Journal of Big Data

RESEARCH Open Access

PCJ Java library as a solution to integrate HPC, Big Data and Artificial Intelligence workloads

Marek Nowicki[1*], Łukasz Górski[2] and Piotr Bała[2]

*Correspondence: faramir@mat.umk.pl
[1] Faculty of Mathematics and Computer Science, Nicolaus Copernicus University in Toruń, ul. Chopina 12/18, 87-100 Toruń, Poland
Full list of author information is available at the end of the article

Abstract
With the development of peta- and exascale size computational systems there is growing interest in running Big Data and Artificial Intelligence (AI) applications on them. Big Data and AI applications are implemented in Java, Scala, Python and other languages that are not widely used in High-Performance Computing (HPC) which is still dominated by C and Fortran. Moreover, they are based on dedicated environments such as Hadoop or Spark which are difficult to integrate with the traditional HPC management systems. We have developed the Parallel Computing in Java (PCJ) library, a tool for scalable high-performance computing and Big Data processing in Java. In this paper, we present the basic functionality of the PCJ library with examples of highly scalable applications running on the large resources. The performance results are presented for different classes of applications including traditional computational intensive (HPC) workloads (e.g. stencil), as well as communication-intensive algorithms such as Fast Fourier Transform (FFT). We present implementation details and performance results for Big Data type processing running on petascale size systems. The examples of large scale AI workloads parallelized using PCJ are presented.

Keywords: Parallel computing, Java, Partitioned Global Address Space, PCJ, HPC, Big Data, Artifical Intelligence

Introduction

Artificial Intelligence (AI), also known as computational intelligence, is becoming more and more popular in a large number of disciplines. It helps to solve problems for which it is impossible or at least very hard to write a traditional algorithm. Currently, a deep learning approach is very famous and widely studied. Deep learning, or more broadly speaking, machine learning and neural network approaches, parses very large training data, learns from it by fixing its internal state. The bigger the volume and variety of the training data the neural network can better *learn* the environment and then give better answers for the previously not observed data. Processing a large amount of data and teaching neural networks with a large number of parameters requires significant computational effort not available at laptops or workstations. Therefore, it is important to have a tool, that could integrate Big Data processing with the Artificial Intelligence workloads on the High-Performance Computing (HPC) systems.

There is an ongoing need to adapt existing systems and design new ones that would facilitate the AI-based calculations. The research tries to push existing limitations in the areas such as the performance of heterogenous systems that employ specialised hardware for AI-based computation acceleration or I/O and networking performance (to enhance the throughput of training or inference data [1]). Whilst the deployment of new solutions is concerned with the advent of new AI-based tools (with Python-based libraries like PyTorch or TensorFlow), their integration with existing HPC systems is not always easy. The Parallel Computing in Java (PCJ) library is presented herein as an HPC-based tool that can be used to bridge together various workloads that are currently running on the existing systems. In particular, we show that it can be used to distribute neural network training and is a good performer as far as I/O is concerned, especially in comparison with Hadoop/Spark. The former corroborates the idea that the library can be used in concert with existing cluster management tools (like Torque or SLURM) to distribute work across a cluster for neural network training or to deploy a production-ready model in many copies for fast inference; the latter proves that training data can be efficiently handled.

Recently, as part of the various exascale initiatives, there has been a strong interest in running Big Data and AI applications on HPC systems. Because of the different tools used in these areas as well as due to the different nature of the algorithms used, the achievement of good performance is difficult. Big Data and AI applications are implemented in Java, Scala, Python and other languages that are not widely used in HPC, which is still dominated by C and Fortran. Moreover, Big Data and AI frameworks rely on dedicated environments such as Hadoop or Spark which are difficult to integrate with the traditional HPC management systems. To solve this problem, vendors are putting a lot of effort to rewrite the most time-consuming parts to C/MPI, but this is a laborious and not easy task and successes are limited.

There is a lot of effort to adapt Big Data and AI software tools to HPC systems. However, this approach does not remove the limitations of existing software packages and libraries. Significant effort is also put to modify existing HPC technologies to make them more flexible and easy to use, but success is limited. The popularity of traditional programming languages such as C and Fortran decreases. Message-Passing Interface (MPI), which is the basic parallelization library, is also criticized because of the complicated Application Programming Interface (API) and difficult 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 peta- and exascale ones. Solutions developed by the hardware providers take a direction of unification of operating systems and compilers and bringing them to workstations. Such an approach is not enough and new solutions are necessary.

Our approach presented in this paper is to use a well-established programming language (Java) to provide users with the easy to use, flexible and scalable programming framework that allows for development of different types of workloads including HPC, Big Data, AI and others. This opens the field to easy integration of HPC with Big Data and AI applications. Moreover, due to the Java portability, user can develop solution on his laptop or workstation and than move, even without recompilation, to the cloud or HPC infrastructure including peta-scale systems.

For these purposes, we have developed the PCJ library [2]. PCJ is implementing the Partitioned Global Address Space (PGAS) programming paradigm [3], as languages adhering to it are very promising in the context of exascale. In the PGAS model, all variables are private to the owner thread. Nevertheless, some variables can be marked as shared. Shared variables are accessible to other threads of execution, which can address the remote variable and modify it or store locally. The PGAS model provides simple and easy to use constructs to perform basic operations which significantly reduces programmers' effort preserving code performance and scalability. 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.

The PCJ library won the HPC Challenge award in 2014 [4] and has been already successfully used for parallelization of various applications. A good example is a communication-intensive graph search from the Graph500 test suite. The PCJ implementation scales well and outperforms the Hadoop implementation by a factor of 100 [5], but not all benchmarks were well suited for Hadoop processing. Paper [6] compares the PCJ library and Apache Hadoop using a conventional, widely used benchmark for measuring the performance of Hadoop clusters, and shows that the performance of applications developed with the PCJ library is similar or even better than the Apache Hadoop solution. The 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 [7]. Recent examples of PCJ usage include parallelization of the sequence alignment [8]. The PCJ library allowed for the easy implementation of the dynamic load balancing for multiple NCBI-BLAST instances spanned over multiple nodes giving the results at least 2 times earlier than the implementations based on the static work distribution [9].

In previous works, we have shown that the PCJ library allows for the easy development of computational applications as well as Big Data and AI processing. In this paper, we focus on the comparison of PCJ with Java-based solutions. The performance comparison with the C/MPI based codes has been presented in previous papers [2, 10].

The remainder of this paper is organized as follows. After remarks on emerging programming languages and programming paradigms ("Prospective languages and programming paradigms" section), we present the basic functionality of the PCJ library ("Methods" section). "Results and discussion" section contains subsections with results and discussion of various types of applications. "HPC workloads" section contains the performance results are presented for a different class of applications including traditional computational intensive (HPC) workloads (e.g. stencil), as well as communication-intensive algorithms such as Fast Fourier Transform (FFT), in "Data analitycs" section we present implementation details and performance results for Big Data type processing running on petascale size systems. The examples of large scale AI workloads parallelized using PCJ are presented in "Artificial Intelligence workloads" section. The section finishes with a description of ongoing work on the PCJ library. The paper concludes in "Conclusion" section.

Prospective languages and programming paradigms

A growing interest in running machine learning and Big Data workloads is associated with new programming languages that have not been traditionally considered for use in high-performance computing. This includes Python, Julia, Java, and some others.

Python is now being viewed as acceptable for HPC applications, due to the 2016 Gordon Bell finalist application PyFR [11], which demonstrated that Python application performance can compete head-to-head against native language applications written in C/C++ and Fortran on the world's largest supercomputers. However, the multiple versions available have limited backward compatibility which requires significant administrative effort to handle them. A good example of problems is a long startup time of the Python application reported [12]. For the large number of nodes it can take hours. The dedicated effort is required to minimize it to acceptable value (see Fig. 1).

Python remains a single-threaded environment with the global interpreter lock as the main bottleneck. Threads must wait for other threads to complete before starting to do their assigned work. In result, the production code is too slow to be useful for large simulations. There are some other implementations with better thread support, but their compatibility could be limited.

The hardware vendors provide a tuned version of Python to improve performance. It is done by using some C functions that perform (when coded optimally) at machine level speeds. These libraries can vectorize and parallelize the assigned workload and understand the different hardware architectures.

Julia is a programming language that is still new and relatively unknown by many in the HPC community but it is rapidly growing in popularity. For the parallel execution, Julia provides `Tasks` and other modules that rely on the Julia runtime library. These modules allow to suspend and resume computations with full control of inter-task communication without having to manually interface with the operating system's scheduler. A good example of the HPC application implemented in Julia is the Celeste project [13]. It was able to attain performance using only Julia source code and the Julia threading model. As a result, it was possible to fully utilize the manycore Intel Xeon Phi processors.

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. There are also solutions based on various implementations of the MPI library [14, 15], distributed Java Virtual Machine (JVM) [16] and solutions based on Remote Method Invocation (RMI) [17]. Such

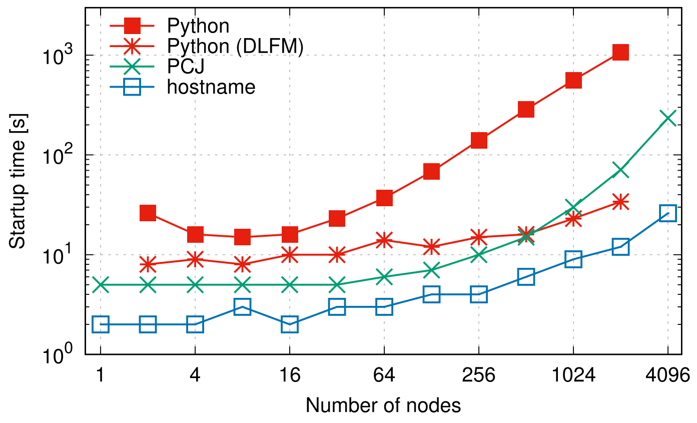

Fig. 1 PCJ startup time (from [10]) compared to the Python loading time for original and modified Python installation (see [12]). The execution time of the `hostname` command run concurrently on the nodes is plotted for reference

solutions rely on the external communication libraries written in other languages which causes many problems in terms of usability, portability, scalability, and performance.

We should also mention solutions motivated by the partitioned global address space approach represented by Titanium—a scientific computing dialect of Java [18]. Titanium defines new language constructs and has to use a dedicated compiler which makes it difficult to follow recent changes in Java language.

Python, Julia and, to some extent, Java follow the well-known path. The parallelization is possible based on the independent task model with limited communication capabilities. This significantly reduces classes of algorithms that can be implemented to trivially parallel ones. An alternative approach is based on the interfacing MPI library, thus using a message-passing model.

Recently, programming models based on PGAS are gaining popularity. It is expected that PGAS languages will be more important at exascale because of the distinct features and development efforts which is lower than for other approaches. The PGAS model can be supported by a library such as SHMEM [19], Global Arrays [20] or Charm++ [21] or by a language, such as UPC [22], Fortran [23] or Chapel [24]. PGAS systems differ in the way the global namespace is organized. Some, such as SHMEM or Fortran, provide a local view of data while others provide a global view of data.

Until now, there was no successful realization of the PGAS programming model for Java. Developed by us, the PCJ library is the successful implementation providing good scalability and reasonable performance. Another prospective implementation is APGAS, a library offering an X10-like programming solution for Java [25].

Methods

The PCJ library

PCJ [2] is an OpenSource Java library available under the BSD license with the source code hosted on GitHub. PCJ does not require any language extensions or special compiler. The user has to download the single jar file and then he can develop and run parallel applications on any system with Java installed. Alternatively, build automation tool like Maven or Gradle can be used, as the library is deployed into Maven Central Repository (*group*: `pl.edu.icm.pcj`, *artifact*: `pcj`). 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 PCJ library can be considered as a simple extension to Java to write parallel programs. It provides necessary tools for easy implementation of data and work partitioning best suited to the problem. PCJ does not provide automatic tools for the data distribution or task parallelization but once the parallel algorithm is given it allows for its efficient implementation.

Idea

The PCJ library follows the common PGAS paradigm (see Fig. 2). The application is run as a collection of threads—called here PCJ threads. Each PCJ thread owns a local copy of variables, each copy has a different location in physical memory. This applies also to the threads run within the same JVM. The PCJ library provides methods to start PCJ threads in one JVM or in a parallel environment—using multiple JVMs. PCJ threads are created at the application launch and stopped during execution termination. The library provides also basic methods to manage threads such as starting execution, finding the total number of threads and number of actual PCJ thread as well as methods to manage groups of threads.

The PCJ library provides methods to synchronize execution (`PCJ.asyncBarrier()`) and to exchange data between threads. The communication is one-sided and asynchronous and is performed by calling `PCJ.asyncPut()`, `PCJ.asyncGet()` and `PCJ.asyncBroadcast()` methods. The synchronous (blocking) versions of communication methods are also available. The data exchange can be done only for specially marked variables. Exposition of local fields for remote addressing is performed with the use of `@Storage` and `@RegisterStorage` annotations.

PCJ provides mechanisms to control the state of data transfer, in particular, to ensure a programmer that asynchronous data transfer is finished. For example, a thread can get a shared variable and stores it in the `PcjFuture<double[]>` object. Then, the received value is copied to the local variable. The whole process can be overlapped with other operations, eg. calculations. The programmer can check the status of data transfer using `PcjFuture`'s methods.

The PCJ API follows successful PGAS implementations such as Co-Array Fortran or X10, however, the dedicated effort has been done to align it with the experience of Java programmers. The full API is presented in the Table 1.

With version 5.1 of the PCJ library, we provide users with the methods for collective operations. These methods implement the most efficient communication using a binary tree which scales with the number of nodes n as $log_2 n$. This reduction algorithm is faster than simple iteration over available threads, especially for a large number of PCJ threads running on a node. Collective methods collect data within a physical node before sending it to other nodes which reduces the number of communication performed between nodes, i.e. between different JVM's.

Implementation details

The use of Java language requires a specific implementation of basic PGAS functionality which is multi-threaded execution and communication between threads.

Execution

PCJ allows for different scenarios such as multiple threads in a single JVM or runs multiple JVMs on a single physical node. Starting a JVM on a remote node relies on Secure Shell (SSH) connection to the machine. It is necessary to set up passwordless login, e.g. by using authentication keys without a passphrase. As presented in Fig. 1 the startup time is lower than for Python. However, it grows up with the number of nodes, but it should be noted that PCJ startup time includes initial communication and synchronization of threads which is not included for other presented solutions.

It is also possible to utilize the execution schema accessible on supercomputers or clusters (like `aprun`, `srun`, `mpirun` or `mpiexec`) that starts selected application on all nodes allocated for the job. In this situation, instead of calling `deploy()`, the `start()` method should be used. However, in that situation, internet address instead of loopback addresses should be used for describing the nodes. The file with node descriptions has to be prepared by the user, e.g. by saving the output of `hostname` command executed on allocated nodes.

Table 1 Summary of the elementary PCJ elements

`org.pcj` (package)		
`StartPoint`		interface with the `main()` method
`ExecutionBuilder`		class for preparing and starting execution (with nodes configuration and other properties)
`addNode`	`addNodes`	adds node(s) to execution builder
`addProperty`	`addProperties`	adds property to execution builder
`deploy`		starts application using internal PCJ mechanism
`start`		starts application using cluster execution system
`PCJ`		class with basic static methods
`myId`		returns current PCJ thread id
`threadCount`		returns count of PCJ threads
`get`	`asyncGet`	gets value of shared variable
`put`	`asyncPut`	puts value to shared variable
`accumulate`	`asyncAccumulate`	accumulates value of shared variable
`broadcast`	`asyncBroadcast`	broadcasts and putting value to shared variable
`waitFor`		waits for modification of variable
`collect`	`asyncCollect`	collects shared variable from all PCJ threads into array
`reduce`	`asyncReduce`	reduces shared variable from all PCJ threads
`barrier`	`asyncBarrier`	execution barrier
`joinGroup`		joins to the subgroup of PCJ threads
`registerStorage`		registers storage classes
`PcjFuture<T>`		class for receiving notifications of asynchronous operations
`ReduceOperation`		serializable iterface used in reduce and accumulate operations
`PcjRuntimeException`		exception used by the PCJ library
`@Storage`		annotation for `enum` with names of shared variables
`@RegisterStorage`		annotation for class implementing StartPoint interface for registering storage

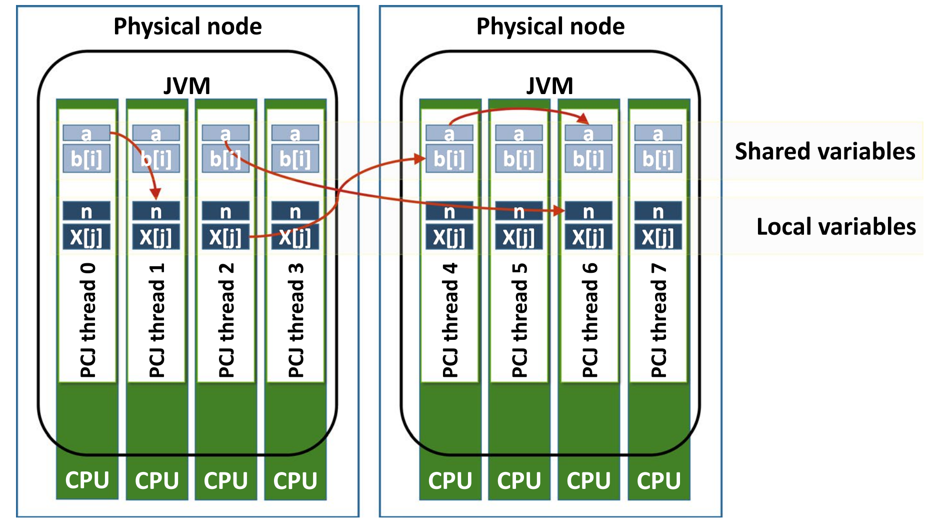

Fig. 2 Diagram of PCJ computing model (from [48]). Arrows present possible communication using `put(...)` or `get(...)` methods acting on shared variables

Communication

The architectural details of the communication between PCJ threads are presented in Fig. 3.

The intranode communication is implemented using the Java Concurrency mechanism. Sending objects from one thread to another requires cloning object' data. Copying just object reference could cause concurrency problems in accessing the object.

PCJ library makes sure that the object is deeply copied by serializing the object and then deserializing it on the other thread. It is done partially by the sending thread (serializing) and partially by *local workers* (deserializing). This way of cloning data is safe, as the data is deeply copied—the other thread has its own copy of data and can use it independently.

`Object.clone()` method available in Java is not sufficient. It does not force to create a deep copy of the object. For example, it creates only a shallow copy of arrays, therefore the data stored in the arrays are not copied between threads. The same stands for the implementation of this method in standard classes like `java.util.ArrayList`. Moreover, it requires implementation of `java.lang.Cloneable` interface for all communicable classes and overriding `clone()` method with a `public` modifier that also had to copy all mutable objects into clone. The serialization/deserialization mechanism is more general and requires only that all used classes be serializable, thus implementing the `java.io.Serializable` interface, and in most cases does not require writing serialization handling methods. Additionally, serialization, so changing objects into bytes stream (array), is also a requirement for sending data between nodes.

The communication between nodes uses standard network communication with *sockets*. The data is serialized by the sending thread and the transferred data is deserialized by *remote workers*. The network communication is performed using Java New I/O classes (i.e. `java.nio.*`). The details of the algorithms used to implement PCJ communication are described in [26].

Example

The example parallel application which sums up *n* random numbers is presented in Listing 1. The `PcjExample` class implements the `StartPoint` interface which provides methods to start the application in parallel. The `PCJ.executionBuilder()` is used to set up the execution environment: the class which is used as the main class for parallel application and a list of nodes provided here in the *nodes.txt* file.

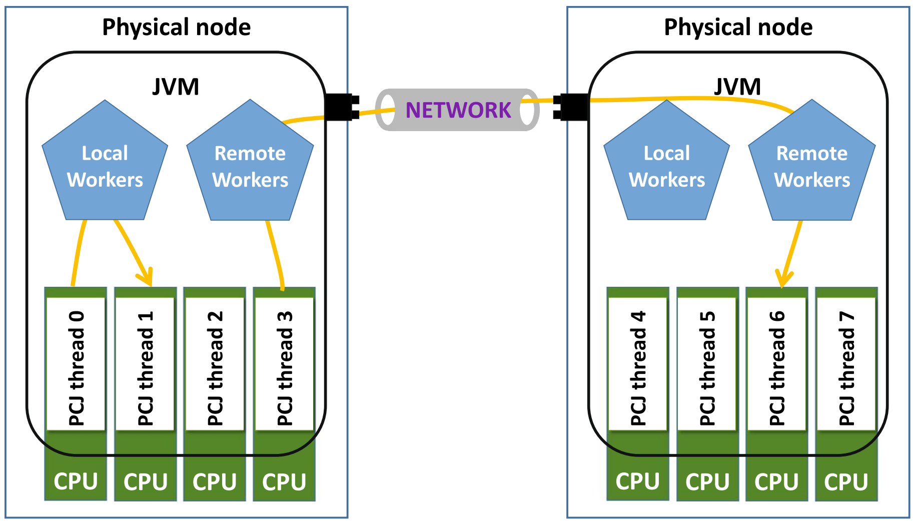

Fig. 3 Diagram of PCJ communication model (from [6]). Arrows present local and remote communication

The work is distributed in block manner—each PCJ thread is summing up part of the data. The parallelization is performed by changing the length of the main loop defined in line 29. The length of the loop is adjusted automatically to the number of threads used for execution.

The partial sums are accumulated in the variable `a` local to each PCJ thread. The variable `a` can be get/put/broadcast as it is defined in lines 12–14. Line 9 ensures that this set of variables can be used in class `PcjExample`.

To ensure that all threads finished calculating partial sums the `PCJ.barrier()` method is used in line 32. Partial sums are then accumulated at PCJ thread #0 using `PCJ.reduce()` method (line 34) and printed out.

```java
 1 import java.io.File;
 2 import java.io.IOException;
 3 import java.util.Random;
 4 import org.pcj.PCJ;
 5 import org.pcj.RegisterStorage;
 6 import org.pcj.StartPoint;
 7 import org.pcj.Storage;
 8
 9 @RegisterStorage (PcjExample.Shareable.class)
10 public class PcjExample implements StartPoint {
11
12   @Storage (PcjExample. class )
13   enum Shareable { a }
14   public double a;
15
16   public static void main(String[] args)
17                                         throws IOException {
18     PCJ.executionBuilder(PcjExample. class )
19         .addNodes( new File( "nodes .txt "))
20         .start();
21   }
22
23   @Override
24   public void main() throws Throwable {
25     Random r = new Random();
26     int n = 1_000_000_000;
27     int nl = (n + PCJ.threadCount() - (PCJ.myId() + 1))
28                                        / PCJ.threadCount();
29     for (int i = 0; i < nl; i++) {
30       a = a + r.nextDouble();
31     }
32     PCJ.barrier();
33     if (PCJ.myId() == 0) {
34       double s = PCJ.reduce(Double::sum, Shareable.a);
35
36       System.out.println(s);
37     }
38   }
39 }
```

Listing 1 Example application assigning summing up 10^9 random floating-point numbers using static work distribution.

Fail-safe

Node or thread failure in the PCJ library uses a fail-safe mechanism. Without that mechanism, the whole computation could be stuck in not a recoverable state. When computations are executed on a cluster system, that situation could cause useless utilization of Central Processing Unit (CPU)-hours without any useful action done up to the job time limit.

In version 5.1 of the PCJ library, there is an added fail-safe mechanism that causes whole computations gracefully finish when failure appears. The fail-safe mechanism is based on alive and abort messages—the heartbeat mechanism.

The alive message is periodically sent to a node's neighbour nodes, i.e. parent and children nodes, by each node, e.g. neighbours of node 1 are nodes: 0, 3 and 4 (cf. Fig. 4). If the node does not receive an alive message from one of its neighbour nodes within predetermined, configurable time, it assumes the failure of the node. Failure of the node is also assumed when an alive message cannot be sent to the node, or one of the node's PCJ threads exits with an uncaught exception.

When the failure occurs, the node that discovers the breakdown removes failed node from its neighbours' list, immediately sends abort messages to the rest of neighbours, and interrupts PCJ threads that are executing on the node. Each node that receives an abort message removes the node that sent the message from its neighbours' list (to avoid sending a message back to already notified node), and sends an abort message to all remaining neighbours and then interrupts its own PCJ threads.

The fail-safe mechanism allows for quicker shutting down after a breakdown, so the cluster's CPU-hours are not uselessly utilized. Users can disable the fail-safe mechanism by setting an appropriate flag of PCJ execution.

Results and discussion

The performance results have been obtained using the Cray XC40 system at ICM (University of Warsaw, Poland) and HLRS (University of Stuttgart, Germany). The computing nodes (boards) are equipped with two Intel Xeon E5-2690 v3 (ICM) or Intel Haswell E5-2680 (HLRS) processors, each processor contains 12 cores. In both cases, there is hyperthreading available (2 threads per core). Both systems have Cray Aries interconnect installed. The PCJ library has been also tested on the other architectures such as Power 8 or Intel KNL [27]. However, we decided to present here results obtained using Cray XC40 systems since one of the first exascale systems will be a continuation of such architecture [28]. We have used Java 1.8.0_51 from Oracle for PCJ and Oracle JDK 10.0.2 for APGAS. For the C/MPI we have used Cray MPICH implementations in version 8.3 and 8.4 for ICM and HLRS machines respectively. We have used OpenMPI in version 4.0.0, that gives Java bindings for the MPI, to collect data for the Java/MPI execution.

HPC workloads

2D stencil

As an example of a 2D stencil algorithm we have used *Game of Life* which can be seen as a typical 9-point 2D stencil—the 2D Moore neighborhood. The *Game of Life* is a cellular automaton devised by John Conway [29]. In our implementation [30] the board is not infinite—it has its maximum width and height. Each thread owns a subboard—a part of the board divided in a uniform way using block distribution. Although there are known fast algorithms and optimizations that can save computational time generating the next universe state, like Hashlife or memorization of the changed cells, we have decided to use a straightforward implementation with a lookup of the state for each cell. However, to save memory, each cell is represented as a single bit, where 0 and 1 mean that the cell is dead and alive respectively.

After generating the new universe state, the border cells of subboards are exchanged asynchronously between proper threads. The threads that have cells on the first and last columns and rows of the universe are not exchanging the cells state to the opposite threads. The state of neighbour cells that would be behind the universe edge is treated as *dead*.

We have measured the performance in the total number of cells processed in the unit of time ($cells/s$). For each test, we performed 11 time steps. We warmed up the Java Virtual Machine to allow the JVM to use Just-in-Time (JIT) compilation to optimize the run instead of execution in interpreted mode. We also ensured that the Garbage Collector (GC) had not much impact on the gained performance. To do so we took peak performance (maximum of steps performance) for the whole simulation. We have used 48 working threads per node.

Figure 5 presents performance comparison of *Game of Life* applications for $604,800 \times 604,800$ cells universe. The performance for both implementations (PCJ and Java/MPI) is very similar and results in almost ideal scalability. C/MPI version presents 3-times higher performance and similar scalability. The performance data shows scalability up to 100,000 threads (on 2048 nodes). For a larger number of threads, the parallel efficiency decreases due to the small workload run on each processor compared to the communication time required for halo exchange. The scaling results obtained in the weak scaling mode (i.e. with a constant amount of work allocated to each thread despite the thread number) show good scalability beyond 100,000 thread limit [10]. The ideal scaling dashed line for PCJ is plotted for reference. Presented results show ability of running large scale HPC applications using Java and the PCJ library.

Inset in Fig. 5 presents the performance statistics calculated based on 11 time steps of the *Game of Life* application executed on 256 nodes (12,288 threads). The ends of *whiskers* are minimum and maximum values, a cross (×) represents an average value, a box

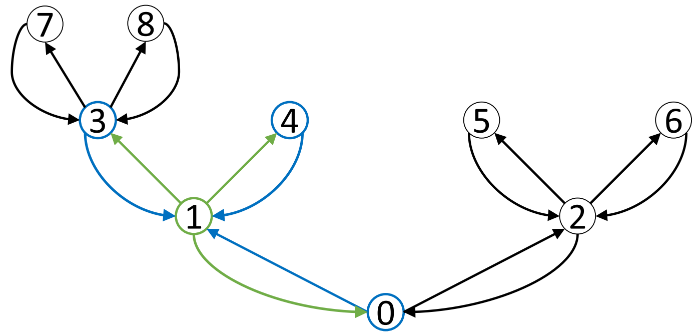

Fig. 4 Communication tree with selected neighbours of node-#1 for fail-safe mechanism. Green arrows represent the node-#1 alive messages sent to its neighbour nodes. Blue arrows represent the node-#1 alive messages received from the neighbour nodes

represents values between 1st and 3rd quartiles, and a band inside the box is a median value. In the case of C/MPI, the box and *whiskers* are not visible, as the execution shows the same performance for all of the execution steps. In the case of JVM executions (PCJ and Java/MPI), minimum values come from the very first steps of execution, when the execution was made in interpreted mode. However, the JIT compilation quickly optimized the run and the vast majority of steps were run with the highest performance. It is clearly visible that Java applications, after JIT compilation, has very stable performance results as the maximum, median and 1st and 3rd quartiles data are almost indistinguishable in the figure.

Fast Fourier Transform

The main difficulty in efficient parallelization of FFT comes from the global character of the algorithm, which involves an extensive all to all communication. One of the efficient distributed FFT implementations available is based on the algorithm published by Takahashi and Kanada [31]. It is used as a reference MPI implementation in the HPC Challenge Benchmark [32], a well-known suite of tests for assessing the HPC systems performance. This implementation is treated as a baseline for the tests of the PCJ version described herein (itself based on [33]), with the performance of all-to-all exchange being the key factor.

In the case of PCJ code [34] we have chosen, as a starting point, PGAS implementation developed for Coarray Fortran 2.0 [33]. The original Fortran algorithm uses a radix 2 binary exchange algorithm that aims to reduce interprocess communication and is structured as follows: 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 is original block layout [33]. Similarly to Random Access implementation, inter-thread communication is therefore localized in the all-to-all routine that is used for a global conversion of data layout, from block to cyclic and *vice verse*. Such implementation allows one to limit the communication, yet makes the implementation of all-to-all exchange once again central to the overall program's performance.

The results for complex one-dimensional FFT of 2^{30} elements (Fig. 6) show how the three alternative PCJ all-to-all implementations compare in terms of scalability. Blocking and non-blocking ones iterate through all other threads to read data from their shared memory areas (`PcjFuture`s are used in a non-blocking version). Hypercube-based communication utilizes a series of pairwise exchanges to avoid network congestion. While nonblocking communication achieved the best peak performance, the hypercube-based solution exploited the available computational resources to the greatest extent, reaching peak performance for 4096 threads when compared to 1024 threads in the case of nonblocking communication. Java/MPI code uses the same algorithm as PCJ for calculation and all-to-all exchange. It is implemented using the native MPI primitive. The scalability of the PCJ implementation follows the results of reference C/MPI code as well as those of Java/MPI. Total execution time for Java is larger when compared to all-native implementation irrespective of the underlying communication library. Presented results

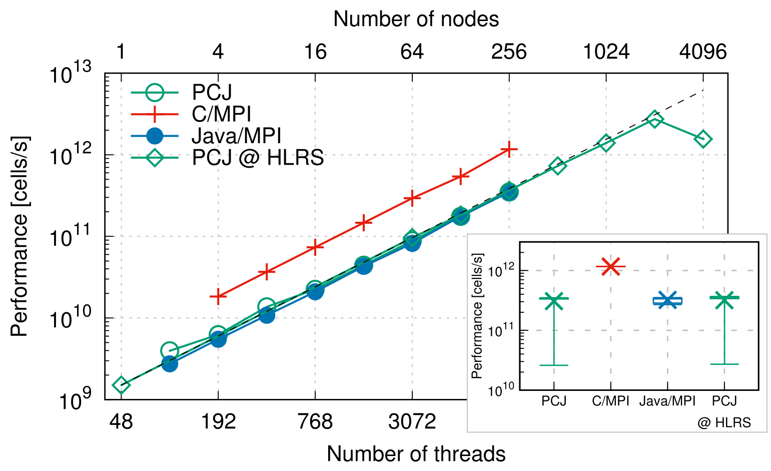

Fig. 5 Performance results of Game of Life implemented with PCJ, C/MPI and Java/MPI for 604, 800 × 604, 800 cells universe. The strong scaling results were obtained on Cray XC40 at ICM (PCJ, C/MPI and Java/MPI) and Cray XC40 at HLRS (PCJ). Ideal scaling for PCJ is drawn (dashed line). Inset plot presents the performance statistics calculated based on 11 time steps on 256 nodes (12,288 threads). In the inset plot, ends of *whiskers* are minimum and maximum values, a cross (×) represents an average value, a box represents values between 1st and 3rd quartiles, and a band inside the box is a median value

confirm, that performance and scalability of PCJ and Java/MPI implementations are similar. The PCJ library is easier to use, less error prone and does not require libraries external to Java such as MPI. Therefore it is good alternative to MPI. Java implementations are slower than HPCC which is implemented using C. This comes from that different ways of storing and accessing data.

Data analitycs

WordCount

WordCount is traditionally used for demonstrative purposes to showcase the basics of the map-reduce programming paradigm. It works by reading an input file on a line-by-line basis and counting individual word occurrences (map phase). The reduction is performed by summing the partial results calculated by worker threads. Full source code of the application is available at GitHub [35]. Herein the comparison between PCJ's and APGAS's performance is presented with the C++/MPI version shown as a baseline. APGAS$_{stat}$ is the basic implementation, APGAS$_{dyn}$ is a version enhanced with dynamic load-balancing capabilities. The APGAS library, as well as its implementation of WordCount code, are based on the prior work [25]. APGAS code was run using SLURM in Multiple Programs, Multiple Data (MPMD) mode, with commands used to start computations and remote APGAS places differing. A range of the number of nodes used to run a given number of threads was tested and the best-achieved results are presented. Due to APGAS's requirements, Oracle JDK 10.0.2 was used in all cases. The tests use 3.3 MB UTF-8 encoded text of English translation of Tolstoy's *War and Peace* as a textual corpus for word counting code. They were performed in a strong scalability regime, with the input file being read 4096 times and all threads reading the same file. The file content is not preloaded into the application memory before the benchmark.

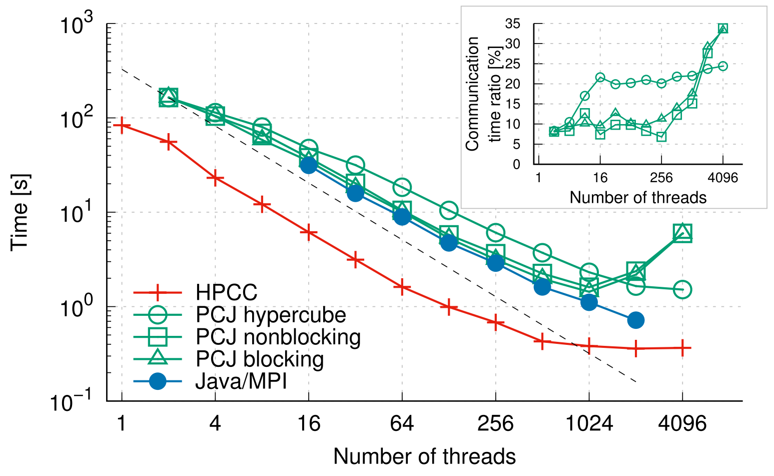

Fig. 6 Performance of the 1D FFT implemented using different algorithms. The data for the HPCC implementation using C/MPI as well as Java/MPI is presented for reference. The array of 2^{30} elements was used. Ideal scaling for PCJ is drawn (dashed line). Inset presents communication time as a percentage of total execution time. The benchmarks were run on Cray XC40 at ICM

The performance of the reduction phase is key for the overall performance [10] and the best results in case of PCJ are obtained using binary tree communication. APGAS solution uses the reduction as implemented in [25] (this work reports the worse performance of PCJ, due to the use of simpler and therefore less efficient reduction scheme).

The results presented in Fig. 7 show good scalability of the PCJ implementation. PCJ's performance was better when compared to APGAS, which can be tracked to the PCJ's reduction implementation. Regarding native code, C++ was chosen as a better-suited language for this task than C, because of its built-in map primitives and higher level string manipulation routines. While C++ code scales ideally, its poor performance when measured in absolute time can be traced back to the implementation of line-tokenizing. All the codes (PCJ, APGAS, C++), in line with our earlier works [5], consistently use regular expressions for this task.

One should note, that different set of results obtained on the Hadoop cluster shows that PCJ implementation is at least 3 times faster than Hadoop one [5] and Spark one [25].

Artificial Intelligence workloads

AI is currently a vibrant area of research, gaining a lot from advances in the processing capabilities of modern hardware. The PCJ library was tested in the area of artificial intelligence to ensure that it provides AI workloads with sufficient processing potential, able to exploit the future exascale systems. In this respect, two types of workloads were considered. Firstly, stemming from the traditional mode of AI research aimed at discovering the inner working of real physiological systems, the library was used to aid the researchers in the task of modeling the *C. Elegans* neuronal circuity. Secondly, it was used to power the training of the modern artificial neural network, distributing the gradient descent calculations.

Neural networks—modeling the connectome of C. Elegans

Nematode *C. Elegans* is a model organism whose neuronal development has been studied extensively and remains the only organism with a fully known connectome. There are currently some experiments that aim to link its structure and actual worm's behavior. In one of those experiments, worm's motoric neurons were ablated using a laser, affecting the changes of its movement patterns [36]. The results of those experiments allowed to create a mathematical model of the relevant connectome fragment by a biophysics expert. The model was defined by a set of ordinary differential equations, with 8 parameters.

The value of those parameters was key to the model's accuracy, yet they were impossible to calculate using the traditional numerical or analytical methods. Therefore a differential evolution algorithm was used to explore the solution space and fit the model's parameters so that its predictions are in line with the empirical data. The mathematical model has been implemented in Java and parallelized with the use of the PCJ library [36, 37]. It should be noted that the library allowed to rapidly (ca. 2 months) prototype the connectome model and align it according to the shifting requirements of the biophysics expert.

In regards to the implementation's performance, Fig. 8 can be consulted, where it is expressed as a number of tested configurations per second. The experimental dataset amounted to a population of 5 candidate vectors affiliated with each thread that was evaluated through 5 iterations in a weak scaling regime. A scaling close to the ideal was achieved both irrespective of the hyperthreading status, as its overhead in this scenario is minimal. The outlier visible in the case of 192 threads is most probably due to the stochastic nature of the differential evolution algorithm and disparities regarding model evaluation time for concrete sets of parameters.

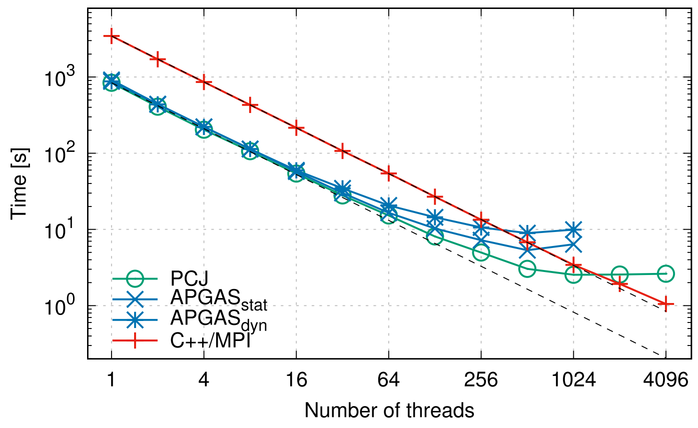

Fig. 7 Strong scalability of WordCount application implemented in PCJ and APGAS. APGAS$_{stat}$ and APGAS$_{dyn}$ denote original fork-join algorithm and algorithm using lifeline-based global load balancing. C++/MPI results are plotted for reference. Ideal scaling for PCJ and C++/MPI are drawn with the dashed lines. The results are presented for 3.3 MB file and were obtained using Cray XC40 at ICM. The APGAS results were not calculated for a number of nodes greater than 1024 due to long startup time exceeding one hour (which can be attributed to the library's bootstrapper which—at the time of running the experiments—was not generic enough to offer a fast startup on a range of systems)

Distributed neural network training

The PCJ library was also tested in workloads specific to modern machine learning applications. It was successfully integrated with TensorFlow for the distribution of gradient descent operation for effective training of neural networks [38], performing very well against the Python/C/MPI-based state-of-the-art solution, Horovod [39].

For presentation purposes, a simple network consisting of three fully connected layers (sized 300, 100 and 10 neurons respectively [40]) was trained for handwritten digits recognition for 20 epochs (i.e. for a fixed number of iterations) on MNIST dataset [41] (composed of 60,000 training images of which 5000 were set aside for validation purposes in this test), with mini-batch consisting of 50 images. PCJ tests two algorithms. The first one uses the same general idea for gradient descent calculations as Horovod (i.e. data-parallel calculations are performed process-wise, and the gradients are subsequently averaged after each mini-batch). The second one implements asynchronous parallel gradient descent as described in [42].

Implementation-wise, Horovod works by supplying the user with simple to use Python package with wrappers and hooks that allow enhancing existing code with distributed capabilities and MPI is used for interprocess communication. In the case of PCJ, a special runner was coded in Java with the use of TensorFlow's Java API for the distribution and instrumentation of training calculations. Relevant changes had to be implemented in Python code as well. Our code implements the reduction operation based on the hypercube allreduce algorithm [43].

The calculations were performed using the Cray XC40 system at ICM with Python 3.6.1 installed alongside TensorFlow v. 1.130-rc1. Horovod was installed with the use of Python's pip tool version 0.16.0. SLURM was used to start distributed calculations, with one TensorFlow process per node. We have used 48 working threads per node.

Results in strong scalability regime presented in Fig. 9 show that the PCJ implementation that facilitates asynchronicity is on a par with MPI-based Horovod. In the case of smaller training data sizes when a larger number of nodes is used, our implementation is at a disadvantage in terms of accuracy. This is because the overall calculation time is small and communication routines are not able to finish in time before thread finish local training. The datapoint for 3072 threads (64 nodes) was thus omitted for asynchronous case in Fig. 9. Achieving full performance of Horovod on our cluster was only possible after using non-standard configuration for available TensorFlow installation. This in turn allowed to fully exploit inter-node parallelism with the use of Math Kernel Library (MKL). TensorFlow for Java available as a Maven package did not exhibit the need for this fine-tuning, as it does not use MKL for computation.

Presented results clearly show that PCJ can be efficiently used for parallelization of AI workloads. Moreover, use of Java language allows for easy integration with existing applications and frameworks. In this case PCJ allowed for easier deployment of most efficient configuration of TensorFlow on HPC cluster.

Future work

From the very beginning, the PCJ library has been using sockets for transferring the data between nodes. This design was straightforward, however, it precludes the full utilization

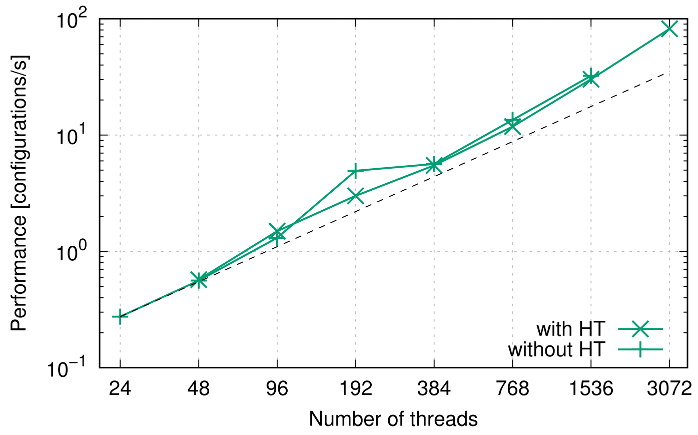

Fig. 8 Performance of the evolution algorithm to search parameters of neural network simulating connectome of *C. Elegans*. The performance data for execution with and without hyperthreading is presented. The benchmarks were run on Cray XC40 at HLRS. Ideal scaling is drawn with the dashed line

of the novel communication hardware such as Cray Aries or InfiniBand interconnects. There is ongoing work to use novel technologies in PCJ. This is especially important for network-intensive applications. However, we are looking for Java interfaces that can simplify integration. DiSNI [44] or jVerbs [45] seems to be a good choice, however, both are based on the specific implementation of communication and their usage by the PCJ library is not easy. There are also attempts to speed up data access in Java using Remote Direct Memory Access (RDMA) technology [46, 47]. We are investigating how to use it in the PCJ library.

Another reason for low communication performance is the problem of data copying during the send and receive process. This cannot be avoided due to the Java design: technologies based on the zero-copy and direct access to the memory do not work in this case. This is an important issue not only for the PCJ library but for Java in general.

As one of the main principles of the PCJ library is not to depend on adding any additional library, PCJ uses a standard Java object serialization mechanism to make a complete copy of an object. There are undergoing works that would allow using external serialization or cloning libraries, like Kryo, that could speed up making a copy of data.

The current development of the PCJ library is focused on the code execution on the multiple, multicore processors. Whilst Cray XC40 is representative for most of the current TOP500 systems, of which only 20% are equipped with Graphics Processing Units (GPUs), the peta- and exascale systems are heterogeneous, and in addition to the CPU's nodes contains accelerators such as GPUs, Field-Programmable Gate Arrays (FPGAs), and others. The PCJ library supports accelerators through JNI mechanisms. In particular one can use JCuda to run Compute Unified Device Architecture (CUDA) kernels on the accelerators. This mechanism has been checked experimentally, the performance results are in preparation. Similarly, nothing precludes the already existing PCJ-TensorFlow code from using TensorFlow's GPU exploitation capabilities.

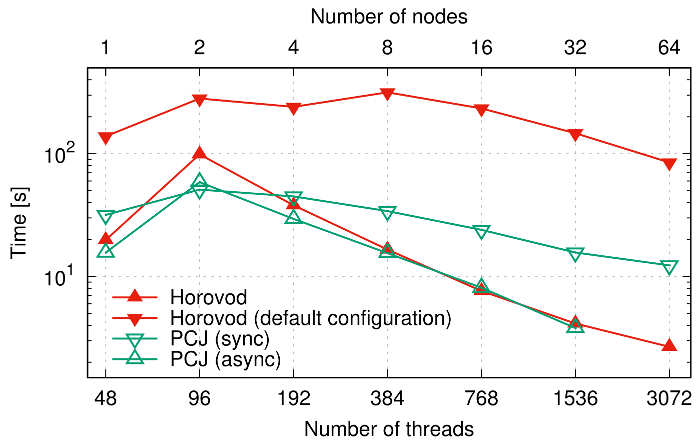

Fig. 9 Comparison of the distributed training time taken by Horovod and PCJ as measured on Cray XC40 at ICM. Accuracy of $\approx$ 96% was achieved

Conclusion

Near perspective of exascale systems and a growing number of petascale computers makes strong interest in new, more productive programming tools and paradigms capable of developing codes for large systems. At the same time, we observe a change in the type of workloads run on supercomputers. There is a strong interest in running Big Data processing or Artificial Intelligence applications. Unfortunately, the majority of the new workloads are not well suited for large computers. They are implemented in languages like Java or Scala which, up to now, were out of interest of the HPC community.

In this paper, we performed a brief review of the programming languages and programming paradigms getting attention in the context of HPC, Big Data and AI processing. We focused on Java as the most widely used programming language and presented its feasibility to implement AI and Big Data applications for large scale computers.

As presented in the paper, the PCJ library allows for easy development of highly scalable parallel applications. Moreover, PCJ puts great promise to be successful for the parallelization of HPC workloads as well as AI, and Big Data applications. Example applications and their scalability and performance have been reported in this paper.

Results presented here, and in previous publications, clearly show the feasibility of Java language to implement parallel applications with a large number of threads. The PGAS programming model allows for easy implementation of various parallel schemas, including traditional HPC as well as Big Data and AI, ready to run on peta- and exascale systems.

The proposed solution will open up new possibilities of applications. Java as the most popular programming language is widely used in business applications. The PCJ library allows, with little effort, to extend the application to include computer simulations, data analysis and artificial intelligence. The PCJ library allows to easily develop applications and run them on a variety of resources from personal workstation computers to cloud resources. The key element is the ease of extending existing applications and the integration of various types of processing while maintaining the advantages offered by Java.

This solution is very important for existing applications and allows for easy and quick adaptation to the growing demand.

Abbreviations

AI: Artificial Intelligence; API: Application Programming Interface; CPU: Central Processing Unit; CUDA: Compute Unified Device Architecture; FFT: Fast Fourier Transform; FPGA: Field-Programmable Gate Array; GC: Garbage collector; GPU: Graphics processing unit; HPC: High-performance computing; JIT: Just-in-Time; JVM: Java Virtual Machine; MKL: Math Kernel Library; MPI: Message-Passing Interface; MPMD: Multiple Programs, Multiple Data; PCJ: Parallel Computing in Java; PGAS: Partitioned Global Address Space; RDMA: Remote Direct Memory Access; RMI: Remote method invocation; SSH: Secure shell.

Acknowledgements

The authors gratefully acknowledge the support of Dr. Alexey Cheptsov and the computer resources and technical support provided by HLRS. We acknowledge PRACE for awarding us access to resource Hazel Hen based in Germany at HLRS under a preparatory access project no. 2010PA4009 and HPC-Europa3 program under application no. HPC17J7Y4M. We acknowledge Interdisciplinary Centre for Mathematical and Computational Modelling (ICM) University of Warsaw for providing computational resources under grants no. GB65-15, GA69-19.

Authors' contributions

MN was responsible for the description of the PCJ library and preparing and running a 2D stencil (Game of Life) benchmark and interpreting the results as well as was a major contributor in writing the manuscript. ŁG was responsible for preparing other presented applications, collecting and interpreting their performance results, and writing the description in the manuscript. PB contributions include the original idea, literature review, interpreting the results and writing conclusions. All authors read and approved the final manuscript.

Funding

This research was carried out with the partial support of EuroLab-4-HPC as cross-site collaboration grant and HPC-Europa3 program (application no. HPC17J7Y4M) for visiting HLRS, Germany.

Availability of data and materials

All the source codes are included on the websites listed in the ***References*** section.

Declarations

Competing interests

All authors declare that they have no competing interests.

Author details

[1]Faculty of Mathematics and Computer Science, Nicolaus Copernicus University in Toruń, ul. Chopina 12/18, 87-100 Toruń, Poland. [2]Interdisciplinary Centre for Mathematical and Computational Modeling, University of Warsaw, ul. Tyniecka 15/17, 02-630 Warsaw, Poland.

Received: 29 January 2021 Accepted: 13 April 2021
Published online: 26 April 2021

References

1. Hadjidoukas P, Bartezzaghi A, Scheidegger F, Istrate R, Bekas C, Malossi A. torcpy: Supporting task parallelism in Python. SoftwareX. 2020;12:100517.
2. Nowicki M, Bała P. Parallel computations in Java with PCJ library. In: 2012 International Conference on High Performance Computing & Simulation (HPCS). IEEE; 2012. p. 381–387.
3. Almasi G. PGAS (Partitioned Global Address Space) Languages. In: Padua D, editor. Encyclopedia of Parallel Computing. Boston: Springer; 2011. p. 1539–45.
4. Challenge Awards HPC, Competition: Awards, . Awards: Class 2. 2014. . http://www.hpcchallenge.org/custom/index.html?lid=103&slid=272. Accessed 29 Jan 2021.
5. Nowicki M, Ryczkowska M, Górski Ł, Bała P. Big Data Analytics in Java with PCJ Library: Performance Comparison with Hadoop. In: International Conference on Parallel Processing and Applied Mathematics. Springer; 2017. p. 318–327.
6. Nowicki M. Comparison of sort algorithms in Hadoop and PCJ. J Big Data. 2020;7:1. https://doi.org/10.1186%2Fs40537-020-00376-9
7. 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. Computational Intelligence Laboratory, Zhengzhou University, Zhengzhou China and Technical Report, Nanyang Technological University, Singapore. 2013.
8. Nowicki M, Bzhalava D, Bała P. Massively Parallel Sequence Alignment with BLAST Through Work Distribution Implemented Using PCJ Library. In: International Conference on Algorithms and Architectures for Parallel Processing. Springer; 2017. p. 503–512.
9. Nowicki M, Bzhalava D, Bała P. Massively parallel implementation of sequence alignment with basic local alignment search tool using parallel computing in java library. J Comput Biol. 2018;25(8):871–81.

10. Nowicki M, Górski Ł, Bała P. Performance evaluation of parallel computing and Big Data processing with Java and PCJ library. Cray Users Group. 2018.
11. Vincent P, Witherden F, Vermeire B, Park JS, Iyer A. Towards green aviation with python at petascale. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Press; 2016. p. 1.
12. Johnson N. Python import scaling; 2014. Accessed: 29.01.2021. http://www.archer.ac.uk/documentation/white-papers/dynamic-import/ARCHER_wp_dynamic-import.pdf.
13. Kincade K. Celeste: A New Model for Cataloging the Universe; 2015. https://newscenter.lbl.gov/2015/09/09/celeste-a-new-model-for-cataloging-the-universe/. Accessed 29 Jan 2021.
14. Carpenter B, Getov V, Judd G, Skjellum A, Fox G. MPJ: MPI-like message passing for Java. Concurrency. 2000;12(11):1019–38.
15. Vega-Gisbert O, Roman JE, Squyres JM. Design and implementation of Java bindings in Open MPI. Parallel Comput. 2016;59:1–20.
16. Bonér J, 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.
17. Nester C, Philippsen M, Haumacher B. A more efficient RMI for Java. In: Java Grande. vol. 99; 1999. p. 152–159.
18. Yelick K, Semenzato L, Pike G, Miyamoto C, Liblit B, Krishnamurthy A, et al. Titanium: a high-performance Java dialect. Concurr Comput. 1998;10(11–13):825–36.
19. Feind K. Shared memory access (SHMEM) routines. Cray Research. 1995.
20. Nieplocha J, Harrison RJ, Littlefield RJ. Global arrays: A nonuniform memory access programming model for high-performance computers. J Supercomput. 1996;10(2):169–89.
21. Kale LV, Zheng G. 13. In: Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects. Wiley; 2009. p. 265–282. https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470558027.ch13.
22. Carlson WW, Draper JM, Culler DE, Yelick K, Brooks E, Warren K. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences; 1999.
23. Reid J. The new features of Fortran 2008. In: ACM SIGPLAN Fortran Forum. vol. 27. ACM; 2008. p. 8–21.
24. Chamberlain BL, Callahan D, Zima HP. Parallel Programmability and the Chapel Language. Int J High Perf Comput Appl. 2007;21(3):291–312.
25. Posner J, Reitz L, Fohry C. Comparison of the HPC and Big Data Java Libraries Spark, PCJ and APGAS. In. IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM). IEEE. 2018;2018:11–22.
26. Nowicki M, Górski Ł, Bała P. PCJ–Java Library for Highly Scalable HPC and Big Data Processing. In: 2018 International Conference on High Performance Computing & Simulation (HPCS). IEEE; 2018. p. 12–20.
27. Nowicki M, Górski Ł, Bała P. Evaluation of the Parallel Performance of the Java and PCJ on the Intel KNL Based Systems. In: International Conference on Parallel Processing and Applied Mathematics. Springer; 2017. p. 288–297.
28. Trader T. It's Official: Aurora on Track to Be First US Exascale Computer in 2021. HPC Wire. 2019;(March 18).
29. Gardener M. Mathematical Games: The fantastic combinations of John Conway's new solitaire game "Life". Sci Am. 1970;223(4):120–3.
30. PCJ implementations of the *Game of Life* benchmark;. Accessed: 29.01.2021. https://github.com/hpdcj/PCJ-examples/blob/3abf32f808fa05af2b7f1cfd0b21bd6c5efc1339/src/org/pcj/examples/GameOfLife.java.
31. Takahashi D, Kanada Y. High-performance radix-2, 3 and 5 parallel 1-D complex FFT algorithms for distributed-memory parallel computers. J Supercomput. 2000;15(2):207–28.
32. Luszczek PR, Bailey DH, Dongarra JJ, Kepner J, Lucas RF, Rabenseifner R, et al. The HPC Challenge (HPCC) Benchmark Suite. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing. SC '06. New York, NY, USA: ACM; 2006.
33. Mellor-Crummey J, Adhianto L, Jin G, Krentel M, Murthy K, Scherer W, et al.. Class II submission to the HPC Challenge award competition Coarray Fortran 2.0. Citeseer.
34. PCJ implementations of the *FFT* benchmar. https://github.com/hpdcj/hpc-challenge-fft/tree/ebd557e40ad50f614a869000321ee822b67d2623. Accessed 29 Jan 2021.
35. PCJ implementations of the *WordCount* application. https://github.com/hpdcj/wordcount/tree/6a265bc92147a89c37176692ccae8dcf8d97df72. Accessed 29 Jan 2021.
36. Rakowski F, Karbowski J. Optimal synaptic signaling connectome for locomotory behavior in Caenorhabditis elegans: Design minimizing energy cost. PLoS Comput Biol. 2017;13(11):e1005834.
37. PCJ implementations of the modeling the connectome of *C. Elegans* application. https://github.com/hpdcj/evolutionary-algorithm/tree/602467a7947fd3da946f70fd2fae646e2f1500da. Accessed 29 Jan 2021.
38. PCJ implementations of the distributed neural network training application. https://github.com/hpdcj/mnist-tf/tree/77fa143e2aa3b83294a8fc607b382c518d4396d7/java-mnist. Accessed 29 Jan 2021.
39. Sergeev A, Balso MD. Horovod: fast and easy distributed deep learning in TensorFlow. arXiv preprint arXiv:180205799. 2018.
40. Géron A. Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems. O'Reilly Media, Inc.; 2017.
41. LeCun Y, Cortes C, J C Burges C. The MNIST Database of handwritten digits. http://yann.lecun.com/exdb/mnist/. Accessed 17 Mar 2021.
42. Keuper J, Pfreundt FJ. Asynchronous parallel stochastic gradient descent: A numeric core for scalable distributed machine learning algorithms. In: Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments. ACM; 2015. p. 1.
43. Grama A, Kumar V, Gupta A, Karypis G. Introduction to parallel computing. Pearson Education; 2003.
44. IBMCode. Direct Storage and Networking Interface (DiSNI); 2018. https://developer.ibm.com/technologies/analytics/projects/direct-storage-and-networking-interface-disni/. Accessed 29 Jan 2021.
45. IBM. The jVerbs library; 2012. https://www.ibm.com/support/knowledgecenter/en/SSYKE2_8.0.0/com.ibm.java.80.doc/docs/rdma_jverbs.html. Accessed 29 Jan 2021.
46. Biswas R, Lu X, Panda DK. Accelerating TensorFlow with Adaptive RDMA-Based gRPC. In: 2018 IEEE 25th International Conference on High Performance Computing (HiPC). IEEE; 2018. p. 2–11.

47. Lu X, Shankar D, Panda DK. Scalable and distributed key-value store-based data management using RDMA-Memcached. IEEE Data Eng Bull. 2017;40(1):50–61.
48. Nowicki M, Ryczkowska M, Górski Ł, Szynkiewicz M, Bała P. PCJ-a Java library for heterogenous parallel computing. Recent Advances in Information Science (Recent Advances in Computer Engineering Series vol 36), WSEAS Press. 2016;p. 66–72.