PCJ. Parallel Computing in Java

Marek Nowicki, Piotr Bała
ICM UW, WMiI UMK

May 5, 2017

Contents

1 Introduction **3**
- 1.1 Motivation . . . 3
- 1.2 PCJ history . . . 3

2 PCJ Fundamentals **5**
- 2.1 Execution in multinode multicore environment . . . 6

3 PCJ basics **7**
- 3.1 Starting PCJ application . . . 7
- 3.2 Number of tasks, tasks id's . . . 8
- 3.3 Task synchronization . . . 8
- 3.4 Shared variables . . . 9
- 3.5 Access to a shared variable . . . 9
  - 3.5.1 get() and put() . . . 9
  - 3.5.2 asyncGet() . . . 10
  - 3.5.3 asyncPut() . . . 10
  - 3.5.4 broadcast() . . . 11
- 3.6 Array as a shared variable . . . 11
  - 3.6.1 get() . . . 11
  - 3.6.2 put() . . . 12
  - 3.6.3 broadcast() . . . 12
- 3.7 Output to console . . . 13
- 3.8 Input from console . . . 13
- 3.9 Reading from file . . . 14
- 3.10 Parallel reads from multiple files . . . 14
- 3.11 Output to the file . . . 15
- 3.12 Java and PCJ library . . . 15

4 Executing PCJ applications **17**
- 4.1 Linux Cluster . . . 18
- 4.2 Linux Cluster with Slurm . . . 18
- 4.3 IBM Power 7 (AIX) with Load Leveler . . . 19
- 4.4 IBM BlueGene/Q . . . 19
- 4.5 Cray XC40 . . . 19
- 4.6 Intel KNL @ Rescale . . . 20

Chapter 1

Introduction

1.1 Motivation

Nowadays, almost everyone interested in parallel and distributed calculations pays a lot of attention to the development of the hardware. However, changes in hardware are associated with changes in the programming languages. A good example is Java with its increasing performance and parallelization tools introduced in Java SE 5 and improved in Java SE 6 [3]. Java, from the beginning, put emphasis on parallel execution introducing as far back as in the JDK1.0 the Thread class. The parallelization tools available for Java include solutions based on various implementations of the MPI library [4], distributed Java Virtual Machine [5] and solutions based on Remote Method Invocation (RMI) [6].

PCJ is a library [1, 2] for Java language that helps to perform parallel and distributed calculations. The current version is able to work on the multicore systems connected with the typical interconnect such as ethernet or infiniband providing users with the uniform view across nodes.

The library implements partitioned global address space model [7] and was inspired by languages like Co-Array Fortran [8], Unified Parallel C [9], and Titanium [10]. In contrast to listed languages, the PCJ does not extend nor modify language syntax. For example, Titanium is a scientific computing dialect of Java, defines new language constructs and has to use a dedicated compiler. When developing the PCJ library, we put emphasis on compliance with Java standards. The programmer does not have to use additional libraries, which are not part of the standard Java distribution. Compared to the Titanium, PCJ does not need a dedicated compiler to preprocess code.

1.2 PCJ history

The first prototype version of PCJ [2] has been developed from scratch using the Java SE 7. Java SE 7 implements Sockets Direct Protocol (SDP), which can increase network performance over infiniband connections. Then the internode communication has been added allowing users to run multiple PCJ threads within single Java Virtual Machine.

PCJ version 3 has been developed in 2013 and includes many bug fixes and improvements compare to the initial version. Especially the user's interface has been stabilized.

PCJ version 4 has been released in November 2014. This version has been quite stable and has been used as ground for further developments. In particular, updated version (PCJ 4.1) allowed calling JCuda code.

Next version, PCJ 5, has been developed in 2016. This version introduced new handling of the *Shared* variables and some changes in the programmers API. The names of the methods have been changed to distinguish between synchronized and asynchronous communication. For example, `PCJ.get()` has been changed to be blocking method, for the asynchronous communication `PCJ.asyncGet()` has been introduced.

Chapter 2

PCJ Fundamentals

The PCJ library was created with some principles.

Tasks (PCJ threads) Each task executes its own set of instructions. Variables and instructions are private to the task. PCJ offers methods to synchronize tasks.

Local variables Variables are accessed locally within each task and are stored in the local memory.

Shared variables Shared variables are accessible from other PCJ threads. Every shared variable has to be a field in a class. This class is pointed by *enum class* by the `@Storage` annotation. Type of the shared variable is inferred from the type of the associated field. There can be multiple enum classes that point to the same class. Every enum class has to be registered in the thread that wants to have a shared variables. Other threads can access shared variables by providing registered enum constant name but do not have to register enum itself. Storages can be automatically registered on application start phase if the `@RegisterStorage` annotation is used on actual `StartPoint` class.

There is a distinction between nodes and tasks (PCJ threads). One instance of JVM is understood as a node. In principle, it can run on a single multicore node. One node can hold many tasks (PCJ threads) – separated instances of threads that run calculations. This design is aligned with novel computer architectures containing hundreds or thousands of nodes, each of them built of several or even more cores. This forces us to use different communication mechanism for inter- and intranode communication.

In the PCJ there is one node called *Manager*. It is responsible for setting unique identifiers to the tasks, sending messages to other tasks to start calculations, creating groups and synchronizing all tasks in calculations. The *Manager* node has its own tasks and can execute parallel programs.

2.1 Execution in multinode multicore environment

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 physical 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 communication between different PCJ 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 mechanism. Moreover, nodes are organized in the graph which allows optimizing global communication.

Chapter 3

PCJ basics

In order to use PCJ library you have to download `PCJ-5.x.y.jar` file from the PCJ website: **pcj.icm.edu.pl**. The `PCJ-5.x.y.jar` should be located in the directory accessible by java compiler and java runtime, for example in the lib directory of your IDE.

3.1 Starting PCJ application

Starting PCJ application is simple. It can be built in the form of a single class which implements `StartPoint` interface. The `StartPoint` interface provides necessary functionality to start required threads, enumerate them and performs the initial synchronization of tasks (PCJ threads).

`PCJ.deploy()` method initializes application using a list of nodes provided as the second argument. List of nodes contains internet address of the computers (cluster nodes) used in the simulations.

```
10 import java.io.IOException;
11 import org.pcj.*;
12
13 public class HelloWorld implements StartPoint {
14
15     public static void main(String[] args) throws IOException {
16         String nodesFile  = "nodes.txt";
17
18         PCJ.start(HelloWorld.class, new NodesDescription("nodes.
               txt"));
19     }
20
21     @Override
22     public void main() throws Throwable {
23                 System.out.println("Hello World!'');
24              }
25 }
```

: PcjHelloWorld.java

The code should be saved in the `PcjHelloWorld.java` file and compiled. Than it can be run using standard java command:

```
javac -cp .:PCJ-5.x.y.jar PcjHelloWorld.java
java  -cp .:PCJ-5.x.y.jar PcjHelloWorld
```

The expected output is presented below:

```
maj 03, 2017 9:04:35 PM org.pcj.internal.InternalPCJ start
INFO: PCJ version 5.0.SNAPSHOT-4a9461f built on 2017-04-29
   18:24:40.743 CEST.
maj 03, 2017 9:04:35 PM org.pcj.internal.InternalPCJ start
INFO: Starting HelloWorld with 2 threads (on 1 node)...
Hello World!
Hello World!
maj 03, 2017 9:04:35 PM org.pcj.internal.InternalPCJ start
INFO: Completed HelloWorld with 2 threads (on 1 node) after 0h 0m
   0s.
```

The above scenario allows running PCJ application within single Java Virtual Machine. The same code can be run using multiple JVM's.

3.2 Number of tasks, tasks id's

In order make clear names, the tasks are called PCJ threads. PCJ library offers two useful methods:

```
40 public static int PCJ.threadCount()
```

which returns a number of tasks (PCJ threads) running and

```
45 public static int PCJ.myId()
```

which returns id of the PCJ thread. PCJ thread id is integer value of the range from `0` to `PCJ.threadCount()-1`.

3.3 Task synchronization

PCJ offers `PCJ.barrier()` method which allows to synchronize all PCJ threads. While this line is reached, the execution is stopped until all PCJ threads reach the synchronization line.

Remember, that this line has to be executed by all PCJ threads.

```
50 public static void PCJ.barrier()
```

The user can provide an argument to `barrier()` which is integer id of the PCJ thread to synchronize with the current.

```
55 public static void PCJ.barrier(int id)
```

In this case, two tasks are synchronized: one with the given id and one which starts `barrier()` method. Please note that both tasks have to execute method.

3.4 Shared variables

The general rule is that variables are local to the tasks and cannot be accessed from another task. PCJ offers a possibility to mark some variables as `Shared` using Java annotation mechanism.

Every shared variable has to be a field in a class. This class is pointed by *enum class* by the `@Storage` annotation. Type of the shared variable is inferred from the type of the associated field. There can be multiple enum classes that point to the same class.

Every enum class has to be registered in the thread that wants to have a shared variables. Other threads can access shared variables by providing registered enum constant name but do not have to register enum itself. Storages can be automatically registered on application start phase if the `@RegisterStorage` annotation is used on actual `StartPoint` class.

```
60 @RegisterStorage(PcjApplication.Shared.class)
61 public class PcjApplication implements StartPoint {
62
63     @Storage(PcjApplication.class)
64     enum Shared { a }
65     public long a;
```

The Shared annotation can be applied to the single variables, arrays as well as more complicated objects.

3.5 Access to a shared variable

The PCJ library provides methods to access shared variables, eg. to get the value stored in the memory of another task (`get()`)or to modify variable located in the memory of another task (`put()`).

3.5.1 get() and put()

Both methods: `get()` and `put()` perform one-sided communication. This means, that access to the memory of another task is performed only by the task which executes get or put methods. The task which memory is contacted does not need to execute these methods.

The example code presents how to assign a value of the variable `a` at PCJ thread 3 to the variable `b` at PCJ thread 0.

```
70 double c;
71 if (PCJ.myId()==0) {
72    c =(double) PCJ.get(3, Shared.a);
73 }
```

Next example presents how to assign value 4.0 to the variable `a` available at the PCJ thread 5. This operation is performed by the PCJ thread 0.

```
75 if (PCJ.myId()==0) {
76    PCJ.put(5.0, 3, Shared.a);
77 }
```

It is important to provide the name of shared variable as a `String`.

The communication is performed in an asynchronous way, which means that user has no guarantee that value has been changed or transferred from a remote task. This may cause some problems, especially for nonexperienced users. PCJ provides additional methods to solve this problem.

3.5.2 asyncGet()

The `asyncGet()` method from PCJ library returns value of type `PcjFuture` and the value has to be casted to the designated type. The execution of the method ensures that result is transferred from the remote node. The next instruction will be executed after the local variable is updated.

PCJ allows also for asynchronous, nonblocking communication. For this purposes the `PcjFuture` is used. The `PcjFuture` stores remote value in the local memory and provides methods for monitoring is process has finished. Additional method `get()` is then used to copy transmitted value to the local variable.

Example code presents how to copy the value of the remote variable a from the task number 5 to task 0.

```
80 if (PCJ.myId() == 1){
81     PcjFuture d = PCJ.asyncGet(0, Shared.a);
82         // some commands
83     double c = (double) d.get();
84     System.out.println("c ="+c);
85 }
```

The remote value is transferred to the variable `d` in an asynchronous way. When data is available it is stored in the local variable `c` using synchronous method `get()`. This command is executed after local variable `d` is updated.

The data transfer can be ensured using `PCJ.waitFor(Shared.a)` command. It waits until data transfer is finished and variable `a` is updated.

aother possibility is to use `isDone()` method from `PcjFuture` class.

3.5.3 asyncPut()

Each PCJ thread can initialize update of the variable stored on the remote task with the `put()` method. In the presented example task number 2 updates variable `a` in the memory of task 0.

```
90         if (PCJ.myId() == 0) {
91             PCJ.monitor(Shared.a);
92         }
93         if (PCJ.myId() == 1) {
94             PCJ.put(10.0, 0, Shared.a);
95         }
96         if (PCJ.myId() == 0) {
97             PCJ.waitFor(Shared.a);
98             System.out.println("a= " + a);
99         }
```

The process is asynchronous, therefore the method `waitFor()` is used to wait for transfer to be completed. Method `monitor()` is used to watch for updates of shared variable `Shared.a`.

3.5.4 broadcast()

In order to access variables at all tasks, PCJ provides a broadcast method. This method puts given value to the shared variable at all tasks. This process is one-sided communication and typically is initialized by a single node.

```
100     PCJ.monitor(Shared.a);
101
102     if (PCJ.myId() == 0) {
103         PCJ.broadcast(Shared.a, 2.14) ;
104     }
105
106     PCJ.waitFor(Shared.a);
107     System.out.println("a="+a);
```

In order to synchronize variables we set up a monitor on the variable `Shared.a`. Then broadcast is performed. Finally, all nodes wait until communication is completed and variable `a` is updated.

3.6 Array as a shared variable

The shared variable can be an array. Methods `put()`/`asyncPut()`, `get()`/`asyncGet()` and `broadcast()` allow to use arrays. Therefore user can provide index of the array variable and the data will be stored in the corresponding array element.

3.6.1 get()

It is possible to communicate the whole array as presented below.

```
110     @Storage(ExampleGetPut.class)
111     enum Shared {
112         array
113     }
114     double array[] = new double[10];
115     ...
116     double[] c = new double[10];
117     ...
118     if (PCJ.myId() == 0) {
119          c = (double)  PCJ.get(1, Shared.array);
120     }
121     System.out.println(''c[2] = ''+c[2]);
```

`PCJ.get()` allows also to communicate elements of array. This is done using an additional argument which tells which array element should be communicated.

```
130     @Storage(ExampleGetPut.class)
131     enum Shared {
132         array
```

```
133     }
134     double array[] = new double[10];
135     ...
136     double[] c = new double[30];
137     ...
138     if (PCJ.myId() == 0) {
139          c[2] = (double)  PCJ.get(1, Shared.array, 5);
140     }
141     System.out.println(array[2]);
```

3.6.2 put()

Similar functionality can be achieved with `put()` method.

```
150     @Storage(ExampleGetPut.class)
151     enum Shared {
152         array
153     }
154     double array[] = new double[10];
155     ...
156     if (PCJ.myId() == 0){
157          array[3] = 30.0;
158          // populate array[]
159     }
160
161     PCJ.monitor(Shared.array);
162     if (PCJ.myId() == 0) {
163         PCJ.asyncPut(array, 1, Shared.array);
164                          // can be replaced by put()
165     }
166     if (PCJ.myId() == 1) {
167         PCJ.waitFor(Shared.array);
168     }
169
170     System.out.println(PCJ.myId() + "array" + array[3]);
```

The process is asynchronous, the methods `waitFor()` and `monitor()` are used to watch for updates of shared variable `array`.

3.6.3 broadcast()

The use of an array in the broadcast is similar to the use of the simple variable.

```
180     @Storage(ExampleGetPut.class)
181     enum Shared {
182         array
183     }
184     double array[] = new double[10];
185     ...
186     // populate array[] at PCJ thread 0
187     ...
188     PCJ.monitor(Shared.array);
189
190     if (PCJ.myId() == 0) {;
```

```
191         PCJ.asyncBroadcast(array, Shared.array);
192     }
193
194   PCJ.waitFor(Shared.array);
195
196   System.out.println(PCJ.myId() + " array bcasted " + array[3]);
```

3.7 Output to console

Since PCJ tasks are independent, the output is realized by every task. Simple `System.out.println()` will result in multiple lines in the output. In principle number of lines will be a number of threads. However once PCJ application is run on multiple VM's, the detailed behavior depends on the mechanism used to launch an application. In many cases a user will output from the local virtual machine.

The good practice is to limit I/O operations to the dedicated PCJ thread, for example, one with id equals to 0. This is easily performed using conditional statements and `PCJ.myId()` method.

```
200     if (PCJ.myId()==0) {
201         System.out.println("Hello!");
202      }
```

One should remember, that outed variables could have a different value on different threads.

The output using files could be performed in a similar way.

3.8 Input from console

The input can be performed by each task independently. This makes some problems while executing with multiple threads. In order to reduce the number of I/O operations, the input form the standard input is performed by designated thread (eg. thread with id equals to 0) and that value of the data is broadcasted to the other threads.

```
210     @Storage(ExampleGetPut.class)
211     enum Shared {
212         a
213     }
214     int a;
215     ...
216     Scanner stdin = new Scanner(System.in);
217
218     PCJ.monitor(Shared.a);
219     if (PCJ.myId()==0) {
220         a = stdin.nextInt();
221         PCJ.broadcast(a, Shared.a);
222     }
223     PCJ.waitFor(Shared.a);
224
225     System.out.println(PCJ.myId() + " a = "+a);
```

The input is performed by PCJ thread 0, therefore all other tasks (PCJ threads) have to wait until value of variable `a` is broadcasted. This is realized using `PCJ.monitor()` and `PCJ.waitFor()` methods. Please note that both methods are executed by all tasks while `broadcast()` is one-sided communication and is executed only by task PCJ thread id 0.

Variable `a` can be of different type such as, `double`, `String` etc.

3.9 Reading from file

The reading from the file is performed independently by each thread. Each PCJ thread creates its own file handler and controls reads/writes from the file.

```
230     String b;
231     Scanner sc = null;
232     try {
233         sc = new Scanner(new File("input.txt"));
234     } catch (FileNotFoundException ex) { }
235     b = sc.next();
236     System.out.println(PCJ.myId() + "> " + b);
```

In result each thread receives handler to the file `input.txt` and reads first line from the file. The output looks like:

```
0 >   line1
1 >   line1
```

Each thread can read file independently line by line. If one of the PCJ threads reads more lines, threads can point to the different lines. In result read performed by all threads can return different values.

```
250     b = sc.next();
251     if (PCJ.myId() == 0) {
252         b = sc.next();
253         System.out.println(PCJ.myId() + "> " + b);
254     }
```

Output is as following:

```
0 >   line1
```

3.10 Parallel reads from multiple files

The reading from a single file requires access to this file from all PCJ threads. In the case of the multinode systems, this requires filesystem mounted at all nodes. Such operation requires heavy access to the shared filesystem and can result in the performance decrease.

This situation can be changed in a simple way. Each thread can read from the local file (e.g. `/tmp/file`) or use a file with the different name.

```
270     Scanner sc = null;
271      String f = "input"+PCJ.myId()+".txt";
```

```
272     try {
273         sc = new Scanner(new File(f));
274     } catch (FileNotFoundException ex) { }
275     b = sc.next();
276     PCJ.log(" " + b);
```

In result each threads receive handlers to the files `input0.txt`, `input1.txt`, `input2.txt` etc.

```
0 >   line1_of_input0.txt
1 >   line1_of_input1.txt
```

If files are stored on the local filesystem the input operations are fully independent and will result in the significant speedup. Please note that similar performance can be achieved using distributed file systems such as lustra, gpfs or hdfs.

3.11 Output to the file

Output to the file is organized in a simillar way as input. The user can either write data to the single file located on the shared filesystem or to the local files created on the local storage. Parallel use of the different files is also possible. Please note that usage of the single file decreases performance, especially if it is located on the shared filesystem.

3.12 Java and PCJ library

The PCJ threads are run independently, therefore all operations are executed in parallel. However, there are situations where some attention should be given to the Java code executed as multiple PCJ threads running within a single virtual machine. In such situation, the Java methods can use internal synchronization and are executed sequentially even when invoked from the different PCJ threads. A good example is generation of the random numbers using `Math.random()`.

```
300 double s = 0;
301 for (long i=0; i<n; i++){
302      s = s + Math.radom();
303 }
```

The above code will not scale while running multiple PCJ threads within a single virtual machine, even if running on the multiprocessor/multicore system.

This problem can be removed by using at each PCJ thread `Random` object an calling `nextDouble()` method to generate random number. In this case, even while running multiple PCJ threads on the single node, each of them is using its own instance of the `Random` object which ensures parallel execution of all operations.

```
310 import java.util.Random;
311 ...
312 double s = 0;
313 Random r = new Random();
```

```
314
315 for (long i=0; i<n; i++){
316       s = s + r.nextDouble();
317 }
```

Please remember that in this case instead of the single stream of the pseudo-random numbers, we are using multiple streams of pseudorandom number which nor necessary has the same statistical properties[1].

[1]The generation of the pseudo random number in the parallel applications is a well known problem which received significant number of publications.

Chapter 4

Executing PCJ applications

The compilation and execution of the parallel applications, especially while using some queueing system or another submission environment is not straightforward. In particular, the information about the nodes parallel application will be running on is not available in advance or even during job submission but is determined when a job starts execution.

Most of the systems provide such information through the environment variables and files with the list of nodes used for job execution.

The list of nodes, especially while multiprocessor nodes are present can contain multiple lines with the same names. The multiple entries are used, for example, while running MPI application, to start multiple instances of the parallel application on the single node.

In the case of PCJ library the execution is simple. The most efficient mechanism is to start single Java Virtual Machine on each node. Within this JVM multiple PCJ threads will be run. While running on multiple node, adequate number of JVMs will be started, using `ssh` or `mpiexec` command.

Please remember, that PCJ threads running within single JVM will use Java Concurrency Library to synchronize and to communicate. Communication between PCJ threads running within different JVMs will be performed using Java Sockets.

In such situation in order to run PCJ application we will use two files:

nodes.unique - file containing list of nodes used to run JVMs. In principle, this list contains unique names (no duplicated names).
This file is used by the `mpiexec` or another command to start a parallel application.

nodes.txt - file containing a list of the nodes used to start PCJ threads. This list may contain duplicated entries showing that on the particular node multiple PCJ threads will be started (within single JVM). The number of PCJ threads used to run application (`PCJ.threadsCount()`) will be equal to the number of lines (entries) in this file.

The list of nodes used to run Java application can be transferred to `PCJ.deploy()` or `PCJ.start()` methods as the string which is a name of the file with the node names.

In order to optimize execution on the multinode system, the single Java VM is started on each node.

```
100     PCJ.deploy(Files.class, new NodesDescription("nodes.txt"));
```

4.1 Linux Cluster

The user has to compile PCJ application with Java (Java 8 and above required). Than the `mpiexec` or `mpirun` command is used to run application. The user has to prepare files nodes.unique and nodes.txt as described above. The `mpirun` commnad executes at each node simple bash script which starts java aplication. Example commands which can be run from script or interactive shell. The first command is used to load openmpi environment.

```
module load opempi
mpiexec -hostsfile nodes.unique bash -c 'java -d64  -Xnoclassgc -
    Xrs -cp pcj.jar PcjExampleHelloWorld'
```

4.2 Linux Cluster with Slurm

The execution is similar to the case of Linux Cluster. However, the proper script submitted to the queue to be prepared.

This file contains a definition of the parameters passed the to the queueing system. The parameters include number of nodes required (nodes=128) and indicate that 1 process per node will be executed (ppn=1).

The execution of java application is preceded by the gathering list of the nodes allocated to the job by the queueing system. The unique list of nodes is then stored in the `nodes.unique` file.

Please remember that `nodes.unique` and `nodes.txt` can be different.

```
#!/bin/csh
#PBS -N go
#PBS -l nodes=128:ppn=1
#PBS -l mem=512mb
#PBS -l walltime=0:10:00
#PBS

module load openmpi

cat $PBS_NODEFILE > nodes.txt
uniq $PBS_NODEFILE > nodes.unique

mpiexec -hostsfile nodes.unique bash -c 'java -d64  -Xnoclassgc -
    Xrs -cp pcj.jar PcjExampleHelloWorld'
```

: go.csh

The job is than executed by submitting it with the `qsub` command:

```
qsub go.csh
```

4.3 IBM Power 7 (AIX) with Load Leveler

In order to optimize execution on the multinode systems like IBM Power 7, the PCJ application should exclusively use computer nodes. However, the number of applications running on each nodes is 1 which is Java VM.

The `poe` command is used to invoke Java VM's on the nodes reserved for the execution.

```
#@ job_type = parallel
#@ node = 2
#@ tasks_per_node= 1
#@ queue

cat $LOADL_HOSTFILE > nodes.txt
uniq $LOADL_HOSTFILE > nodes.unique

poe  "java -Xnoclassgc -Xmx6g -cp .:pcj.jar  PcjHelloWorld" -hfile
    nodes.unique -statistic print -bindproc yes -task\_affinity
   core
```

: go_power7.csh

The job is than executed by submitting it with the `llsubmit` command:

```
llsubmit go_power7.csh
```

4.4 IBM BlueGene/Q

The java runtime environment is not yet available on the computing nodes, therefore PCJ applications cannot be run. The work on porting Java to bgq nodes is in progress.

4.5 Cray XC40

There is no specific configuration. The srun from `srun` can be used to start PCJ application.

```
#!/bin/bash -l
#SBATCH -N 2
#SBATCH -n 96
#SBATCH --ntasks-per-node 48
#SBATCH --samples=01:00:00
#SBATCH -A GB56-15

module load java

srun -N 2 -n 2 hostname | sort > nodes.txt

srun -N 2 -n 2 -c 1 java  -d64 -cp .:PCJ-5.x.y.jar HelloWorld
```

: go_xc40.sh

```
sbatch go_xc40.sh
```

4.6 Intel KNL @ Rescale

The rescale web portal allows user to run PCJ code on Intel KNL system. However, in standard configuration Java is not included in the execution path and has to be added manually. Please note that MPI 2017 has to be selected.

```
export JAVA_HOME=/home/rescale/shared/program/java/jre1.8.0_60
export PATH=$PATH:$JAVA_HOME/bin

java -version

mpirun  hostname > nodes.all
uniq nodes.all > nodes.uniq

mpirun -hostfile nodes.uniq -ppn 68 hostname > nodes.txt
mpirun -hostfile nodes.uniq -ppn 1 java -cp .:PCJ-5.0.3-bin.jar
   PcjHelloWorld
```

Acknowledgments

This work has been performed using the PL-Grid infrastructure. The CHIST-ERA support through NCN grant 2014/14/Z/ST6/00007 is acknowledged.

Bibliography

[1] `http://pcj.icm.edu.pl`

[2] M. Nowicki, P. Bała. Parallel computations in Java with PCJ library In: W. W. Smari and V. Zeljkovic (Eds.) *2012 International Conference on High Performance Computing and Simulation (HPCS)*, IEEE 2012 pp. 381-387

[3] Java Platform, Standard Edition 6, Features and Enhancements http://www.oracle.com/technetwork/java/javase/features-141434.html

[4] B. Carpenter, V. Getov, G. Judd, T. Skjellum and G. Fox. MPJ: MPI-like Message Passing for Java. Concurrency: Practice and Experience, Volume 12, Number 11. September 2000

[5] J. Boner, E. Kuleshov. Clustering the Java virtual machine using aspect-oriented programming. In AOSD '07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development, 2007.

[6] Nester Ch., Philippsen M., and Haumacher B. A more efficient RMI for Java. In Proceedings of the ACM 1999 conference on Java Grande (JAVA '99). ACM, New York, NY, USA, 152-159. 1999

[7] D. Mallón, G. Taboada, C. Teijeiro, J.Tourino, B. Fraguela, A. Gómez, R. Doallo, J. Mourino. Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures In: M. Ropo, J. Westerholm, J. Dongarra (Eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface (*Lecture Notes in Computer Science 5759)* Springer Berlin / Heidelberg 2009, pp. 174-184

[8] R. W. Numrich, J. Reid. Co-array Fortran for parallel programming ACM SIGPLAN Fortran Forum Volume 17(2), pp. 1-31, 1998

[9] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, K. Warren. *Introduction to UPC and Language Specification* IDA Center for Computing 1999

[10] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A High-Performance Java Dialect Concurrency: Practice and Experience, Vol. 10, No. 11-13, September-November 1998.

[11] Java Grande Project: benchmark suite http://www.epcc.ed.ac.uk/research/-java-grande/