), e1005834 (2017)
33. Reid, J.: The new features of Fortran 2008. In: ACM SIGPLAN Fortran Forum, vol. 27, pp. 8–21. ACM (2008)
34. Ryczkowska, M., Nowicki, M., Bala, P.: Level-synchronous BFS algorithm implemented in Java using PCJ library. In: 2016 International Conference on Computational Science and Computational Intelligence (CSCI), pp. 596–601. IEEE (2016)
35. Scudéry, M.D.: Artamène ou le grand Cyrus (1972)
36. Takahashi, D., Kanada, Y.: High-performance radix-2, 3 and 5 parallel 1-d complex FFT algorithms for distributed-memory parallel computers. J. Supercomputing 15(2), 207–228 (2000)
37. Tardieu, O.: The apgas library: resilient parallel and distributed programming in java 8. In: Proceedings of the ACM SIGPLAN Workshop on X10, pp. 25–26. ACM (2015)
38. Thirunavukkarasu, S.: Foundation of learning structure infusion for high execution processing in the cloud. Int. J. Pure Appl. Math. 119(12), 5253–5264 (2018)
39. Trader, T.: It’s Official: Aurora on Track to Be First US Exascale Computer in 2021. HPC Wire, 18 March 2019. https://www.hpcwire.com/2019/03/18/its-official-aurora-on-track-to-be-first-u-s-exascale-computer-in-2021/. Accessed 18 Feb 2020
40. Vega-Gisbert, O., Roman, J.E., Squyres, J.M.: Design and implementation of Java bindings in Open MPI. Parallel Comput. 59, 1–20 (2016)
41. Yelick, K., et al.: Titanium: a high-performance Java dialect. Concurrency Comput. Pract. Experience 10(11–13), 825–836 (1998)
42. Yokoyama, D., Schulze, B., Borges, F., McEvoy, G.: The survey on ARM processors for HPC. J. Supercomputing 75(10), 7003–7036 (2019). https://doi.org/10.1007/s11227-019-02911-9