Performance evaluation of parallel computing and Big Data processing with Java and PCJ library

Marek Nowicki
Faculty of Mathematics and Computer Science
Nicolaus Copernicus University
Toruń, Poland
faramir@mat.umk.pl

Łukasz Górski, Piotr Bała
Interdisciplinary Centre for Mathematical
and Computational Modeling
University of Warsaw
Warsaw, Poland
lgorski, bala@icm.edu.pl

***Abstract*—In this paper, we present PCJ (Parallel Computing in Java), a novel tool for scalable high-performance computing and big data processing in Java. PCJ is Java library implementing PGAS (Partitioned Global Address Space) programming paradigm. It allows for the easy and feasible development of computational applications as well as Big Data processing. The use of Java brings HPC and Big Data type of processing together and enables running on the different types of hardware. In particular, the high scalability and good performance of PCJ applications have been demonstrated using Cray XC40 systems. We present performance and scalability of PCJ library measured on Cray XC40 systems with standard benchmarks such as ping-pong, broadcast, and random access. We describe parallelization of example applications of different characteristics including FFT and 2D stencil. Results for standard Big Data benchmarks such as word count are presented. In all cases, measured performance and scalability confirm that PCJ is a good tool to develop parallel applications of different type.**

***Keywords*-PCJ, Java, Big Data, parallel computing, performance**

I. Introduction

In the recent years, we observe a number of exascale initiatives which focus on the big challenges of exascale for hardware and software architecture. We observe the emergence of the phenomena of Big Data in a wide variety of scientific fields. Big Data processing represents a shift that is transforming the entire research landscape on which plans for exascale computing must play out.

The problem is that the two paradigms split both in the hardware and software used. The traditional packages used for computing do not address Big Data type of processing. Moreover, they are based on the computer languages such as FORTRAN or C/C++ which are not that popular in data analytics which nowadays relies on Java, Scala, Python and other script languages. On the other hand, installation of the data analytics framework on the HPC systems is not easy. It is especially difficult to achieve good performance and scalability. Integration with the queueing systems is also a challenge. Therefore there is a need for new standards that will govern the interoperability between data and compute [1].

To address this problem we have developed PCJ [2], [3] which is a library for scalable parallel computing in Java. The PCJ brings high-performance computing in Partitioned Global Address Space (PGAS) paradigm to Java programmers and the same time allows for efficient data processing. The applications developed with the PCJ library can be run on the traditional HPC systems as well as on Big Data infrastructures such as Hadoop/Spark. PCJ library provides a user with the easy to use and flexible programming tools which allow implementing different parallelization schemas. With just few programming constructs programmer can implement the map-reduce algorithm as well as any other computational or data-intensive algorithm. The PCJ application can be run on wide range of the systems including laptops and workstations as well as top supercomputers.

The paper is organized as follows: section II describes PCJ library and its main functionality. In the section III we present performance results for the selected HPC benchmarks run on the Cray XC40 confirming good scalability and performance of the PCJ library. The scalability and numerical efficiency of the selected computational kernels such as FFT or 2D stencil are presented in the section IV. The performance results of the selected Big Data benchmarks running on the Cray XC40 systems are presented in the section V. The conclusions and future work are described in the sections VI and VII.

The performance experiments have been performed on Cray XC40 systems at ICM University of Warsaw, Poland (Okeanos, 1024 nodes) and at HLRS, University of Stuttgart, Germany (HazelHen, 7742 nodes). The PCJ version 5.0.6 with Oracle Java 1.8.0_51 were used. The C/MPI code was compiled using GNU compuler 5.1 and Cray-mpich 7.6.

II. PCJ library

PCJ is an OpenSource Java library developed under BSD license. 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. A programmer is provided with the PCJ class with a set of methods to implement

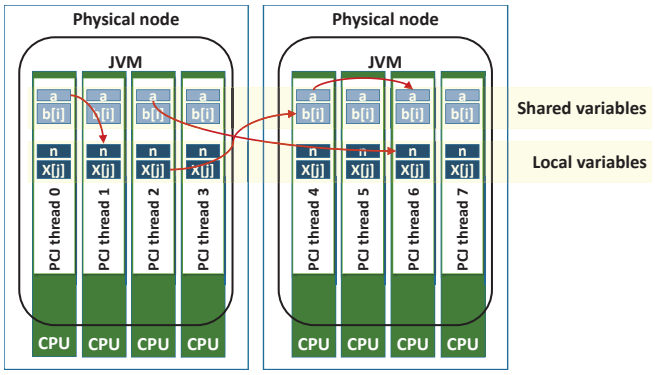

Figure 1. Diagram of PCJ computing model (from [4] p. 67). Arrows present possible communication using `put()` or `get()` methods acting on shared variables.

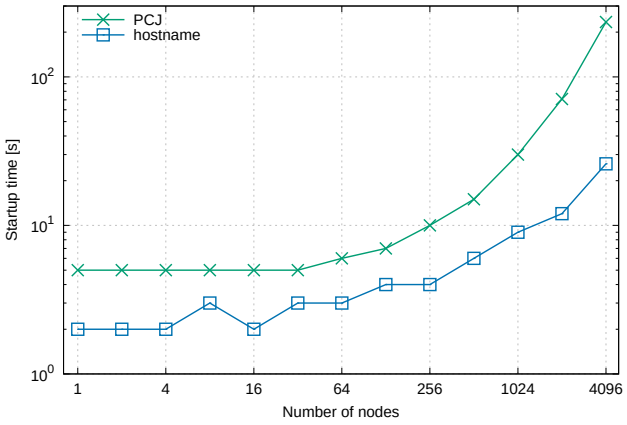

Figure 2. The PCJ startup time for different number of nodes. The execution time of `hostname` command is plotted for reference. Data collected at Cray XC40 at HLRS.

necessary parallel constructs. All communication details like threads administration or network programming are hidden from the programmer. Instead of modifying problem to fit given programming model such as map-reduce the user implements his algorithm in an optimal manner. In particular, a programmer can easily implement data and work partitioning best suited to the problem he is solving. PCJ library provides necessary tools for it including threads numbering, data transfer and threads synchronization. The communication is one-sided and asynchronous which makes programming easy and less error-prone. Figure 1 shows the diagram of PCJ computing model together with possible exchange of data.

The PCJ applications can run on the PC's, x86 clusters and supercomputers including Cray XC40 systems. PCJ has been tested on Intel KNL [5] processors as well as Power8 systems. The applications implemented with PCJ and Java scale up to hundreds of thousands of cores. PCJ is well integrated with the whole range of the system management tools including most popular batch systems and execution environments. Thanks to the Java portability, the PCJ application developed and tested on the laptop or workstation can be moved to supercomputer without any modifications or even without recompilation.

PCJ has been already used for parallelization of selected applications. A good example is communication intensive graph search from Graph 500 test suite. The PCJ implementation scales well and outperforms Hadoop implementation by the factor of 100 [6], [7]. 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 [8] and to find parameters of the neural network simulating connectome of *C. Elegans* [9], [10]. Recent examples include parallelization of the sequence alignment [11], [12]. PCJ library allowed for the easy implementation of the dynamic load balancing for multiple NCBI-BLAST instances spanned over multiple nodes. The obtained performance was at least 2 times higher than for the implementations based on the static work distribution. The PCJ based implementation scales well up to the hundreds of nodes and allows for significant reduction of the analysis time.

A. *Multinode execution*

The application using PCJ library is run as typical Java application using Java Virtual Machine (JVM). In the multinode environment, one (or more) JVM has to be started on each node. PCJ library takes care of this process and allows a user to start execution on multiple nodes, running multiple threads on each node. The number of nodes and threads can be easily configured, however, the most reasonable choice is to limit on each node number of threads to the number of available cores. Typically, single Java Virtual machine is run on each physical node although PCJ allows for multiple JVM scenario.

The startup time for typical PCJ application (see Fig. 2) is similar to the startup time of any parallel application launched using system tools such as `srun` or `aprun`. The startup time increases with the number of nodes used. The scalability is similar to one obtained for running simple commands such as `hostname`.

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

The PCJ library handles both situations hiding details from the user. It distinguishes between inter- and intranode communication and picks up proper data exchange mech-

anism. Moreover, nodes are organized in the graph which allows optimizing global communication.

B. PCJ methods

Each PCJ task (PCJ thread) executes its own set of instructions. Variables and instructions are private to the task, however, some variables can be communicated (shared) between tasks. Every shared variable is declared as a field of a class. Listing 1 can be referred for a sample declaration. Exposition of local fields for remote addressing is performed with the use of annotations supplied with the PCJ library. Firstly, the use of `@Storage` annotation coupled with a custom enumeration allows PCJ communication methods to reference a shared variable by name in a safe manner. Therefore all enum's constants point to the fields that a given class shares with other threads of execution. Type of the shared variable is inferred from the type of the associated field. Secondly, every enumeration and a storage associated with it has to be registered by the thread that owns a given shared variable. Other threads can access the shared variable by providing relevant registered enum's constant name, but do not have to register the enum itself. Storages can be automatically registered on application start phase if the `@RegisterStorage` annotation is used on actual `StartPoint` class. While using the `PCJ.registerStorage(...)` method during runtime allows for a greater flexibility, in practice library's users tend to use annotation-based registration.

```
1 @RegisterStorage(Example.Shared.class)
2 public class Example implements StartPoint {
3     @Storage(Example.class)
4     enum Shared { a }
5     public double a;
6                         ...
```

Listing 1. Shared variable declaration.

The basic primitives of PGAS programming paradigm offered by the PCJ library are listed below. They may be executed by all the threads of execution or only over a subset forming a group:

- `get(int threadId, Enum<?> name)` – allows to read a shared variable (tagged by name) published by another thread identified with *threadId*; this method is blocking – there exists a non-blocking method `asyncGet(...)` with the same parameters;
- `put(T newValue, int threadId, Enum<?> name)` – dual to `get()`, writes to a shared variable (tagged by name) owned by a thread identified with *threadId*; the operation is blocking – does not return till the update acknowledge is received; there is also non-blocking method `asyncPut(...)` with the same parameters which can be used for busy waiting for acknowledgment;
- `barrier()` – blocks the threads until all come to the synchronization point in the program; a two-point version of barrier that synchronizes only the selected two threads is also supported; moreover there exists non-blocking barrier method called `asyncBarrier()` that allows for proceed with execution and for checking if all threads pass the synchronization point;
- `broadcast(T newValue, Enum<?> name)` – broadcasts the *newValue* and writes it to each thread's shared variable tagged by name; the broadcast is blocking – it blocks the invoking thread until all threads receive new value; there also exists nonblocking `asyncBroadcast(...)` method with the same parameters;
- `waitFor(Enum<?> name)` – due to the asynchronicity of communication, there is a method that allows to block thread until a change of its shared variable (tagged with a name) is done;
- `monitor(Enum<?> name)` – resets the modification count of a shared variable (tagged by name) used by `waitFor(...)` method.

All non-blocking, asynchronous methods return `PcjFuture<?>` object that can be used for checking for the completion of the operation, or waiting for completion with a defined timeout or even without a timeout. The blocking methods are just wrappers for non-blocking methods that wait for completion without timeout (e.g. call to `get(threadId, name)` is equivalent to `asyncGet(threadId, name).get()`).

III. PCJ BENCHMARKS

In order to determine basics characteristics of communication within PCJ library, we have performed ping-pong and broadcast microbenchmarks. Those tests measure the performance of simplest forms of interprocess communication. In addition, we present performance of the random access test which combines communication with the memory access.

A. Ping-pong

We have measured the bandwidth for sending an array of double elements ranged from 1 element (8 bytes) up to 4,194,304 elements (32 MB). We have done 5 tests, each sending 100 times the array and calculating average time necessary to finish the sending loop. The best time (the lowest value) is taken as a result. The JVM was warmed up so first runs were not accounted. Listing 2 shows code fragments of the ping-pong benchmark implementation in PCJ using `PCJ.get()` method.

The results obtained for the ping-pong between threads running on the same node are presented in Fig. 3. For the PCJ the best results are obtained for the non-blocking method `PCJ.put()` (see Listing 3). In this case, the next block of data is sent just after previous data was transferred to sending buffer. The confirmation of correctly received message is not used here.

Similar results are obtained for the ping-pong benchmark running on two nodes. As presented in Fig. 4 for the PCJ the nonblocking communication is the fastest and reaches

```
@Storage(PingPong.class)
public enum Shared { a }
double[] a;
final int ntimes = 100;
final int number_of_tests = 5;
                          ...
int n = numberOfElementsToTransmit;
                          ...
PCJ.barrier();
for (int k = 0; k < number_of_tests; k++) {
  long time = System.nanoTime();
  for (int i = 0; i < ntimes; i++)
    if (PCJ.myId() == 0)
      b = PCJ.<double[]>get(1, Shared.a);
  time = System.nanoTime() - time;
  PCJ.barrier();
}
```

Listing 2. Code fragments of ping-pong benchmark. The implementation based on `PCJ.get()` method.

```
PCJ.monitor(Shared.a);
PCJ.barrier();
                          ...
for (int k = 0; k < number_of_tests; k++) {
  long time = System.nanoTime();
  for (int i = 0; i < ntimes; i++)
    if (PCJ.myId() == 0)
      PCJ.asyncPut(b, 1, Shared.a);
    else PCJ.waitFor(Shared.a);
  time = System.nanoTime() - time;
  PCJ.barrier();
}
```

Listing 3. Code for ping-pong benchmark using `PCJ.asyncPut(...)` method.

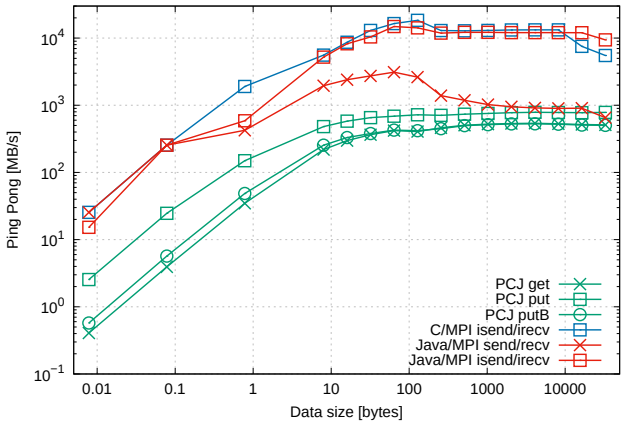

Figure 3. Performance of the ping-pong communication between two threads running on the same node of Cray XC40. Performance as function of the data size is presented for implementations using different PCJ methods and for C/MPI and for Java/MPI.

80 MB/s for the largest datasets. This is a physical limit of the IP over InfiniBand communication used by PCJ library.

For both cases, one and two node communication, PCJ shows similar performance. The measured bandwidth, compared to C/MPI, is couple times lower for small data sizes and several times lower for large data sets. This is due to

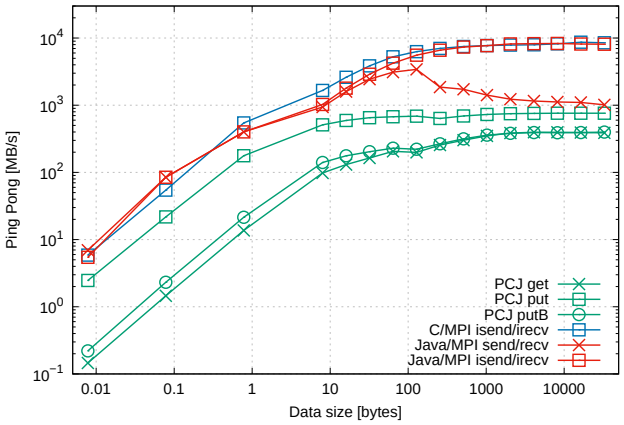

Figure 4. Performance of the ping-pong communication between two threads running on the different nodes of Cray XC40. Performance as function of the data size is presented for implementations using different PCJ methods and for C/MPI and for Java/MPI.

the fact, that the messages in the PCJ have to be serialized to have a complete copy (clone) of the data, even when the sender and receiver are on the same node. Another important reason is a type of communication channel used. PCJ realizes all communication between nodes using NIO methods utilizing TCP/IP sockets while MPI takes advantage of the Cray Aries interconnect. The performance of Java/MPI (Java MPI bindings) [13] is similar to C/MPI, however, blocking communication for large data sizes is slower than for non-blocking one. It is because in nonblocking communication *direct buffer* contains the data to be transferred and there is no need to create an additional copy of the data. The data is placed in the *direct buffer* while receiving and to be able to use it, it is necessary to convert it back to proper type (e.g. `double[]`). In the blocking communication, it is possible to directly use an array of primitive type, and the array's binary data is copied to the proper buffers for underlying MPI function in the JNI call.

B. Broadcast

Broadcast benchmark measures time needed for sending value from one selected thread (thread 0) to all threads. Like in the ping-pong, the message size is calculated as a length in bytes of the data without adding a size of any header. Similarly, the data sent in this benchmark is an array of double elements ranged from 1 element (8 bytes) to 4,194,304 elements (32 MB). The benchmark method was invoked 100 times and the average time necessary to finish broadcasting (with notification) was calculated as a result. Listing 4 shows code fragments of the PCJ implementation.

The results presented in Fig. 5 show time needed for transferring data of various sizes to a number of threads ranged from 48 to 98,304 (1 to 2048 nodes of XC40).

The time spent for the broadcast operation using PCJ on average is significantly longer than for C/MPI for both small

```
@Storage(Broadcast.class)
public enum SharedEnum { a }
double[] a;
final int ntimes = 100;
final int number_of_tests = 5;
                          ...
int n = numberOfElementsToTransmit;
                          ...
PCJ.barrier();
for (int k = 0; k < number_of_tests; k++) {
  long time = System.nanoTime();
  for (int i = 0; i < ntimes; i++)
    if (PCJ.myId() == 0)
      PCJ.broadcast(b, SharedEnum.a);
  time = System.nanoTime() - time;
  PCJ.barrier();
}
```

Listing 4. Code for broadcast benchmark. The code for filling table with data is omitted.

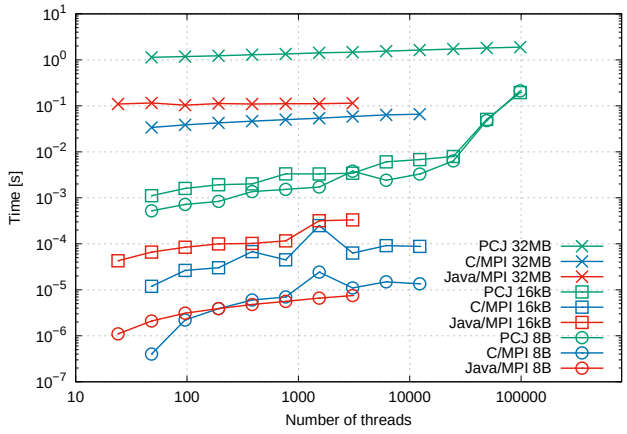

Figure 5. Performance (execution time) of the `PCJ.broadcast()` for different size of data. The data is presented for 32MB, 16kB and 8B. The MPI performance for the same data sizes is plotted for reference.

and large arrays. The transfer is limited by the performance of the TCP/IP communication used by the PCJ library. For the largest data size, PCJ library achieves an aggregated speed of 27 MB/s for 1 node and 17 MB/s for 2048 nodes which is significantly (more than 10 times) less than for MPI.

C. Random Access

Random Access is a benchmark that identifies the program's ability to perform random updates on an array that is distributed among all program's threads of execution. There is a little relationship between successive memory updates. Each update is an operation that involves reading random memory location, performing a modification of the value stored therein (usually the *xor* operation), and subsequent write of new value to that location – thus introducing a potential race condition in case of at least two threads modifying the same location at the same time. This is accounted for in the benchmark description, as at most 1% of random memory updates can be lost due to data races [14]. Yet, most of the known contemporary implementation avoid any data races and PCJ follows suite. Additionally, as far as the benchmark's requirements go, its underlying assumptions disapprove of using large internal look-ahead buffers of random numbers, limiting them to 1024 elements. That means that at most 1024 numbers (that identify memory locations) can be cached locally before performing distributed memory updates. This precludes from implementing an embarrassingly parallel version of the benchmark, in which every thread of execution locally generates and inspects the same stream of random values and updates only those distributed memory regions that are affiliated with that thread. In PCJ implementation caching limit enforces the performance of total inter-thread data exchange every time a series of 1024 random numbers is generated. Thus the performance of all-to-all communication scheme lays at the crux PCJ performance in case of random access test.

Typical runs of random access tests for HPC system performance assessment use about 50% of available system memory. In case of this work, strong scalability was tested, therefore all benchmarked runs used the same-sized global distributed array of 2^{26} elements, irrespective of a number of threads used.

Three all-to-all communication scenarios have been tested in the case of PCJ – namely (i) hypercube-based and those that utilize (ii) blocking and (iii) non-blocking all-to-all communication. Each scenario was executed every time a local look-ahead limit of 1024 generated remote locations was hit. When browsing the stream of generated numbers, each thread of execution fills its shared `updatesShared` variable (cf. Listing 5 for its initialization procedure) so that its i-th cell holds a stream of random locations that are affiliated with the fragment of shared array held by i-th thread. Upon the completion of this local phase, a communication phase is initiated.

In case of blocking communication, a simple algorithm based on prior CAF implementation was used, as suggested for Cray XC30 in [15]. It uses a simple loop, iterating from `myId()+1` to `numThreads()-1` and then from 0 to `myId()-1` to avoid network congestion. On each iteration, a value is received from target thread in blocking manner (see Listing 6). In case of non-blocking implementation (Listing 7) `PcjFuture`s are used for the communication initialization and the list of futures is constantly traversed in search of futures which have already finished communicating and made remote data available.

In hypercube-based communication algorithm data is transferred in $log_2 PCJ.threadCount()$ phases. This scheme is generally implemented for thread numbers that are a power of two and in each of d phases of communication ($0 \leq d < log_2 PCJ.threadCount()$) each thread communicates with its neighbor, i.e. a thread whose identification

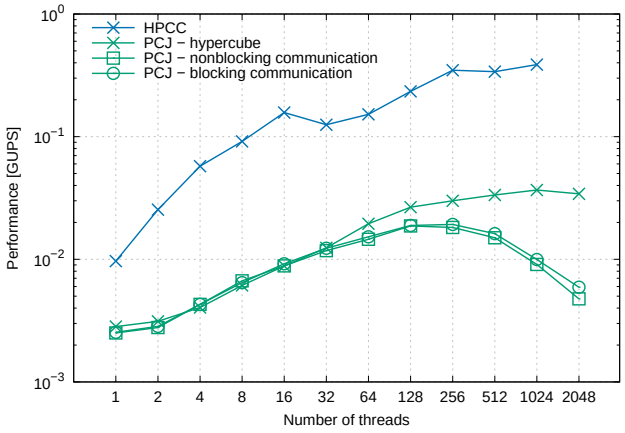

Figure 6. Performance of the *Random Access* benchmark implemented with PCJ using different algorithms. The data for the HPCC implementation using C/MPI is presented for reference.

```
@Storage(RandomAccessNew.class)
enum Shared { updatesShared }
List<Long> updatesShared[];
                          ...
//prepare properly-sized arrays for communication
List<Long>[] updatesToAdd
               = new ArrayList[PCJ.threadCount()];
for (int i = 0; i < updatesToAdd.length; i++)
  updatesToAdd[i] = new ArrayList<>();
//initialize shared variable
PCJ.putLocal(updatesToAdd, Shared.updatesShared);
```

Listing 5. Shared variable used for inter-thread communication in blocking and non-blocking implementation of all-to-all communication primitive.

number differs only on a d-th bit [16]. For reference, code that uses hypercube-based communication is presented in Listing 11 for reduction collective and all-to-all collective implementation is analogous. Presented codes exhibit that standard Java collection classes can be used in PGAS environment unproblematically.

Additionally, a facultative time-limit for the benchmark was implemented mimicking reference MPI implementation. It is based on *poison pill* pattern – when a thread 0 approaches time limit, a special stop-signaling value is transmitted between all the threads during communication phase ensuring a deadlock-free graceful program shutdown.

As expected, the best performing PCJ implementation is based on the hypercube communication scheme as it is presented in the Fig. 6. The scalability of PCJ implementations exhibit behavior that is similar to that for MPI, however, in the whole range the performance is 5–10 times lower. The performance of PCJ library in case of a communication scenario that involves a frequent exchange of relatively small messages is still a subject of our scrutiny.

```
private void alltoallBlocking() {
  PCJ.barrier();
  for (int num = 0,
     image = (PCJ.myId() + 1) % PCJ.threadCount();
     num != PCJ.threadCount() - 1;
     image = (image + 1) % PCJ.threadCount()) {
    List<Long> recv = (List<Long>) PCJ.get(
         image, Shared.updatesShared, PCJ.myId());
    performUpdates (recv);
    num++;
  }
  PCJ.barrier();
}
```

Listing 6. Simple blocking implementation of total all-to-all exchange with the use of PCJ library. Data intended for exchange is held in `updatesShared` variable.

```
private void allToAllNonBlocking() {
  PCJ.barrier();
  //prepare futures array
  PcjFuture<List<Long>>[] futures
               = new PcjFuture[PCJ.threadCount()];
  //get the data
  for (int num = 0,
     image = (PCJ.myId() + 1) % PCJ.threadCount();
     num != PCJ.threadCount() - 1;
     image = (image + 1) % PCJ.threadCount()) {
    if (image != PCJ.myId())
      futures[image] = PCJ.asyncGet(
        image, Shared.updatesShared, PCJ.myId());
    num++;
  }

  //receive the data
  int numReceived = 0;
  while (numReceived != PCJ.threadCount() - 1) {
    for (int i = 0; i < futures.length; i++) {
      if (futures[i] != null
            && futures[i].isDone()) {
        List<Long> recv = futures[i].get();
        processUpdates(recv);
        numReceived++;
        futures[i] = null;
      }
    }
  }
}
```

Listing 7. Non-blocking implementation of total all-to-all exchange with the use of PCJ library. Data intended for exchange is held in `updatesShared` variable.

IV. Applications

A. FFT

Fast Fourier Transform available as a reference MPI implementation in HPC Challenge Benchmark is based on the algorithm published by Takahashi and Kanada [17]. In the case of PCJ, as a starting point, we have chosen PGAS implementation developed for Coarray Fortran 2.0 [18]. The original Fortran algorithm uses a radix 2 binary exchange algorithm that aims to reduce interprocess communication: firstly, a local FFT calculation is performed based on the bit-reversing permutation of input data; after this step all threads

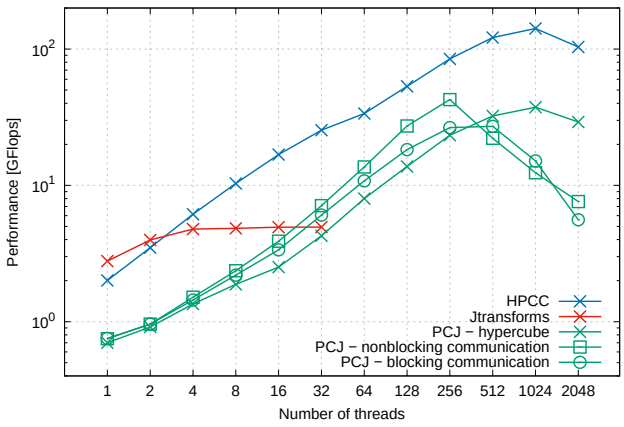

Figure 7. Performance of the FFT implemented using different algorithms. The data for the HPCC implementation using C/MPI is presented for reference.

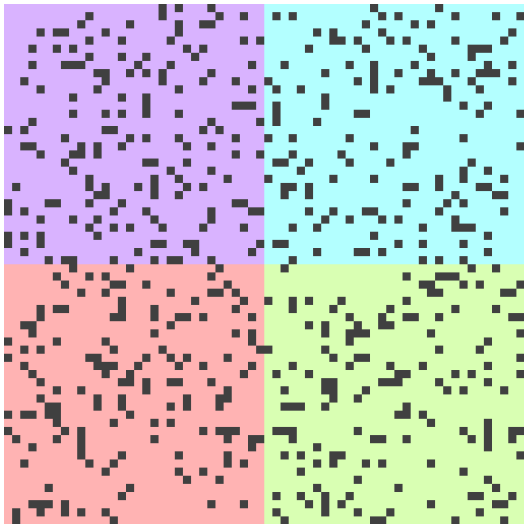

Figure 8. Representation of *Game of Life* 64×64 cells universe divided for processing by four PCJ threads represented by different color squares. Small black squares represent alive cells.

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 [18]. Similarly to Random Access implementation, interthread 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 to limit the communication, yet makes the implementation of all-to-all exchange once again central to the overall programme's performance.

In the case of FFT, the three all-to-all exchanges are implemented in the same fashion as in Random Access test (see Sec. III-C). The performance results for complex one-dimensional FFT of 2^{26} elements (Fig. 7) show how the three alternative all-to-all implementations compare in terms of scalability. It should be noted, that those performance results were to an extent limited by the queueing system's policy. At most 256 computing nodes were available for single-run performance testing and PCJ library reaches the best performance when each node is affiliated with single PCJ thread. While nonblocking communication allowed to achieve the best peak performance, the hypercube-based solution allowed to exploit the available computational resources to the greatest extent, reaching peak performance for 1024 threads when compared to 256 threads in case of nonblocking communication. In the case of latter, the performance decreases due to the available resources exhaustion can be clearly seen from the plot. For reference, Jtransforms [19] performance is plotted. It is a Java implementation of FFT that is universally regarded as offering the best performance, yet its scalability is inherently limited to 4 threads of execution. Thus its performance is surpassed by PCJ implementations when more computational resources are available.

B. *Game of Life*

The *Game of Life* is a cellular automaton devised by the John Conway [20]. In the original problem, there is two dimensional possibly infinitive board called universe with evenly placed cells. Each cell can be either in the *alive* or *dead* state. The universe, so the state of the cells, changes with time that elapses discretely. The state of a cell in the next step depends on the number of its alive neighbors in the current moment. In the original rules, the cell is born, so changes the state from the *dead* to *alive*, when it has exactly three alive neighbors. Any *alive* cell stays alive if has two or three alive neighbors, if the number of alive neighbors is lower or higher, the cell dies from under- or overpopulation. The *Game of Life* can be seen as a typical stencil algorithm with a 9-point 2D stencil – the 2D Moore neighborhood.

We modified the problem to have finite universe – the board has its maximum width and height. Each thread owns subboard – a part of the board divided in a uniform way using block distribution (see Fig. 8). Although there are known fast algorithms and optimizations that can save computational time generating next universe state, like Hashlife or memoization 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 – a part of `int` array for C and by using `BitSet` class in Java – where 0 and 1 means that cell is dead and alive respectively.

After generating the new universe state, the border cells of subboards are exchanged between proper threads. This is done asynchronously: the state of cells is sent to neighbors, and then the received state is incorporated into the board for the next step. The code fragment of exchanging data is

```
// sending West column to neighbour' East
if (!isFirstColumn) {
  for (int row=1; row<=rowsPerThread; ++row)
    sendShared.W[row - 1] = board.get(1, row);
  PCJ.asyncPut(sendShared.W,
        PCJ.myId() - 1, Shared.E);
}
// sending North-West cell to neighbour' South-
    East
if (!isFirstColumn && !isFirstRow) {
  sendShared.NW = board.get(1, 1);
  PCJ.asyncPut(sendShared.NW,
        PCJ.myId() - threadsPerRow-1, Shared.SE);
}
                        ...
// receiving East side
if (!isLastColumn) {
  PCJ.waitFor(Shared.E);
  for (int row=1; row<=rowsPerThread; ++row)
    board.set(colsPerThread + 1, row,
            recvShared.E[row - 1]);
}
// receiving South-East cell
if (!isLastColumn && !isLastRow) {
  PCJ.waitFor(Shared.SE);
  board.set(colsPerThread + 1, rowsPerThread + 1,
            recvShared.SE);
}
```

Listing 8. Code fragments of sending and receiving column and corner cell states. The code is presented only for selected columns and corner cells. The exchange of the other data is performed in analogous way.

available on Listing 8. 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 universe is not a 2D torus, but a grid. The state of neighbor cells that would be behind universe edge is treated as *dead*.

1) Performance results: We have measured the performance in a total number of cells processed in the unit of time ($cells/s$). For the $N \times N$ cells universe, we measured time using high-resolution time source (in nanoseconds) needed for calculating cells next state and exchanging state of cells on the border of subboards, and then performance was calculated as $\frac{N \times N}{time}$. For each test, we performed eleven time steps. We warmed up the Java Virtual Machine to allow the JVM to use just-in-time compilation (JIT) to optimize run instead of execution in interpreted mode. We also ensured that garbage collector (GC) had not much impact on the gained performance. To do so we took average performance for steps 4–11 or peak performance (maximum of steps performance) for the whole simulation.

2) Single node execution: The first benchmark was made to determine the impact of hyper-threading on execution. The tests were run on HazelHen supercomputer using `aprun` command that allows allocating 24 or 48 CPUs per node depending on the value of `-j` parameter (1 and 2 respectively; depth (`-d`) parameter was accordingly set to 24 and 48). In the benchmark, we also tried to determine, if it was better to run two JVMs, each using 12/24 PCJ threads

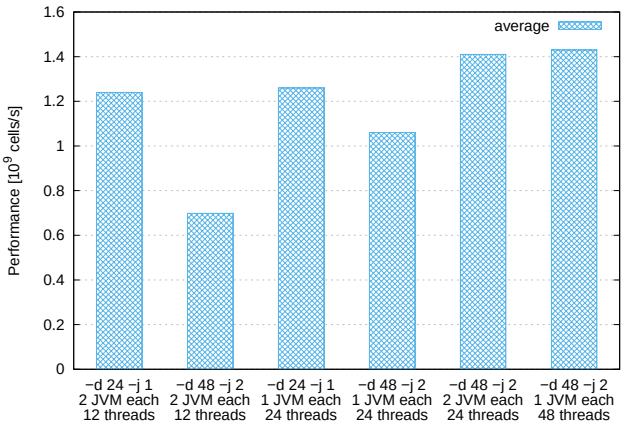

Figure 9. Performance results for various parameters, number of JVMs and PCJ threads for $604{,}800 \times 604{,}800$ cells running on 1 node of Cray XC40 at HLRS.

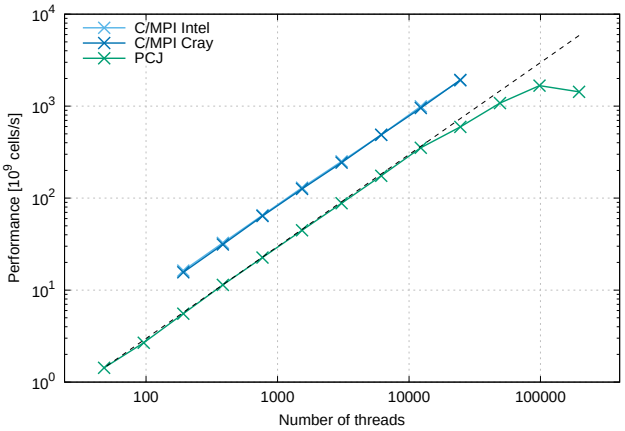

Figure 10. Strong scalability of *Game of Life* implemented with Java/PCJ and C/MPI for $604{,}800 \times 604{,}800$ cells running on Cray XC40 at HLRS. Ideal scaling for PCJ is drawn. Results for MPI using Intel and Cray compilers practically overlap.

or one JVM with 24/48 PCJ threads.

As it is presented in Fig. 9, the best results were obtained utilizing 48 PCJ threads per node. In this situation, the number of JVMs did not matter much – the average performance is similar, slightly better if only one JVM per node was used. Additionally, when using a smaller number of PCJ threads, setting the proper value of used CPUs per node is very important as it can affect the performance.

3) Strong scaling: We have also done benchmarks to see the strong scaling of the *Game of life* application on the HazelHen system. Based on the previous results we decided to fully occupy computational nodes – to use 48 PCJ threads per node.

In the first run, we decided to use universe size $604{,}800 \times 604{,}800$ that could be allocated on the single node. The results presented in Fig. 10 show that the application scales almost linearly up to $98{,}304$ PCJ threads (2048

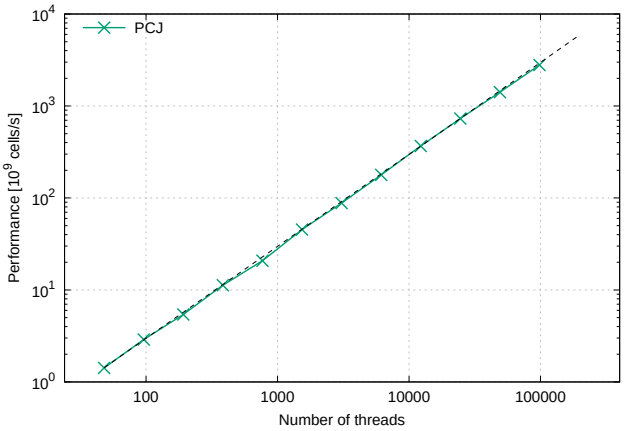

Figure 11. Weak scalability of *Game of Life* application when processing about 221,184,000 cells per thread running at Cray XC40 at HLRS.

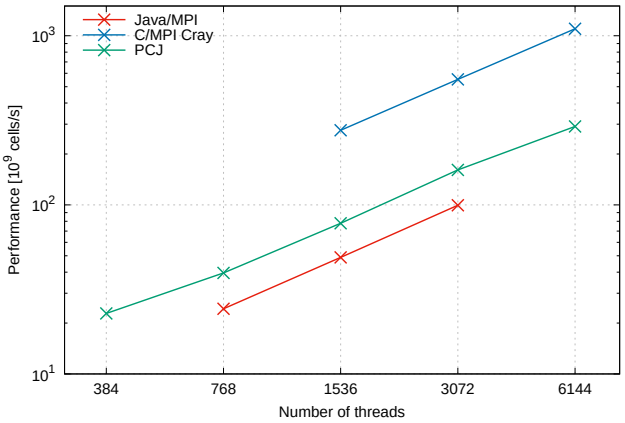

Figure 12. Strong scalability of *Game of Life* implemented with Java/PCJ, Java/MPI and C/MPI for 2,419,200×2,419,200 cells running at Cray XC40 at ICM with 24 threads/ranks per node.

nodes). Usage of 196,608 PCJ threads (4096 nodes) gives similar performance. We also compared PCJ implementation with a similar application written in C using MPI. We were able to run MPI code only from 1 node to up to 512 nodes due to the limits of the available computational time. However, for 1 and 2 nodes (48 and 96 MPI ranks), the MPI version could not be computed due to *Cannot allocate memory* error. For the larger number of nodes, the performance of MPI version was up to 3 times higher than for PCJ version.

We also decided to run benchmark using larger universe (2,419,200×2,419,200 cells). The size was so big, that it did not fit in the memory of 16 nodes (768 PCJ threads). We were able to run a benchmark using from 32 to 2048 nodes (from 1536 up to 98,304 PCJ threads). The scalability for that size was almost ideal for the whole range of nodes. The same scalability was achieved for MPI version but the performance was up to 3 times higher than for PCJ implementation. However, it was not possible to execute test using MPI for 32 nodes (1536 MPI ranks) due to lack of memory.

4) Weak scaling: Next benchmark that we run on HazelHen supercomputer was weak scaling. Each thread processed about 221,184,000 $= 2^{16} \cdot 3^3 \cdot 5^3 cells/step$. The exact value depends on the number of threads used in the execution that threads generated universe of square size (103,008×103,008 cells for 48 PCJ threads to 4,662,960×4,662,960 cells for 98,304 PCJ threads).

The results presented in Fig. 11 show ideal scaling for a whole range of nodes and threads used (more than 98% parallel efficiency for 2048 nodes).

5) Comparison with MPI: As the Open MPI in version 3.0.0 gives Java bindings for the MPI, we decided to run benchmarks using this technology. We run the following benchmarks on Okeanos supercomputer where we were able to run only 24 Java/MPI ranks per node. Usage of 48 Java/MPI ranks per node resulted in unpredictable behavior and errors (*error creating local BTE/FMA CQ*, *error creating local SMSG CQ*, *error creating remote SMSG CQ*). However, those error messages in some unclear situations did not prevent the application to run and finish successfully. Moreover, due to the limit of 4096 ranks per job for Java/MPI in our installation, it was not possible to execute benchmark using 6144 Java/MPI ranks (on 256 nodes).

Figure 12 presents the performance results obtained on universe consisting of 2,419,200×2,419,200 cells. For benchmarking Java/MPI, PCJ and C/MPI version of *Game of Life* we decided to run each version with the same number of threads/ranks – i.e. 24 per node. It was not possible to run the benchmark using Java/MPI on 16 nodes using 384 Java/MPI ranks as there were thrown `java.lang.OutOfMemoryError`. The same error occurred for PCJ, but using 8 nodes (192 PCJ threads) and not for 16 nodes. For pure MPI version, the errors associated with insufficient memory (*Cannot allocate memory* error) happened for 32 nodes (768 MPI ranks). The performance of PCJ version, similarly to results obtained on HazelHen, is about three times lower than for C/MPI but is about two times higher than for Java/MPI. In our opinion, it is due to fact, that it was necessary for Java/MPI to use direct buffers for exchanging data to be able to use nonblocking sending method (`MPI.COMM_WORLD.iSend` ≡ `MPI_Isend`), so after preparing data, it had to be copied to the buffers before sending and the Java/MPI is binding so before sending data it was necessary to call native methods from within JVM.

6) MPI compilation optimizations: We have also performed a comparison of the MPI performance depending on the compiler optimizations level using Cray and Intel MPI compilers as well as a comparison of Oracle JVM with GraalVM. GraalVM [21], [22] is the Java Virtual Machine based on OpenJDK that has just-in-time compiler focused on bringing the best peak performance for the

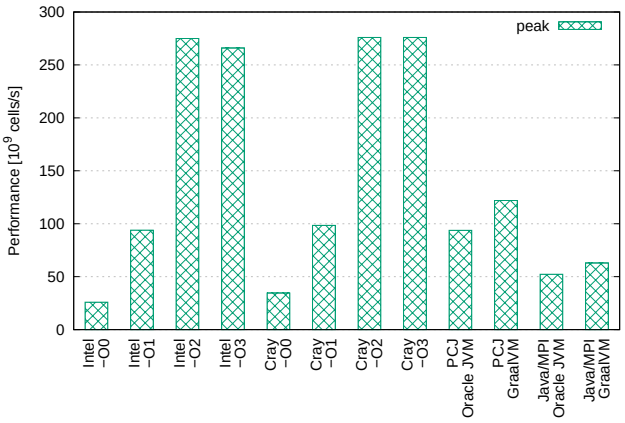

Figure 13. Performance results for various implementations and compiler options. Data obtained for universe consisting of 2,419,200×2,419,200 cells running at 64 nodes of Cray XC40 at ICM.

```
@Storage(MyClass.class)
enum Shared { sum, usersCount }
long sum;
int usersCount;
                          ...
myUsers = loadUsers(PCJ.myId());
                          ...
long s = myUsers.stream()
                .mapToLong(User::getAge)
                .sum();
PCJ.putLocal(s, Shared.sum);
PCJ.putLocal(myUsers.size(), Shared.usersCount);
PCJ.barrier();
s = pcj_reduce(Shared.sum);
int count = pcj_reduce(Shared.usersCount);
if (PCJ.myId() == 0) {
     double average = (double) s / count;
}
```

Listing 9. Implementation of the map-reduce algorithm in Java using PCJ library. The distributed data is processed by each thread independently.

Java applications. For abovementioned comparison, we have used the newest freely available version of GraalVM – Community Edition 1.0 RC1.

The results of *Game of Life* application that use universe consisting of 2,419,200×2,419,200 cells are presented in Fig. 13. The compilation without optimization flags implies the second level of optimizations. The Java and PCJ version runs like MPI code compiled using the first level of optimizations. The Java/MPI version runs almost two times less efficient than the first level of optimizations, but almost two times more efficient than MPI without optimization. The performance for code compiled with Cray and Intel compilers is similar. The usage of GraalVM gives performance improvements of 12% for PCJ based code and 20% for Java/MPI version of the application. The improvement is obtained without modifying or recompiling the code – just by using different Java Virtual Machine for the execution.

V. Big Data processing

In the map-reduce paradigm, usually connected with Big Data processing, the parallel execution is performed through data distribution. Each thread is performing calculations on the local data and partial results are gathered in the reduction process. Partial sums and the number of elements processed are gathered in the second phase – the number of elements processed by each thread is usually the same but can differ. Upon obtaining of data from all threads, the reduction operation can be performed. This scenario can be easily implemented in the PGAS model using PCJ library as presented in Listing 9. Despite the usage of distributed data, the computation kernel is identical to the sequential implementation. The number of additional lines of code needed to perform distributed calculations is limited, with the code responsible for results reduction being the main addition.

A. WordCount

WordCount is a simple piece of code that demonstrates basics of programming in map-reduce paradigm. Test program reads an input file line-by-line and counts the number of unique words occurring in each line. Reduction step gathers computed partial results. In the end, a mapping between all the unique words in whole text and number of their occurrences are emitted.

In PCJ code the word calculations are divided into two steps. Mapping phase utilities the word-division code and partial results are stored in a shared global variable, unique to every thread of execution (see Listing 10). No prior text transformations are performed (for example, stop word list and stemming are not utilized), so – depending on input formatting and used word division algorithms – glyphs like punctuation marks and their combinations might be considered a unique word; the same goes for different grammatical forms of the same word. After this phase, a reduction occurs with thread 0 as root. No overlap between two phases is facilitated. The reduction policies are a major contribution to the overall scalability results and in case of PCJ three distinct policies were implemented:

- *Reduction 1* – 2-step reduction scheme; the first step consists of intra-node reduction, in second step thread 0 collects partial results from remote computation nodes.
- *Reduction 2* – 2-step reduction scheme in which all threads affiliated with node 0 performed remote reduction; remote computation nodes are assigned to node 0 threads on a round-robin fashion; after this step intra-node 0 reduction is performed.
- *Reduction 3* – hypercube-based reduction (presented on Listing 11).

1) Input data: Two novels were chosen as a textual corpus of the text. We have used an UTF-8 encoded plain text English translation of Lev Tolstoy's *War and Peace* [23],

```
//Shared variable definitions
@Storage(WordCountPcj.class)
enum Shared {
  //global wordcount, used in reduction phase
  reducing,
  //thread-local wordcounts
  localCounts
}
public HashMap<String, Integer> localCounts;
public HashMap<String, Integer> reducing;

private void countWords() throws IOException {
  Pattern WORD_BOUNDARY
        = Pattern.compile("\\s*\\b\\s*");
  Map<String, Integer> wordCounts
        = new HashMap<>();
  Files.readAllLines(Paths.get(myFileName),
                     StandardCharsets.ISO_8859_1)
    .stream()
    .map(WORD_BOUNDARY::split)
    .flatMap(Arrays::stream)
    .filter(word -> !word.isEmpty())
    .forEach(word ->
        wordCounts.merge(word, 1, Integer::sum));
  PCJ.putLocal(wordCounts, Shared.localCounts);
}
```

Listing 10. Mapping phase and shared variable definitions in PCJ WordCount implementation.

```
private void getGlobalCountHypercube() {
  resultMap = PCJ.getLocal(Shared.localCounts);
  int mask = 0;
  int d = Integer.numberOfTrailingZeros(
                          PCJ.threadCount());
  int myId = PCJ.myId();
  PCJ.barrier();
  for (int i = 0; i < d; i++) {
    if ((PCJ.myId() & mask) == 0) {
      if ((myId & (1 << i)) != 0) {
        int dest = myId ^ (1 << i);
        PCJ.put(resultMap, dest, Shared.reducing);
        PCJ.barrier(dest);
      } else {
        int src = myId ^ (1 << i);
        PCJ.barrier(src);
        HashMap<String, Integer> reducedArgument
                  = PCJ.getLocal(Shared.reducing);
        reducedArgument.forEach((k, v) ->
             resultMap.merge(k, v, Integer::sum));
      }
    }
    mask ^= (1 << i);
    PCJ.barrier();
  }
}
```

Listing 11. Hypercube-based reduction in PCJ WordCount example.

a file of 3.3 MB, and lesser-known, but nevertheless considerate in length plain ISO 8859-1 encoded text of original French version of Georges de Scudéry's *Artamène ou le Grand Cyrus* [24], one of the longest novels ever written, totaling in 10 MB file size. Different encodings were accounted for in the code. Whilst datasizes itself are quite small, they have been the basis for the weak scalability testing, thus forming a sizable dataset for larger numbers of threads.

During the weak scalability testing, each file was replicated n times, for $n \in \{1, 2, 4, 8, 16, \ldots, 8192\}$. This allowed to affiliate each PCJ thread with one input file: each PCJ thread read its own copy of input data. We have used default filesystem available on XC40 (Lustre), yet there exists a possibility to use other solutions (e.g. a version of PCJ WordCount that utilizes HDFS was successfully developed and is described elsewhere [6]). For comparison, we have implemented a scenario in which all threads read data from the same file located in the shared Lustre filesystem accessible by all nodes. To establish a configuration that allowed for the best performance, different combination of thread counts and number of nodes were evaluated (for example, 4 threads could have been started on 4 nodes – using 1 PCJ thread per node; or 2 nodes – 2 PCJ threads per node; or 1 node running 4 threads). The effect of hyper-threading on the overall performance was also studied (the option `--hint=[no]multithread` was passed to the queuing system). The figures show the times used for the best-performing configuration of nodes and hyper-threading policy per given number of threads. For mapping phase, which is I/O-bound, the best results were achieved, *grosso modo*, with a single PCJ thread per node. Conversely, for reduction phase the best timing results were achieved with hyper-threading turned off and with the largest possible number of PCJ threads running on the same node (thus reducing interconnect congestion). This corroborates the results achieved earlier with hypercube-based all-to-all code in case of FFT. .

2) Mapping phase: The performance of the mapping phase for the de Scudéry's novel is presented in Fig. 14. Results are obtained in the weak scaling mode, therefore with the increased number of threads, the size of processed data increases. The best processing time is plotted, therefore up to 256 threads, the time for execution of the one thread per node is presented. The execution time of the mapping phase does not depend on the number of threads up to 256 which expresses almost ideal scaling. For the larger number of threads, the time necessary for the mapping increases which is related to the increase of the I/O time. With the multiple threads running on the single node, we have a race condition in the access to the disk for reading data which increases processing time. The race situation occurs while reading data from the single file or when multiple files are used. More detailed description of the PCJ I/O performance

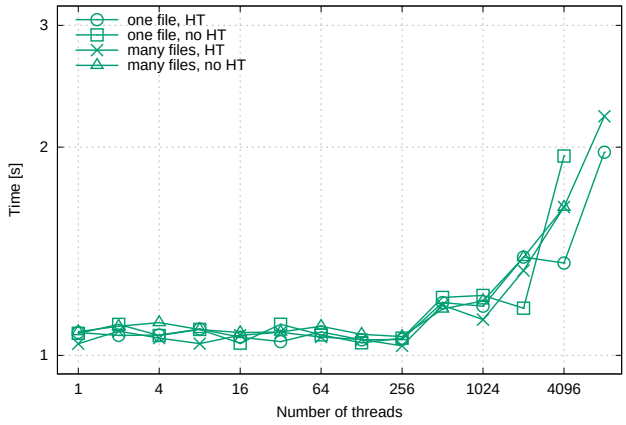

Figure 14. Execution time of the mapping phase of the PCJ *WordCount* application for different methods of reading input data. The results are presented for weak scaling mode with and without hyper-threading.

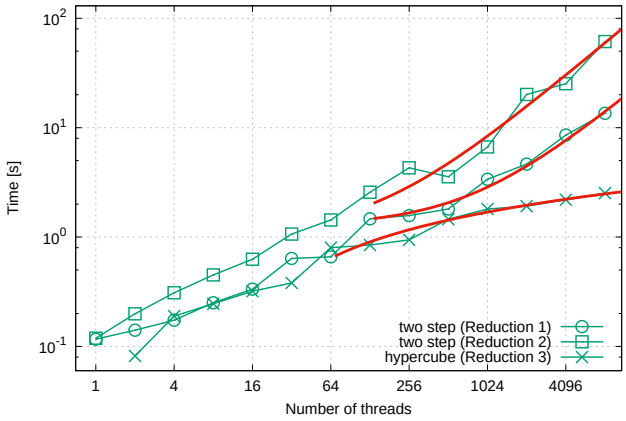

Figure 15. Execution time of the different reduction scenarios in the weak scaling mode. Asymptotic behavior fitted as linear (Reduction 1 and Reduction 2) or logarithmic function (Reduction 3) is shown.

for different filesystems is described elsewhere [12].

3) Reduction phase: The execution time of the reduction phase for the different scenarios described above is presented in Fig. 15. The *Reduction 1* policy scales linearly ($O(n)$) with the increased number of threads (intuitively: increase in the number of threads used increases the number of nodes used as well, thus the time node 0 takes to read intra-node reduction results from remote nodes grows). Similar scaling is obtained for the Reduction 2, however, it is slower due to the increased processing time. The hypercube-based reduction (Reduction 3) scales as $O(log(n))$ and especially for a large number of threads is the fastest one.

4) Overall performance: The total execution time for the hypercube-based reduction for both input files used in tests is presented in Fig. 16. The execution time is dominated by the reduction phase, therefore in all cases, it increases with the number of threads. The difference between small and large file used is visible for a small

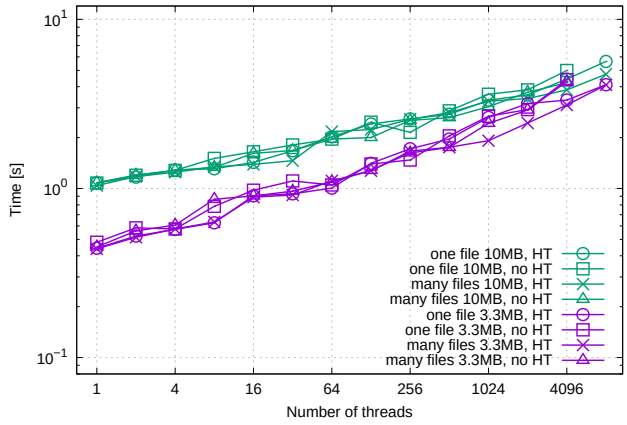

Figure 16. *WordCount* performance results (total execution time, weak scaling) for the different methods of reading data with and without hyper-threading. Lines plotted for hypercube type reduction and for both file sizes used.

number of threads where the time for reduction phase is smaller than for mapping. The execution time of mapping phase is proportional to the size of the data processed, the execution of the reduction phase depends less on the size of the input data. Since the installation of the Hadoop or Spark on the Cray XC40 system is not trivial, we were not able to perform direct performance comparison. However, such comparison performed using x86 cluster confirms that PCJ implementation of the *WordCount* application is at least 7 times faster than Hadoop one [6].

VI. Future work

Since beginning PCJ library is using sockets for transfer data between nodes. This design was straightforward, however, the novel communication hardware such as Cray Aries or InfiniBand interconnects cannot be fully utilized. There is ongoing work to use novel technologies in PCJ, however, we are looking for Java interfaces which can simplify integration. DiSni or iVerbs seems to be a good choice, however, both are based on the specific implementation of communication and their usage by PCJ library is not easy. We hope to solve this issue soon.

Another reason for low communication performance is mentioned the problem of a copy of the data during send and receive process. This cannot be avoided due to the Java design, and 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 PCJ library but for Java in general as it is degrading Java/MPI ping-pong performance for large datasets.

Compare to other tools, PCJ library has no fault tolerance mechanism, however, an experimental version exists [25] and will be integrated with the main release soon.

VII. Conclusion

PCJ library is highly scalable, easy to use tool for development of parallel applications, including Big Data processing. PCJ library works well especially for large applications where the overlap between calculations and communications can be utilized. In such cases, PCJ library allows for easy code development resulting in very good scalability and performance. One should note that PCJ implementation is much easier and provides better application performance than Java MPI bindings. Moreover, the development with PCJ is much easier than in the case of other tools. It requires fewer libraries to use and minimizes the number of language constructs used. The resulting code is usually shorter and more readable.

PCJ applications can be developed and tested using standard Java environment, the time-consuming installation and maintenance of the infrastructure tools such as Hadoop are not required.

Acknowledgment

This research was carried out with the support of the Interdisciplinary Centre for Mathematical and Computational Modelling (ICM) University of Warsaw providing computational resources under grants no GB65-15, GA69-19. The authors would like to thank CHIST-ERA consortium for financial support under HPDCJ project (Polish part funded by NCN grant 2014/14/Z/ST6/00007) and PRACE for awarding access to resource HazelHen at HLRS. The HazelHen results were obtained during a visit to HLRS sponsored by EuroLab-4-HPC as cross-site collaboration grant – Marek Nowicki would like to thank Dr. Alexey Cheptsov for being the host and for the received help during the stay.

References

[1] Anderson, M., Smith, S., Sundaram, N., Capotă, M., Zhao, Z., Dulloor, S., Satish N., Willke, T. L.: Bridging the gap between HPC and Big Data frameworks. Proceedings of the VLDB Endowment, 10(8), pp. 901–912. (2017)

[2] PCJ Home: http://pcj.icm.edu.pl [Accessed: 12 April 2018.]

[3] Nowicki, M., Górski, Ł., Grabrczyk, P., Bała, P.: PCJ - Java library for high performance computing in PGAS model. In: International Conference on High Performance Computing and Simulation, HPCS 2014, IEEE, pp. 202–209 (2014)

[4] Nowicki, M., Ryczkowska, M., Górski, Ł., Szynkiewicz, M., Bała, P.: PCJ - a Java library for heterogenous parallel computing. In: Recent Advances in Information Science (Recent Advances in Computer Engineering Series vol 36), WSEAS Press, pp. 66–72 (2016)

[5] 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. PPAM 2017. Lecture Notes in Computer Science, vol 10778. Springer, Cham, pp. 288–297 (2018)

[6] 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. PPAM 2017. Lecture Notes in Computer Science, vol 10778. Springer, Cham, pp 318–327 (2018)

[7] Ryczkowska, M., Nowicki, M., Bała, P.: Level-synchronous BFS algorithm implemented in Java using PCJ Library. In: International Conference on Computational Science and Computational Intelligence (CSCI), Las Vegas, NV, USA, 2016, pp. 596–601.

[8] Liang, J. J., Qu, B. Y., Suganthan, P. N.: Problem Definitions and Evaluation Criteria for the CEC 2014 Special Session and Competition on Single Objective Real-Parameter Numerical Optimization. *Technical Report 201311*, Computational Intelligence Laboratory, Zhengzhou University, Zhengzhou China, 2014

[9] Górski, Ł., Rakowski, F., Bała, P.: Parallel Differential Evolution in the PGAS Programming Model Implemented with PCJ Java Library. In: International Conference on Parallel Processing and Applied Mathematics. PPAM 2015. Lecture Notes in Computer Science, vol 9573. Springer, Cham, pp. 448–458 (2016)

[10] Rakowski, F., Karbowski, J.: Optimal synaptic signaling connectome for locomotory behavior in Caenorhabditis elegans: Design minimizing energy cost. PLoS Computational Biology 13 (11), e1005834

[11] 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. ICA3PP 2017. Lecture Notes in Computer Science, vol 10393. Springer, Cham, pp. 503–512 (2017)

[12] Nowicki, M., Bzhalava, D., Bała, P.: Massively Parallel Implementation of Sequence Alignment with BLAST Using PCJ Library. J. Comput. Biol. (in press) (2018)

[13] Vega-Gisbert, O., Roman, J. E., Squyres, J. M.: Design and implementation of Java bindings in Open MPI. Parallel Computing, vol. 59, pp. 1–20 (2016)

[14] RandomAccess Rules: http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/ [Accessed: 12 April 2018.]

[15] Manninen, P., Richardson, H.: First Experiences on Collective Operations with Fortran Coarrays on the Cray XC30. In: 7th International Conference on PGAS Programming Models, page 222 (2013)

[16] Foster, I.: Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. Addison-Wesley Longman Publishing Co., Inc. 1995

[17] Takahashi, D., Kanada, Y.: High-Performance Radix-2, 3 and 5 Parallel 1-D Complex FFT Algorithms for Distributed-Memory Parallel Computers. The Journal of Supercomputing, 15(2), pp. 207–228 (2000)

[18] Mellor-Crummey, J., Adhianto, L., Jin, G., Krentel, M., Murthy, K., Scherer, W., Yang, Ch.: Class II submission to the HPC Challenge award competition Coarray Fortran 2.0. http://www.hpcchallenge.org/presentations/sc2011/hpcc11_report_caf2_0.pdf. [Accessed: 12 April 2018.]

[19] Wendykier P., Nagy J.G.: Large-Scale Image Deblurring in Java. In: International Conference on Computational Science – ICCS 2008. ICCS 2008. Lecture Notes in Computer Science, vol 5101. Springer, Berlin, Heidelberg, pp. 721–730 (2008)

[20] Gardner, M.: Mathematical Games: The fantastic combinations of John Conway's new solitaire game "Life". Scientific American, vol. 223, no. 4, October 1970, pp. 120–123.

[21] GraalVM Home: http://www.graalvm.org [Accessed: 15 May 2018.]

[22] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, M. Wolczko. One VM to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 187–204. ACM, 2013.

[23] Tolstoy, L.: War and Peace. Random House 2016

[24] Scudéry, M. de: Artamène ou le grand Cyrus Chez Avgvstin Covrbe', Imprimeur & Libraire ordinaire de Monseigneur le Duc d'Orleans, dans la petite Salle du Palais, à la Palme 1972

[25] Szynkiewicz, M., Nowicki, M.: Fault-Tolerance Mechanisms for the Java Parallel Codes Implemented with the PCJ Library. In: Parallel Processing and Applied Mathematics. PPAM 2017. Lecture Notes in Computer Science, vol 10778. Springer, Cham, pp. 298–307 (2018)