PROCEEDINGS OF SCIENCE

Scalable computing in Java with PCJ Library. Improved collective operations

Marek Nowicki,[a,b,*] **Łukasz Górski**[b] **and Piotr Bała**[b]

[a] *Faculty of Mathematics and Computer Science, Nicolaus Copernicus University in Toruń, Poland*
[b] *Interdisciplinary Centre for Mathematical and Computational Modeling, University of Warsaw, Poland*

E-mail: faramir@mat.umk.pl, lgorski@icm.edu.pl, bala@icm.edu.pl

Machine learning and Big Data workloads are becoming as important as traditional HPC ones. AI and Big Data users tend to use new programming languages such as Python, Julia, or Java, while the HPC community is still dominated by C/C++ or Fortran. Hence, there is a need for new programming libraries and languages that will integrate different applications and allow them to run on large computer infrastructure. Since modest computers are multinode and multicore, parallel execution is an additional challenge here.

For that purpose, we have developed the PCJ library, which introduces parallel programming capabilities to Java using the Partitioned Global Address Space model. It does not modify language nor running environment (JVM). The PCJ library allows for easy development of parallel code and runs it on laptops, workstations, supercomputers, and the cloud.

This paper presents an overview of the PCJ library and its usage in parallelizing selected workloads, including HPC, AI, and Big Data. The performance and scalability are presented. We present recent addition to the PCJ library, which are collective operations. The collective operations significantly reduce the number of lines of code to write, ensuring good performance.

International Symposium on Grids & Clouds 2020, ISGC2021
22-26 March, 2021
Academia Sinica, Taipei, Taiwan (online)

*Speaker

PoS(ISGC2021)007

1. Introduction

Recent years are associated with the increasing utilization of clusters, supercomputers, and computer clouds. In addition to the computations based on the mathematical models, we observe increasing Artificial Intelligence and Big Data processing. Both are getting an increasing amount of computing power and other computer resources. Unfortunately, the software is hardly adapted to run on typical supercomputers. Users running machine learning and Big Data workloads use new programming languages not considered for use in HPC, like Python, Julia, or Java, while C/C++ or Fortran still dominates the HPC community. Hence, there is a need for new programming libraries and languages that will integrate different applications and efficiently utilize large computer infrastructures. Since modest computers are multinode and multicore, parallel execution is an additional challenge here.

Our approach is to use Java, a well-established programming language, and extend it by an easy-to-use library for parallel execution. Java is popular in many areas, including Big Data and AI. Moreover, due to Java portability, the user can develop a solution on his laptop or workstation and then move to the HPC and cloud infrastructures.

For that purpose, we have developed the PCJ library [1]. The PCJ library introduces parallel programming capabilities using the Partitioned Global Address Space model. Most importantly, it does not require modifications in language syntax nor the running environment. The PCJ library allows for parallelization of Java code and allows to run them on different computational systems practically without recompilation.

In this paper, we present an overview of the PCJ library and its usage in the parallelization of selected workloads from HPC, AI, and Big Data domains. The performance and scalability are shown. We present collective operations, which are a recent addition to the PCJ library. The collective operations significantly reduce the number of lines of code to write, ensuring good performance. Recent improvements in collective operations are presented.

The rest of the paper is organized as follows. After an introduction to various programming paradigms (Section 2), we briefly present the PCJ library (Section 3). Section 4 contains a description of recently introduced collectives methods. Section 5 shows the performance of the library in various workload schemes: HPC Workload (Fast Fourier Transform), Artificial Intelligence Workloads (Distributed Neural Network Training), and Big Data Workload (WordCount). A final outlook and concluding remarks are presented in Section 6.

2. Programming paradigms

Supercomputer systems pose challenges more than what we have for current systems. For programming models and systems, the key indicators are simplicity and performance. There is also a requirement to be open towards new application areas not present in HPC up to now.

Current HPC programming is dominated by the threaded execution of the parallel code based on the shared memory (i.e. OpenMP) or by distributed memory model utilizing the passing of communicates (i.e. MPI). Since both approaches have advantages and disadvantages, hybrid modes such as MPI+X have been developed. The additional programming tools are still not commonly adopted since there are various them for CPUs and for accelerators such as GPU, TFU, or FPGA.

2.0.1 PGAS

Recently, programming models based on Partitioned Global Address Space (PGAS) are gaining popularity. PGAS provides programmers with an easy-to-use set of constructions, functions, or methods that allow implementing any parallel algorithm. One-sided asynchronous communication allows for data exchange between threads and the easy overlay of simulations and communication. Users point out that programming in the PGAS model is more productive than in other parallel paradigms and allows for faster and less error-prone software development. It is expected that PGAS languages will be more important at the exascale because of the distinct features and development efforts which is lower than for other approaches. PGAS languages can be implemented either as a single system or even for hybrid programming with MPI.

The PGAS model can be supported by a library such as SHMEM [2], or Global Arrays [3] or can be supported by a language, such as UPC [4] or Fortran [5]. 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, the PGAS programming model has not had a successful realization in the Java language. There exists the PGAS implementation in a Java-like language called Titanium [6]. However, it modifies the language syntax and uses a dedicated compiler, making it challenging to follow recent Java language and syntax changes. The PCJ library, described in Section 3, is a working implementation of the PGAS model for the Java language. It provides a convenient API for writing parallel codes and gives good scalability with reasonable performance.

2.0.2 Map-Reduce

Map-Reduce [7] is a well-known parallel programming paradigm that became popular with the availability of parallel file systems such as Hadoop [8, 9]. The computational kernel is based on the distribution of the data, calculations performed independently on the distributed data, and then a reduction of the results by, for example, summing up partial results.

The mapping of the data can be performed outside the main application by saving data to the local storage or by using a parallel file system. The reduction step requires an exchange of data between nodes and involves communication.

The Map-Reduce model is easy to use by the user with limited or even practically no experience in parallel or distributed computing. Unfortunately, there are some important drawbacks such as low performance, limited scalability, and the need to modify the user's algorithm in such a way that it fits the Map-Reduce paradigm.

There have been some attempts to improve Map-Reduce performance. One of them is Apache Spark which allows for in-memory data processing. As a result, a higher performance is achieved; unfortunately, it is still lower than for other Java-based solutions. In particular, it has been demonstrated that the performance of applications built using the PCJ library or APGAS (a branch of IBM's X10 language project) is higher than for Hadoop/Spark implementations [10, 11]. For the calculation-intensive and communication-intensive HPC workloads, PCJ is several up to hundreds of times faster [12].

2.1 Actor programming model

The actor programming model uses agents to perform operations. Its realization in Java is available thanks to the Akka library [13]. A similar solution, although not such popular, has been developed for Python. Akka uses the actor model to overcome the limitations of traditional object-oriented programming models and meet the unique challenges of highly distributed systems. Agents provide an asynchronous change in individual locations. Agents are bound to a single storage location for their lifetime and only allow mutation of that location (to a new state) to occur as a result of an action.

The afore discussed short review of the programming models for peta- and exascale HPC shows that PGAS and its realization in Java is one of the most promising solutions. Moreover, Java is still one of the most popular languages and has a large user base. Java provides many tools for developers to organize, test, and maintain code not present for other languages. Another essential feature is the wide availability of Java, which can be run on most of the devices of interest. In the case of the PCJ library, the programmer can use standard tools and environments. The application can be developed and tested on the laptop or developer's workstation and then moved to the large HPC or cloud systems without modifications. Due to the Java binary compatibility, there is even no need to recompile the code.

3. The PCJ Library

PCJ [1, 14] is an HPC Challenge award-winning Java library for high-performance parallel computing. The library source code is available under the BSD license [15]. PCJ is a standard Java library that does not introduce any language syntax changes nor requires a special compiler. The user has only to download a small (about 240 KB) single jar file, and then the user can develop and run parallel applications on any system with Java installed. Alternatively, the user can use Maven [16, 17], Gradle [18, 19], or other dependency resolution systems to download the library, as the library is placed in the Maven Central Repository (*group-id*: `pl.edu.icm.pcj`, *artifact-id*: `pcj`).

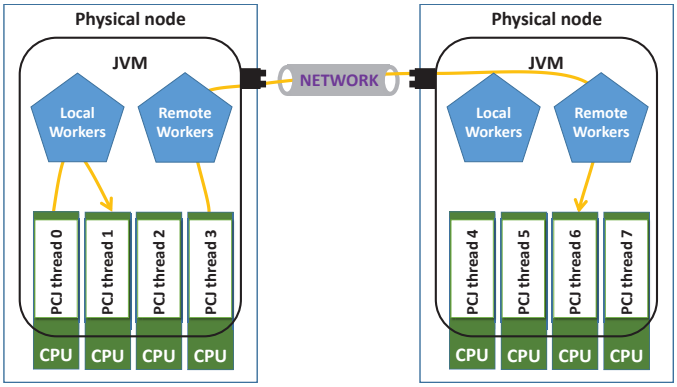

Figure 1: Diagram of PCJ communication model [20]. Arrows present local (inside JVM) and remote (through the network) communication.

The PCJ library is designed to support the application running on multicore, multinode systems. It takes care of the technical details hiding them from the users. As with most of the other solutions, the library assumes that a number of cores used to execute the application are given at startup and do not change during execution. This simplifies parallelization of the algorithm, data distribution and makes implementation of the algorithm easier. PCJ allows running multiple threads in a single JVM, multiple JVMs on a single physical node, and multiple JVMs with multiple threads on multiple physical nodes. The communication between threads inside a single JVM utilizes a local worker who serializes and deserializes messages to ensure that the data is deeply copied. As presented in Figure 1, the communication between JVMs on single or multiple nodes is realized using the TCP/IP connection.

PCJ takes care of application deployment in various situations. As shown in Listing 1, four threads are started. Two localhost threads that share the same default port number (one implicitly set and one explicitly set to 8091) will be run on the same virtual machine that executes the code. Additional JVMs will be spawned in the case of the other two threads – one on the local machine and one on the remote host (on *example.com* domain). A larger number of machines can be supplied by chaining more `addNode(...)` methods or by calling `addNodes(...)` method providing an array with hostnames or name of a file with the list of hostnames (one hostname per line).

PCJ uses an SSH connection to start JVM on a remote node. The connection uses *BatchMode* option that disables the passphrase/password querying. It is necessary to configure nodes to allow for logging in without any prompt, e.g. by using authentication keys. The deploy mechanism is mostly used for testing on local machine or running application on the cloud systems [21].

There is `start()` method that is used with the execution schema available on the clusters or supercomputers for starting an application on job allocated nodes (i.e. `mpiexec`, `srun`, `aprun` commands). There is also ongoing work to integrate other job scheduling mechanisms like YARN for launching the application on Apache Hadoop clusters [22].

In the PCJ library, the threads are represented by classes that implement the *StartPoint* interface. The `public void main() throws Throwable` method corresponds to the `run()` method of the standard *Thread* class, or more precisely, to the `public void run()` method from *Runnable* interface. It allows the user not to catch checked exception from the code like the standard entry point `public static void main(String[] args)` method. All fields declared in the class implementing a *StartPoint* interface are local to the particular thread. However, some fields of a class can be communicated (shared) between tasks. These fields are called *shareable variables*. Figure 2 shows the diagram of the PCJ computing model, a division of the variables into shareable and local variables, and the possible exchange of data directions.

The main attribute of the PGAS programming model is the locality of all variables and the possibility to share variables between tasks. A class field can be marked for remote addressing with the use of `@Storage` and `@RegisterStorage` annotations.

The `@Storage` annotation coupled with a custom enumeration allows referencing a shareable variable by name in a safe manner. Each enum's constant points to a field to mark it as *shareable*. The type of the variable is inferred from the type of the associated field. The type may not be serializable, as `Object` class or `Map` interface, but the stored object must be serializable to exchange data between threads.

The `@RegisterStorage` annotation allows registering every enumeration and storage associated

```java
import org.pcj.PCJ;
import org.pcj.RegisterStorage;
import org.pcj.StartPoint;
import org.pcj.Storage;

@RegisterStorage(PcjExample.Shareable.class)
public class PcjExample implements StartPoint {

    @Storage(PcjExample.class)
    enum Shareable { count, n }
    private long count;
    private int n;

    public static void main(String[] args)
                            throws java.io.IOException {
        PCJ.executionBuilder(PcjExample.class)
            .addNode("localhost")
            .addNode("localhost:8091")
            .addNode("localhost:9095")
            .addNode("example.com")
            .deploy();
    }

    @Override
    public void main() throws Throwable {
        if (PCJ.myId() == 0)
           PCJ.broadcast(1_000_000_000, Shareable.n);
        PCJ.waitFor(Shareable.n);
        int nl = (n + PCJ.threadCount() - (PCJ.myId() + 1))
                                / PCJ.threadCount();
        java.util.Random r = new java.util.Random();
        for (int i = 0; i < nl; i++) {
            double x = r.nextDouble();
            double y = r.nextDouble();
            if (Math.hypot(x,y) <= 1.0) ++count;
        }
        PCJ.barrier();
        if (PCJ.myId() == 0) {
            long total = PCJ.reduce(Long::sum, Shareable.count);

            System.out.println(4.0 * total / n);
        }
    }
}
```

Listing 1: Application to estimate π value using Monte Carlo method. `PCJ.reduce(...)` is used to sum up partial results calculated by each PCJ thread.

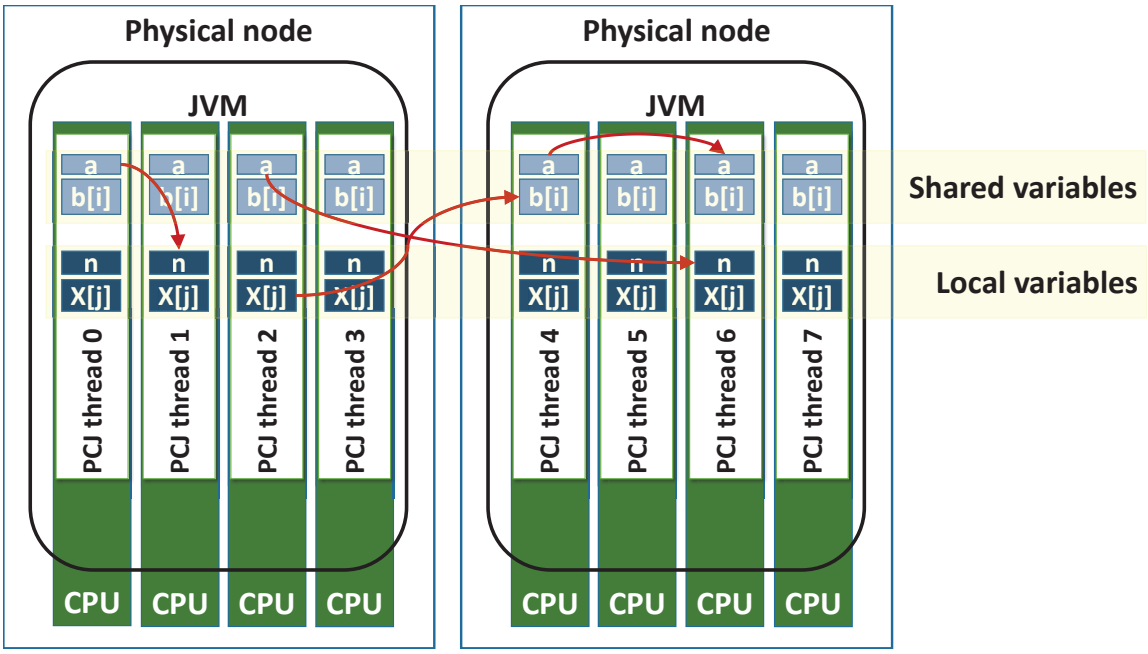

Figure 2: Diagram of PCJ computing model [23]. Arrows present possible communication using `put(...)` or `get(...)` methods acting on shared variables.

with it to the thread that owns a given shareable variable. Other threads can access the shareable variable by providing relevant registered enum's constant name but do not have to register the enum itself. With the `@RegisterStorage` annotation storages are automatically registered during the application start phase on the actual `StartPoint` class. Another option is the usage of the *PCJ*`.registerStorage(...)` method during runtime that allows for greater flexibility. In practice, library users tend to use annotation-based registration.

Two PGAS most basic operations – read and modification of shareable variable – are implemented as `get(int threadId, Enum<?> name)` and `put(T newValue, int threadId, Enum<?> name)` methods. The methods' arguments are an identifier of target thread and shareable variable. In the case of the `put(...)` method, the first argument is the variable's new value. Both methods can be used with the optional parameter *indices*, which can be used only for arrays, to indicate the specified element in the array to be communicated. PCJ offers `asyncGet(...)` and `asyncPut(...)` methods which are asynchronous versions of the `put(...)` and `get(...)` with the same parameters. The communication is asynchronous and can be overlapped with the other operations. The `get(...)` method invoked on `PcjFuture<?>` object can be performed at any time but complete only if data has arrived. PCJ also supports `get(long timeout, TimeUnit unit)` method that causes waiting, if necessary, for arriving data or throwing a *TimeoutException* if data is not available before the timeout occurs.

One of the newest function implemented in the PCJ library is `accumulate(ReduceOperation<T> function, T newValue, int threadId, Enum<?> name)` method. It works in a very similar way to `put(...)`, but instead of replacing the stored value in the shareable variable, it performs a reduction operation on the stored value and sent one. The asynchronous `asyncAccumulate(...)` method also exists.

As the library is based on one-sided communication, the methods allowing for variable monitoring were introduced. In particular, the `waitFor(Enum<?> name)` method blocks the caller until the variable it monitors is changed by another thread. Moreover, the `waitFor(Enum<?> name, int count)` method is also available. The count parameter indicates the number of changes the task is waiting for. The method returns the number of changes that were not *waited for*. It is possible to clear the counter of changes using `monitor(Enum<?> name)` method.

The `barrier()` method implements the classical synchronization primitive known from other PGAS libraries and languages. It blocks the threads until all come to the synchronization point in the program. A two-point version of the barrier that synchronizes only the selected two threads is also supported. Moreover, there is a non-blocking barrier method `asyncBarrier()` that allows for proceeding with execution and checking if all threads have passed the synchronization point.

4. Collectives in PCJ

Collective communication primitives come along with parallel programming tools and have been popularized by MPI. Common examples of collective operations are: *gather* (in which data is collected from all nodes), *scatter* (in which a set of data is broken up into pieces, and a different piece is sent to different nodes), and *broadcast* (in which the same data is sent to all nodes). The gather operation is usually associated with some operation performed on gathered data such as sum, multiplication, or taking the minimum or maximal value. Therefore in many cases, gather operation is replaced by reduction, which can be optimized and presents better performance.

A global view aspect to collective communication makes them attractive to PGAS languages since some algorithms can be trivially expressed with, for example, an all-to-all communication. Most PGAS languages provide users with broadcast, but other collective operations usually are not available as methods or procedures. Therefore it is up to the user to implement collectives. Optimized and scalable collective operations are a crucial performance factor. An example of such a solution is an implementation of all to all communication in 1-dimensional FFT presented in section 5.1.

To address this problem, we decided to extend the PCJ library with a number of collective methods. We tried to make them as generic and possible keeping method call simple and easy to use by the programmer. The list of the collectives is not closed and could be extended in the future based on the users' experience.

The PCJ library from the very beginning implements the broadcast operation: `broadcast(T newValue, Enum<?> name)`. The caller thread uses the `broadcast(...)` method to initiate the operation, passing the new value and shareable variable. The receiving thread can wait to receive a new value using the `waitFor(...)` method. There is `broadcast(...)` method with the optional parameter *indices* to put broadcasted value into a specified multiple-level array index. The `asyncBroadcast(...)` method is also offered by PCJ.

Version 5.1 of the PCJ library introduces `reduce(ReduceOperation<T> function, Enum<?> name)` and `collect(Enum<?> name)` methods. Version 5.2 of the library introduces `scatter(...)` method. As of version 5.3, the signature of the method is as following: `scatter(Map<Integer, T> newValueMap, Enum<?> name)`. Moreover, version 5.3 of the library changed `collect(...)` method name to `gather(...)` to not confuse the Java programmers as there exists

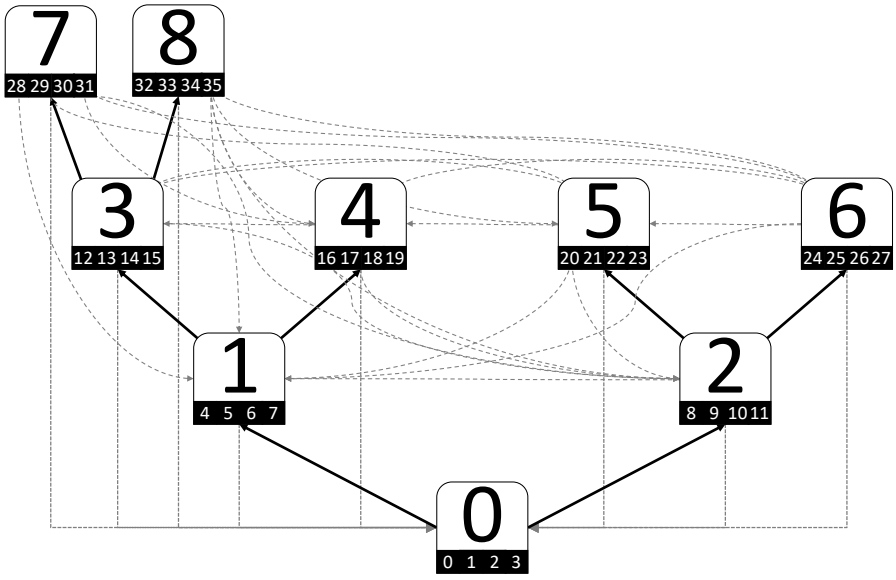

Figure 3: Communication scheme of nodes in PCJ. Node is represented by round single corner rectangles – large number represents node id, tiny numbers represent thread ids. Various types of arrows mean established TCP/IP connections during startup phase. Elbow arrow represents connection made to node-#0. Bold, straight arrow represents connection to child node that creates communication binary tree. Curved arrows are other connections between each pair of nodes.

`collect(...)` method in the *Java Streams* with a more powerful meaning. At the same time, it introduces `collect(SerializableSupplier<Collector<T, ?, R>> collectorSupplier, Enum<?> variable)` method with a meaning similar to the aforementioned method from the *Java Streams*.

The above-mentioned methods implement communication using a binary tree of the nodes presented in Figure 3. The first two methods collect data within a physical node before sending it to other nodes. This reduces the number of communication performed between nodes, i.e. between different JVMs. Similarly, when the `scatter(...)` method is used, there is only one message to each JVM reducing the number of communication performed. The `reduce(...)` method performs reduction during communication, while `gather(...)` only gathers data. The `scatter(...)` method treats map keys as thread ids and puts an associated value into adequate thread storage.

The `collect(...)` method performs more advanced and powerful reduction during communication, like `collect(Collector)` method of *Java Streams* introduced in Java 8. It can return a map of values like the `gather(...)` method and return a single reduced value like `reduce(...)` method, but the method can also do additional filtering, grouping and other processing during collecting data, and return other types of collections such as *set* or *list*.

Like other methods that use variables, these collective methods also have the optional parameter *indices* for accessing arrays element, and there exists an asynchronous version of these methods: `asyncReduce(...)`, `asyncGather(...)`, `asyncScatter(...)`, and `asyncCollect(...)`.

All the collective communication primitives use the communication tree. However, the exact view of the tree depends on the node of the PCJ thread that initializes the operation (*requester*). In version 5.1, we introduced a simpler communication schema, where the *requester* sends the request to node-#0 and the communication through tree occurs – from the root through intermediate

nodes to leaves and back from leaves to intermediate nodes to root. The result value or at least the notification about operation completion was collected in node-#0 and then transmitted to the requester. However, when multiple threads try to execute the collective operation, the performance dropped as all use the node-#0 that was becoming the bottleneck.

In version 5.2, we introduced simple *shifting* of the tree, so the *requester* node becomes the root of the tree, and thus communication performance is better. Figure 4 presents the original and *shifted* trees where the *requester* thread is placed on node-#5.

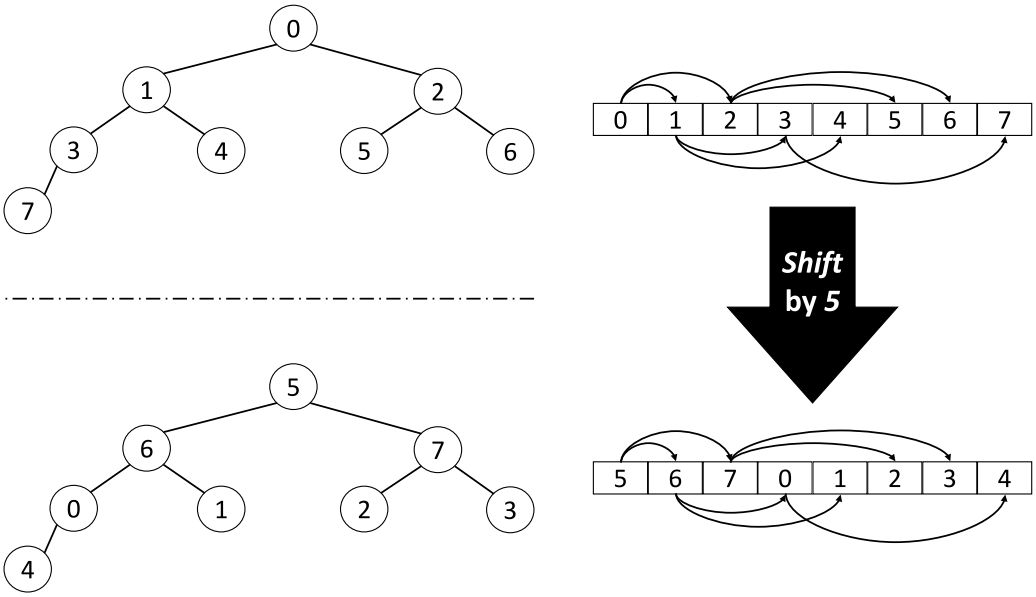

Figure 4: Communication scheme for collective operations using *shifted* tree.

5. Performance results

The performance results presented in the paper have been obtained using the Cray XC40 systems at ICM (Okeanos; University of Warsaw, Poland). Each computing node is equipped with two Intel Xeon E5-2690 v3 processors with hyperthreading available (2 threads per core) that give the possibility to execute 48 computing threads in parallel. Moreover, there is 128 GB of RAM for the computing node. The system utilizes Cray Aries interconnect in the butterfly topology.

For each benchmark workload, we have executed at least 5 times with a various number of threads and nodes configurations and take the result that gives the best performance. We assume that worse results are due to the operation of other users on the shared supercomputer as well as other nondeterministic actions, like an unnecessary run of the *Garbage Collector* (GC). The required GC are included in the results, as we assume that the best result contains the necessary GC run – we do not deduct the time of GC run from the total execution time.

5.1 HPC Workload: Fast Fourier Transform

Fast Fourier Transform (FFT) was chosen as an exemplary computational kernel for presenting the PCJ library's feasibility in the HPC workloads due to its versatile catalog of use cases.

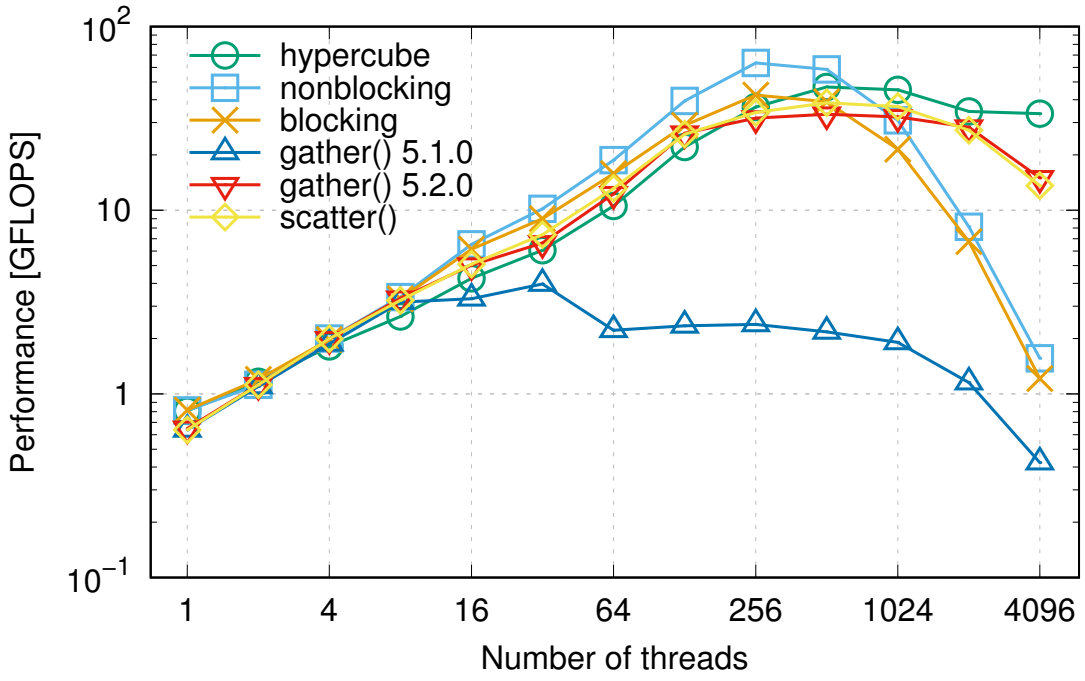

Figure 5: Performance of the 1D FFT implemented using different algorithms. The array of 2^{27} elements was used.

PCJ version is based on the PGAS implementation developed for Coarray Fortran 2.0 [24]. It is an example of a radix-2 binary exchange algorithm. As described in [24], it is based on the following schema: local FFT computation on the bit-reversing permutation of the input data, followed by communication-involving data transposition from block to the cyclic layout. This phase is followed by the subsequent local FFT calculation and concluded by the reverse transposition to the original layout (again, communication-involving). Communication between threads is centralized in the all-to-all routine.

In the case of our code, the three all-to-all exchanges are implemented as follows. Hypercube-based is described thoroughly in [25] and is presented in Listing 2. The communication is organized by a series of pair-wise data exchanges between the processes that are partners in a hypercube's given dimension in this case. In the case of the blocking communication, which implementation is based on [26], each thread loops through other threads' memory and gets the relevant data using blocking communication routines, starting from the thread with `id == PCJ.myId() + 1` to reduce network congestion [27]. Non-blocking communication mode enhances the blocking scheme with the use of non-blocking `asyncGet(...)` routine to initialize the communication.

Figure 5 shows the performance results, in terms of scalability, for complex one-dimensional FFT of 2^{27} elements. The three alternative self-implemented all-to-all reduction schemas are compared with the `gather(...)` and `scatter(...)` collective operations presented in Listing 3 and Listing 4, respectively. An improvement from version 5.1 to 5.2 can easily be spotted, making it almost on par with the fine-tuned hypercube version. The very poor performance of *gather* in version 5.1 is due to the usage of the same communication channel and bottleneck on node-#0 that has to send results to all of the requester threads. The performance of *gather* and *scatter* methods in version 5.2 is slightly lower than for fine-tuned hypercube version because the hypercube version

```
int logNumProcs = 0;
for (int j = PCJ.threadCount(); (j >>>= 1) > 0;) ++logNumProcs;
PCJ.putLocal(new double[logNumProcs][][],
             Shareable.blocksHypercube);

double[][] blocked = new double[PCJ.threadCount()][2 * blockSize];
for (int i = 0; i < blocked.length; i++)
  System.arraycopy(src, i * 2 * blockSize,
                   blocked[i], 0, 2 * blockSize);
PCJ.barrier();

double[][] toSend = new double[PCJ.threadCount() / 2][];
for (int dimension = 0; dimension < logNumProcs; dimension++) {
  int mask = 1 << dimension;
  int partner = PCJ.myId() ^ mask;
  for (int i = 0, j = 0; i < PCJ.threadCount(); i++) {
    if (partner < PCJ.myId()) {
      if ((i & mask) == 0) toSend[j++] = blocked[i];
    } else {
      if ((i & mask) != 0) toSend[j++] = blocked[i];
    }
  }
  PCJ.put(toSend, partner, Shareable.blocksHypercube, dimension);

  // receive
  double[][] checked;
  do {
    checked = PCJ.getLocal(Shareable.blocksHypercube, dimension);
    if (checked != null) {
      PCJ.putLocal(null, Shareable.blocksHypercube, dimension);
      for (int i = 0, j = 0; i < PCJ.threadCount(); i++) {
        if (partner < PCJ.myId()) {
          if ((i & mask) == 0) blocked[i] = checked[j++];
        } else {
          if ((i & mask) != 0) blocked[i] = checked[j++];
        }
      }
    }
  } while (checked == null);
}

for (int i = 0; i < blocked.length; i++) {
  System.arraycopy(blocked[i], 0,
                   dest, i * 2 * blockSize, 2 * blockSize);
}
```

Listing 2: All-to-all hypercube collection in FFT implementation.

```
double[][] blocks = new double[PCJ.threadCount()][2 * blockSize];
for (int i = 0; i < blocks.length; i++) {
  System.arraycopy(source, 2 * i * blockSize,
                   blocks[i], 0, 2 * blockSize);
}
PCJ.putLocal(blocks, Shareable.blocks);
PCJ.barrier();

Map<Integer, double[]> allRecv = PCJ.gather(Shareable.blocks,
                                            PCJ.myId());

for (Map.Entry<Integer, double[]> recv : allRecv.entrySet()) {
    System.arraycopy(recv.getValue(), 0,
                     dest, recv.getKey() * 2 * blockSize,
                     2 * blockSize);
}
```

Listing 3: All-to-all collection with `PCJ.gather()` in FFT implementation.

```
PCJ.putLocal(new double[PCJ.threadCount()][], Shareable.blocks);

                                ...

Map<Integer, double[]> toSendMap
    = IntStream.range(0, PCJ.threadCount()).boxed()
        .collect(Collectors.toMap(Function.identity(),
                i -> Arrays.copyOfRange(source,
                                        i * 2 * blockSize,
                                        (i + 1) * 2 * blockSize)));
PCJ.scatter(toSendMap, Shareable.blocks, PCJ.myId());

PCJ.waitFor(Shareable.blocks, PCJ.threadCount());
for (int i = 0; i < blocks.length; ++i) {
    System.arraycopy(blocks[i], 0,
                     dest, i * 2 * blockSize, 2 * blockSize);
}
```

Listing 4: All-to-all collection with `PCJ.scatter()` in FFT implementation.

sends data directly to receiver threads, whereas the *gather* and *scatter* involves the communication of all nodes on each method invocation. However, the code with *gather* or *scatter* method is clearer and easier to maintain.

5.2 Artificial Intelligence Workloads: Distributed Neural Network Training

Due to the advent of machine-learning-based methods and ongoing convergence of AI and HPC, the PCJ library's performance was also tested in the case of distributed neural network training, compared with state-of-the-art solution, Horovod [28]. For this test, we have trained a

simple network, based on [29], comprising of three fully connected layers of 300, 100, and 10 neurons for MNIST image classification. PCJ library was treated as a calculation instrumenter, responsible for their start, implementation of communication, while the calculations themselves were delegated to TensorFlow. This implementation manner is thus conceptually similar to that of Horovod. In the case of Horovod implementation, MNIST-classifying Python code was enhanced with calls and hooks for the calculation distribution. For the PCJ, hooks were added to the original Python-based computation graph creation code, which was then exploited by a Java-based runner for adjusting per-thread local neural network weights.

In the case of the PCJ, two algorithms were implemented. Synchronous one performs gradient averaging after each mini-batch, while asynchronous is based on the work by other authors [30].

The software stack used for this implementation consisted of TensorFlow 1.15 (due to the severe API changes we are working on upgrading to version 2.x), Horovod v. 0.16.1, Python 3.7.10. This version of the performance test uses PCJ 5.2.0 and trained the network for 30 epochs with 100 mini-batch sizes ([31] can be consulted for the results achieved for the older version of the software stack). Python/Horovod version is shown for either the default configuration of MKL library or with its parameters tuned for our computing system. Our asynchronous version (for this simple network) was the best performer.

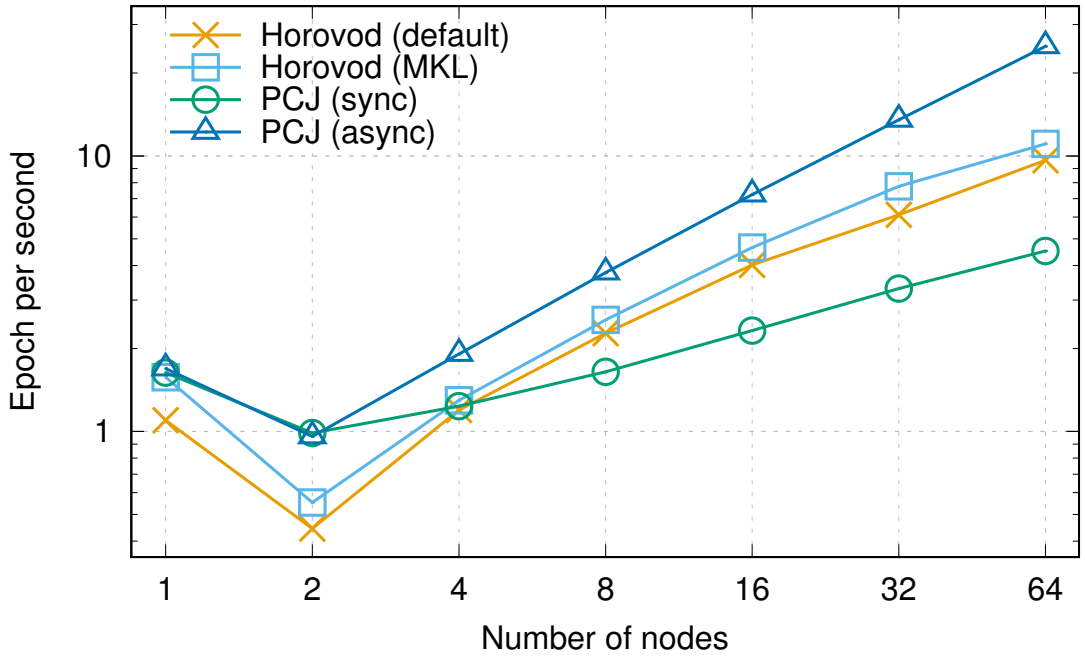

Figure 6: Comparison of the distributed training time taken by Horovod and PCJ. Accuracy of ≥ 90% was achieved.

5.3 Big Data Workload: WordCount

For the demonstration of Big Data workload, we typically use the *WordCount* example. Called a *Hello world of Big Data*, it is a conceptually simple code that reads many files and counts the histogram of occurrences of each word therein. It can be implemented in an embarrassingly parallel way, with each thread reading a subset of input files and the calculations finalized with the reduction.

```
@Storage(WordCount.class)
enum Shareable { localCounts }

private Map<String, Integer> localCounts;

private void countWords() throws IOException {
    Pattern WORD_BOUNDARY = Pattern.compile("\\s*\\b\\s*");
    // map phase
    localCounts = Files.readAllLines(Paths.get(myFileName),
                                     StandardCharsets.ISO_8859_1)
                       .stream()
                       .map(WORD_BOUNDARY::split)
                       .flatMap(Arrays::stream)
                       .filter(word -> !word.isEmpty())
                       .collect(Collectors.groupingBy(
                                     Function.identity(),
                                     HashMap::new,
                                     Collectors.counting()));
    PCJ.barrier();

    // reduce phase
    if (PCJ.myId() == 0) {
        Map<String, Long> mergedMap = PCJ.reduce( (mine, their)
                       -> Stream.concat(mine.entrySet().stream(),
                                        their.entrySet().stream())
                                .collect(Collectors.toMap(
                                        Map.Entry::getKey,
                                        Map.Entry::getValue,
                                        Long::sum)),
                       Shareable.localCounts);
        saveToFile("counts.txt", mergedMap);
    }
}
```

Listing 5: Implementation of the *WordCount* problem with usage of the PCJ library.

Listing 5 presents our implementation for the *WordCount* problem. The `barrier()` ensures that all threads read their own file and prepares a *map* with the occurrences. The `reduce(...)` method is used only by thread-#0. The reduce method uses a lambda expression to combine the partial results of all threads.

We have traditionally used a text of the real book for the word-counting code performance analyses (with other authors already following suit [11]). Georges de Scudéry's *Artamène ou le Grand Cyrus* [32] is one of the largest novels ever written, and its textual representation takes 10 MB of disk space. We have performed weak scalability measurement; each thread reads the whole file, so with the increasing number of threads, the total size of processed data also increases.

The results presented in Figure 7 show good scalability of the PCJ implementation when the

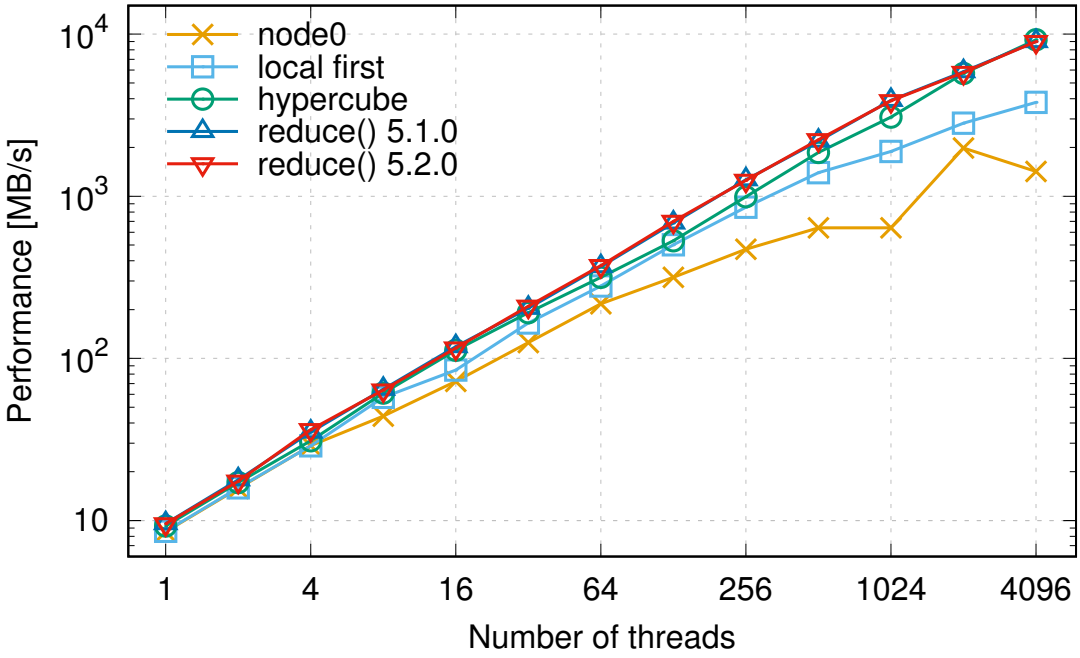

Figure 7: Weak scalability of *WordCount* application implemented with PCJ. The scaling results are presented for 10 MB file.

proper communication schema is used. The performance decreases with the number of threads due to the increasing size of data to communicate. However, the scalability is almost linear for *hypercube* and `reduce(...)` implementations reflecting the good implementation of the reduction part of the algorithm. There is no performance degradation for version 5.1 compared to version 5.2 of the PCJ library, as the reduce operation is done only once by the thread-#0. The two other implementations present worse performance. Both implementations assume that the same number of threads is running on each node. In the *node0* implementation, all threads on node-#0 collect the value from other nodes, then local reduction occurs. In the *local first* implementation, local threads on each node reduce the value to the thread with the lowest *id* inside a node, then thread-#0 collect reduced data from selected threads of all nodes.

6. Conclusions

The PGAS programming model realized by the PCJ library allows for convenient implementation of various parallel schemas, including HPC, AI, and Big Data workloads. The results presented in this paper and previous publications show the possibility of using the Java language to implement parallel applications for multinode systems. Collective operations presented in this paper, introduced in PCJ version 5.1 and improved in versions 5.2 and 5.3, reduce programming effort and allow for even easier development of scalable code.

The PCJ library offers a flexible and easy-to-use computation distribution framework. It can be a base for the implementation of sophisticated and easy-to-maintain systems. In the near future, it will be used to parallelize computations within the Human Exposome Assessment Platform (HEAP).

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 grant GA69-19.

References

[1] M. Nowicki and P. Bała, *Parallel computations in Java with PCJ library*, in *2012 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 381–387, IEEE, 2012.

[2] K. Feind, *Shared memory access (SHMEM) routines*, *Cray Research* (1995) .

[3] J. Nieplocha, R.J. Harrison and R.J. Littlefield, *Global arrays: A nonuniform memory access programming model for high-performance computers*, *The Journal of Supercomputing* **10** (1996) 169.

[4] W.W. Carlson, J.M. Draper, D.E. Culler, K. Yelick, E. Brooks and K. Warren, *Introduction to UPC and language specification*, Tech. Rep. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences (1999).

[5] J. Reid, *The new features of Fortran 2008*, in *ACM SIGPLAN Fortran Forum*, vol. 27, pp. 8–21, ACM, 2008.

[6] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy et al., *Titanium: a high-performance Java dialect*, *Concurrency and Computation: Practice and Experience* **10** (1998) 825.

[7] J. Dean and S. Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, *Communications of the ACM* **51** (2008) 107.

[8] "Apache Hadoop homepage." https://hadoop.apache.org.

[9] D. Borthakur, *The Hadoop Distributed File System: Architecture and Design*, *Hadoop Project Website* **11** (2007) 21.

[10] M. Nowicki, Ł. Górski and P. Bała, *PCJ–Java Library for Highly Scalable HPC and Big Data Processing*, in *2018 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 12–20, IEEE, 2018.

[11] J. Posner, L. Reitz and C. Fohry, *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.

[12] M. Nowicki, M. Ryczkowska, Ł. Górski and P. Bala, *Big Data Analytics in Java with PCJ Library: Performance Comparison with Hadoop*, in *International Conference on Parallel Processing and Applied Mathematics*, pp. 318–327, Springer, 2017.

[13] "Akka - build concurrent, distributed, and resilient message-driven applications for Java and Scala." https://akka.io/.

[14] "PCJ homepage." https://pcj.icm.edu.pl.

[15] "PCJ at GitHub." https://github.com/hpdcj/PCJ.

[16] "Apache Maven Project homepage." https://maven.apache.org.

[17] F.P. Miller, A.F. Vandome and J. McBrewster, *Apache Maven*, Alpha Press (2010).

[18] "Gradle Build Tool homepage." https://gradle.org.

[19] A.L. Davis, *Gradle*, in *Learning Groovy 3*, pp. 105–114, Springer (2019).

[20] M. Nowicki, *Comparison of sort algorithms in Hadoop and PCJ*, *Journal of Big Data* **7** (2020) .

[21] M. Nowicki, Ł. Górski and P. Bała, *Performance Evaluation of Java/PCJ Implementation of Parallel Algorithms on the Cloud*, in *Euro-Par 2020: Parallel Processing Workshops*, vol. 12480, p. 213, Nature Publishing Group.

[22] M. Nowicki, *Running Java/PCJ application using YARN on Ethernet Cluster*, Tech. Rep. HPI Technical Report (in print).

[23] M. Nowicki, M. Ryczkowska, Ł. Górski, M. Szynkiewicz and P. Bała, *PCJ-a Java library for heterogenous parallel computing*, *Recent Advances in Information Science (Recent Advances in Computer Engineering Series vol 36), WSEAS Press* (2016) 66.

[24] J. Mellor-Crummey, L. Adhianto, G. Jin, M. Krentel, K. Murthy, W. Scherer et al., "Class II submission to the HPC Challenge award competition Coarray Fortran 2.0."

[25] S.J. Plimpton, R. Brightwell, C. Vaughan, K. Underwood and M. Davis, *A simple synchronous distributed-memory algorithm for the HPCC RandomAccess benchmark*, in *2006 IEEE International Conference on Cluster Computing*, pp. 1–7, IEEE, 2006.

[26] P. Manninen and H. Richardson, *First Experiences on Collective Operations with Fortran Coarrays on the Cray XC30*, in *7th International Conference on PGAS Programming Models*, vol. 222, 2013.

[27] M. Nowicki, Ł. Górski and P. Bała, *Performance evaluation of parallel computing and Big Data processing with Java and PCJ library*, *Cray Users Group* (2018) .

[28] A. Sergeev and M.D. Balso, *Horovod: fast and easy distributed deep learning in TensorFlow*, *arXiv preprint arXiv:1802.05799* (2018) .

[29] A. Géron, *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*, " O'Reilly Media, Inc." (2017).

[30] J. Keuper and F.-J. Pfreundt, *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*, p. 1, ACM, 2015.

[31] M. Nowicki, Ł. Górski and P. Bała, *PCJ Java library as a solution to integrate HPC, Big Data and Artificial Intelligence workloads*, *Journal of Big Data* **8** (2021) .

[32] M.d. Scudéry, *Artamène ou le grand Cyrus* (1972).